
Large and Cloud-based Software Systems

Lecture Notes

author: Prof. Dr. René Wörzberger
contact: rene.woerzberger@th-koeln.de
saved on 30 August 2026 18:12:39

© 2000 – 2026 René Wörzberger. All Rights Reserved.

Table of Contents

Large and Cloud-based Software Systems Lecture Notes.....	1
1 Introduction.....	15
1.1 Large Scale Systems.....	15
1.2 Topics and Structure.....	16
1.3 Disclaimer.....	17
1.3.1 The State of Software Engineering.....	17
1.3.2 Hype Cycles.....	17
1.3.3 The Pendulum of Trends.....	18
1.3.4 Metaphors.....	18
1.4 Teaching by Examples.....	18
1.5 Recurring Example System ReWo-Rate.....	18
1.6 Some Technicalities.....	19
2 Software Systems.....	20
2.1 The Nature of Software.....	20
2.2 Software Characteristics.....	20
2.3 Basic Concepts.....	22
2.3.1 Software.....	22
2.3.2 Code and Compilers.....	22
2.3.3 Phases.....	23
2.3.4 Machines.....	23
2.3.5 Operating Systems.....	23
2.3.6 Processes.....	24
2.3.7 Systems.....	24
2.3.8 Networks.....	24
2.3.9 Services.....	25
2.3.10 Applications.....	25
2.3.11 Middleware and Infrastructure.....	26
2.3.12 Client-server Systems.....	26
2.3.13 Messaging.....	26
I. Drivers.....	27
3 Stakeholders.....	28
3.1 Stakeholder Groups.....	28
3.2 Stakeholder Analysis.....	32
4 Quality Attributes.....	35
4.1 Requirements.....	35
4.1.1 Functional Requirements.....	36
4.1.2 Non-functional Requirements.....	36
4.2 Quality Attributes / Ilities.....	36
4.3 Constraints.....	41
5 Performance.....	42
5.1 Workloads and Hardware Limits.....	42

5.1.1 Processing Units (CPU, GPU).....	43
5.1.2 Memory.....	43
5.1.3 Storage.....	44
5.1.4 Network.....	44
5.1.5 Workload Bottlenecks.....	44
5.2 Network Latency and Response Time.....	46
5.2.1 Network Metrics.....	46
5.2.2 Network Latency.....	47
5.2.3 Impact of Response Time.....	48
5.2.4 Response Time Variance.....	48
5.2.5 Measuring Response Time.....	48
5.3 Throughput.....	49
5.3.1 System Throughput.....	50
5.3.2 Network Throughput.....	51
5.4 Utilization.....	52
6 Capacity.....	54
6.1 Database Capacity.....	54
6.2 Memory Capacity.....	55
6.3 Common Latencies, Throughputs and Capacities.....	56
7 Scalability.....	58
7.1 Transactional Requests Workload Profiles.....	58
7.2 Vertical and Horizontal Scaling.....	60
8 Reliability.....	61
8.1 Faults and Failures.....	61
8.2 Recoverability.....	62
8.3 Availability.....	62
8.3.1 Mean Availability.....	62
8.3.2 Nines.....	63
8.3.3 Combined Availability.....	63
8.3.4 Planned Downtimes.....	64
8.4 Faultlessness.....	64
8.5 Fault Tolerance / Resilience.....	65
8.5.1 Exception Handling.....	65
8.5.2 Failover for Parallel Resources.....	65
8.5.3 Fallback for Serial Resources.....	66
9 Consistency.....	67
9.1 CAP Theorem.....	67
9.2 PACELC Theorem.....	68
10 Maintainability.....	69
10.1 Analysability.....	69
10.1.1 Observability.....	69
10.1.2 Simplicity.....	71
10.2 Modifiability.....	72
11 Security.....	74
11.1 Aspects.....	74

11.2 Common Vulnerability Types and Attack Types.....	75
II. Design.....	77
12 Design Goals.....	78
12.1 Trading Goals.....	78
12.1.1 Interest Trade-offs.....	78
12.1.2 Quality Attribute Trade-offs.....	79
12.2 Rating Goals.....	80
13 Design Requirements.....	82
13.1 Quality Attribute Scenarios.....	82
13.2 Risks.....	85
14 Design Principles.....	86
14.1 Abstraction.....	86
14.2 Structuring.....	86
14.3 Locality.....	87
14.4 Strong Cohesion.....	87
14.5 Loose Coupling.....	88
14.5.1 Dependencies.....	89
14.5.2 Connascence.....	90
14.6 Information Hiding.....	92
15 Design Patterns.....	93
15.1 Patterns on Different Levels.....	93
15.2 Catalogues and Languages of Patterns.....	94
16 Modeling Systems.....	95
16.1 Models and Originals.....	95
16.2 Model Properties and Qualities.....	95
16.2.1 Precision of a Model.....	96
16.2.2 Scope of a Model.....	96
16.2.3 Unambiguousness of a Model.....	96
16.2.4 Correctness within a Model.....	97
16.2.5 Compliance between an Original and a Model.....	97
16.3 Decomposition.....	98
16.4 Viewpoints.....	99
16.5 The Unified Modeling Language (UML).....	101
16.5.1 UML Use Case Diagram.....	102
16.5.2 UML Activity Diagram.....	104
16.5.3 UML State Machine Diagram.....	106
16.5.4 UML Component Diagram.....	108
16.5.5 UML Class Diagram.....	110
16.5.6 UML Deployment Diagram.....	112
16.5.7 UML Sequence Diagram.....	114
16.6 The C4 Model.....	115
17 Software Architecture.....	118
17.1 Modularization in General.....	118

17.1.1 Modularize by Technical Responsibilities.....	118
17.1.2 Modularize by Functionality.....	118
17.2 Layered Architectures.....	119
17.3 Hexagonal Architectures.....	120
17.4 Pipes-and-Filters Architectures.....	120
17.5 Microkernel Architectures.....	121
18 Scaling Systems.....	122
18.1 Horizontal Scaling.....	122
18.2 Tiers.....	122
18.2.1 User agent separation.....	122
18.2.2 Standalone middleware.....	122
18.3 Tiers versus Layers.....	124
18.4 Node-to-Machine Allocation.....	124
18.4.1 Specialized Machines.....	125
18.4.2 Cookie-Cutter Scaling.....	125
18.4.3 Automated Allocation.....	125
18.5 Cell-based Architectures.....	126
18.6 Intra-Tier Communication.....	127
18.7 Redeployment of Tiers.....	128
19 Load Balancing.....	130
19.1 Load Balancing Goals.....	130
19.2 Load Balancing Strategies.....	130
19.2.1 Service-Discovery based Load Balancing.....	130
19.2.2 Dedicated Load Balancer.....	131
19.3 Load Balancing Algorithms.....	132
19.3.1 Static Load Balancing.....	132
19.3.2 Dynamic Load Balancing.....	135
19.4 Load Balancing Network Layer.....	137
20 Distributing Systems.....	139
20.1 Avoiding Distribution with Monoliths.....	139
20.2 Downsides of Monoliths.....	139
20.2.1 Organizational Downsides.....	139
20.2.2 Architectural and Technical Downsides.....	139
20.3 Distributed Systems.....	141
20.4 Architecture Styles for Distributed Systems.....	142
20.4.1 Accidental Distribution.....	142
20.4.2 Microservices.....	142
20.4.3 Event-driven Systems.....	144
20.4.4 Domain-driven Design.....	144
20.4.5 Client-Server Systems and Peer-to-Peer Systems.....	145
20.5 Downsides of Distributed Systems (Microservices).....	146
20.5.1 Increase of Complexity.....	146
20.5.2 Distribution of Systems.....	147
20.5.3 Increase in Overall Footprint.....	148
20.6 Redeployment in Distributed Systems.....	148

20.6.1 Release Trains.....	148
20.6.2 Keeping Backward Compatibility.....	149
21 Communication.....	151
21.1 Communication Directions.....	151
21.1.1 General Patterns.....	151
21.1.2 Pull and Push.....	152
21.1.3 Communication Origin.....	152
21.2 Communication States (Sessions).....	153
21.2.1 Hierarchy of States.....	153
21.2.2 Server-side Sessions.....	154
21.2.3 Sessions vs. Scalability.....	155
21.2.4 Streaming and Full-Duplex Communications vs Scalability.....	156
21.3 Communication Contents.....	156
21.3.1 Meaning of Messages.....	156
21.3.2 Encoding of Messages.....	156
21.4 Synchronization.....	156
21.4.1 Synchronous Communication.....	156
21.4.2 Asynchronous Communication.....	156
21.4.3 Implications for Programming.....	157
21.5 Messaging.....	160
21.6 Proxying.....	161
21.7 Flow Control.....	162
21.7.1 Limits for Scaling.....	162
21.7.2 Flow Control Mechanisms.....	162
21.7.3 Rate Limiting.....	163
21.7.4 Backpressure.....	165
22 Caching.....	167
22.1 Caches in General.....	167
22.2 Read Caches and Write Caches (Buffers).....	167
22.3 Use of Caching.....	167
22.3.1 Caches for Hardware Components.....	167
22.3.2 Network Caches.....	170
22.3.3 Materialized Views in Databases.....	170
22.3.4 In-Process Caches.....	170
22.4 Caching Goals.....	170
22.5 Private and Shared Caches.....	171
22.6 Population Policies.....	171
22.6.1 Cache Population on Writes.....	172
22.6.2 Cache Population on or Prior to Reads.....	172
22.7 Eviction Policies.....	172
22.8 Bloom Filters.....	174
III. Infrastructure.....	175
23 Middleware.....	176
23.1 Application Server.....	176
23.2 Client.....	177

23.2.1 Browsers.....	177
23.2.2 Specific User Agents.....	177
23.3 Inbound Gateway.....	178
23.3.1 Message Routing.....	178
23.3.2 Message Transformation.....	179
23.3.3 TLS Offloading.....	179
23.3.4 Authentication and Authorization.....	179
23.3.5 Network Intrusion Detection and Prevention.....	179
23.3.6 Rate Limiting.....	180
23.3.7 Usage Monitoring.....	180
23.3.8 Load Balancing.....	180
23.4 Database Management.....	180
23.5 Caching Service.....	180
23.6 Analytics / Data Warehouse.....	180
23.7 Partner Systems.....	180
23.8 Domain Name System (DNS).....	180
23.9 Web Servers / Content Delivery Network (CDN).....	182
23.10 Cloud Storage.....	183
23.11 Outbound Gateway.....	183
23.12 Identity Provider.....	183
23.13 Monitoring Server.....	184
23.14 CI/CD/MLOps Pipeline.....	184
23.15 Source Code Repository.....	185
24 Cloud Computing.....	186
24.1 Cloud Characteristics.....	186
24.2 Service Models.....	186
24.2.1 Infrastructure-as-a-Service (IaaS).....	188
24.2.2 Container-as-a-Service (CaaS).....	189
24.2.3 Platform-as-a-Service (PaaS) / Backend-as-a-Service (BaaS).....	189
24.2.4 Function-as-a-Service (FaaS).....	189
24.2.5 Serverless Computing.....	190
24.2.6 Software-as-a-Service.....	190
24.3 Deployment Models.....	190
24.4 Utilization and Costs.....	192
24.5 Cloud Services.....	193
24.6 Legal Considerations.....	194
24.7 Cloud Native.....	194
24.8 Zero Trust Architectures.....	195
24.9 Google Cloud.....	196

24.9.1 Resources.....	196
24.9.2 Projects.....	196
24.9.3 Administration.....	197
24.9.4 Regions and Zones.....	198
24.9.5 Pricing.....	199
24.9.6 Billing.....	199
24.9.7 Storage.....	200
24.9.8 A GCP-based System Cluster.....	201
24.9.9 Scaling SQL Instances.....	203
24.9.10 Database Failover.....	204
25 Virtualization.....	205
25.1 Kinds of Virtualization.....	205
25.1.1 Processes.....	205
25.1.2 Containers.....	206
25.1.3 Virtual Machines.....	206
25.1.4 Bare-Metal Machines.....	207
25.2 Virtualization Technologies Trade-offs.....	207
25.2.1 Isolation.....	207
25.2.2 Host Dependence.....	208
25.2.3 Waste.....	209
25.2.4 Provisioning.....	209
25.3 Container-based Virtualization.....	210
25.3.1 Isolation via Namespaces.....	210
25.3.2 Resource Sharing with Cgroups.....	211
25.3.3 Docker.....	211
25.4 Container Orchestration.....	216
25.4.1 Container Orchestration Features.....	216
25.4.2 Basic Kubernetes Concepts.....	217
25.5 Service Meshes.....	222
26 Messaging Systems.....	224
26.1 Message Brokers.....	225
26.1.1 Load-balancing and Fan-Out.....	225
26.1.2 Redelivery.....	226
26.1.3 Event Sourcing.....	226
26.2 Streaming Platforms.....	227
26.2.1 Complex Event Processing.....	227
26.2.2 Event Stream Processing.....	227
26.3 Scaling Messaging Systems.....	228
IV. Web Technology.....	229
27 Web Application Layer Protocols.....	230
27.1 Hypertext Transfer Protocol.....	230

27.1.1 Resources.....	230
27.1.2 Request and Response Messages.....	231
27.1.3 Uniform Resource Identifiers.....	231
27.1.4 Request Methods.....	232
27.1.5 HTTP Caching.....	233
27.1.6 HTTP 1.1 Shortcomings.....	236
27.1.7 HTTP/2 Multiplexing.....	236
27.1.8 HTTP/3: QUIC over UDP.....	237
27.1.9 HTTP Omissions.....	238
27.2 HTTP Streaming.....	238
27.3 Server-sent Events (SSE).....	238
27.3.1 SSE Protocol.....	239
27.3.2 SSE API.....	240
27.4 gRPC.....	240
27.4.1 Protocol Buffer Messages.....	240
27.4.2 Protocol Buffer Services.....	241
27.5 WebSockets.....	242
27.5.1 WebSocket Protocol.....	242
27.5.2 WebSocket API.....	242
27.6 Comparison of Protocols.....	244
28 Representational State Transfer (REST).....	247
28.1 Client-Server Architecture.....	247
28.2 Statelessness.....	247
28.3 Uniform Interface.....	247
28.3.1 Resource identification in requests.....	248
28.3.2 Resource manipulation.....	248
28.3.3 Self-descriptive messages.....	248
28.3.4 Hypermedia as the Engine of Application State (HATEOAS).....	248
28.4 Cacheability.....	250
28.5 Layered System.....	250
28.6 Richardson Maturity Model.....	250
29 API Access Standards.....	251
29.1 OpenAPI.....	251
29.2 GraphQL.....	253
29.3 Model Context Protocol (MCP).....	254
29.3.1 The Problem.....	254
29.3.2 Model Context Protocol Concepts.....	255
29.3.3 Transport Layer.....	255
29.3.4 MCP Server Capabilities.....	260
29.3.5 Comparison with HATEOAS.....	262
29.3.6 MCP Client Capabilities.....	262
30 Web Security Protocols.....	264
30.1 Web Security Threats.....	264
30.2 Cryptographic Basics.....	265

30.2.1 Hashing Algorithms.....	265
30.2.2 Secrets and Keys.....	266
30.2.3 Symmetric Ciphers.....	266
30.2.4 Asymmetric Ciphers.....	268
30.2.5 Key Derivation.....	272
30.2.6 Public Key Certificates.....	273
30.2.7 Certificate Chains.....	275
30.2.8 Certificate Formats.....	275
30.3 Transport Layer Security.....	276
30.3.1 Cipher Suites.....	276
30.3.2 TLS Handshake Protocol.....	277
30.3.3 TLS Record Protocol.....	279
30.4 OAuth2.....	279
30.4.1 Credentials and Security Tokens.....	280
30.4.2 Clients.....	280
30.4.3 Endpoints.....	280
30.4.4 Client Registration and Authentication.....	281
30.4.5 Authorization Code Grant Flow.....	282
30.4.6 Refreshing an Access Token.....	290
30.5 JSON Web Token (JWT).....	291
30.5.1 JOSE Header.....	291
30.5.2 JWT Claim Set.....	292
30.5.3 JWS Signature.....	292
30.6 OpenID Connect.....	293
30.6.1 User-info Endpoint.....	293
30.6.2 ID Token.....	293
30.6.3 OpenID Connect Authorization Code Grant Flow.....	294
V. Databases.....	297
31 Persistence in Databases.....	298
31.1 Databases and File Systems.....	298
31.2 Database Server.....	299
31.3 Database Storage.....	300
32 Database Paradigms.....	302
32.1 Data Model Criteria.....	302
32.1.1 Schematic or Non-schematic Data.....	302
32.1.2 Relationality.....	302
32.1.3 Locality.....	304
32.1.4 Query Languages.....	304
32.1.5 Impedance Match.....	304
32.2 Data Models.....	304
32.2.1 Key-value oriented.....	306
32.2.2 Wide-Column / Column-Family.....	307
32.2.3 Document-oriented.....	308
32.2.4 Relational.....	310
32.2.5 Graph.....	311
32.3 Derived Usages.....	313

32.3.1 Search & Analytics.....	313
32.3.2 Time Series.....	314
32.3.3 Vector.....	315
33 Consistency in Unscaled Databases.....	317
33.1 Denormalization Anomalies.....	317
33.1.1 Insertion Anomaly.....	318
33.1.2 Deletion Anomaly.....	318
33.1.3 Update Anomaly.....	318
33.2 Atomicity.....	318
33.3 Concurrent Access Anomalies.....	319
33.3.1 Dirty and Skewed Concurrent Access.....	319
33.3.2 Repetitive Concurrent Access.....	320
33.4 Isolation of Transactions.....	321
34 Scaling Databases.....	323
34.1 Partitioning.....	323
34.1.1 Vertical Partitioning.....	323
34.1.2 Horizontal Partitioning.....	323
34.2 Replication.....	324
34.2.1 Leader-follower Replication.....	324
34.2.2 Multi-Leader Replication.....	324
34.2.3 Leaderless Replication.....	325
34.3 Coordination and Load Balancing.....	325
35 Consistency in Scaled Databases.....	326
35.1 Consistency Levels.....	326
35.2 Data Versioning and Quorums.....	328
35.3 Consensus.....	329
35.3.1 Crash-Fault Tolerant.....	329
35.3.2 Byzantine-Fault Tolerant Consensus.....	330
35.4 Conflict Resolution.....	330
35.4.1 State-based CRDT: Convergent Replicated Data Type (CvRDT).....	332
35.4.2 Operation-based Commutative Replicated Data Type (CmRDT).....	334
35.4.3 Automerger.....	335
36 Consistency in Distributed Databases.....	336
36.1 Distributed Database Transactions.....	336
36.1.1 Upsides of DBMS Managed DDTs and Downsides of Client Managed DDTs.....	336
36.1.2 Downsides of DBMS Managed DDTs and Upsides of Client Managed DDTs.....	337
36.2 DBMS Managed Distributed Database Transactions.....	337
36.2.1 Two-phase Commit (2PC).....	337
36.2.2 XA Interface for Distributed Transaction Processing.....	337
36.2.3 Drawbacks of 2PC and XA Interface.....	338
36.3 Client Managed Distributed Database Transactions.....	338

36.3.1 Commutative Transactions.....	338
36.3.2 Reread Values.....	338
36.3.3 Semantic Locks.....	338
36.3.4 Compensation Transactions.....	338
36.3.5 Sagas.....	340
37 References.....	344

1 Introduction

Software systems vary in complexity. For such systems, complexity comes in many forms: the overall size of the code base, the number processes involved that may be distributed among machines in a network, the number of different stakeholders with different interests or concerns about the system, quality attributes that have to be traded off against each other and so on.

1.1 Large Scale Systems

A very simple software system is the well-known "Hello World" system. The only function of "Hello World" is to print out "Hello World" to a console and then exit. Typically, the code base consists of less than 10 lines of code, the only stakeholder is the developer whose only "interest" is to make some first steps in a new programming language, and quality attributes such as reliability or maintainability are of diminishing importance.

At the other end of the spectrum, there are complex systems¹ such as those used by customers of Google, Amazon, or Facebook. Such systems are highly distributed, have a huge code base, many stakeholders with different interests or concerns about the respective system and a plethora of quality requirements to consider.

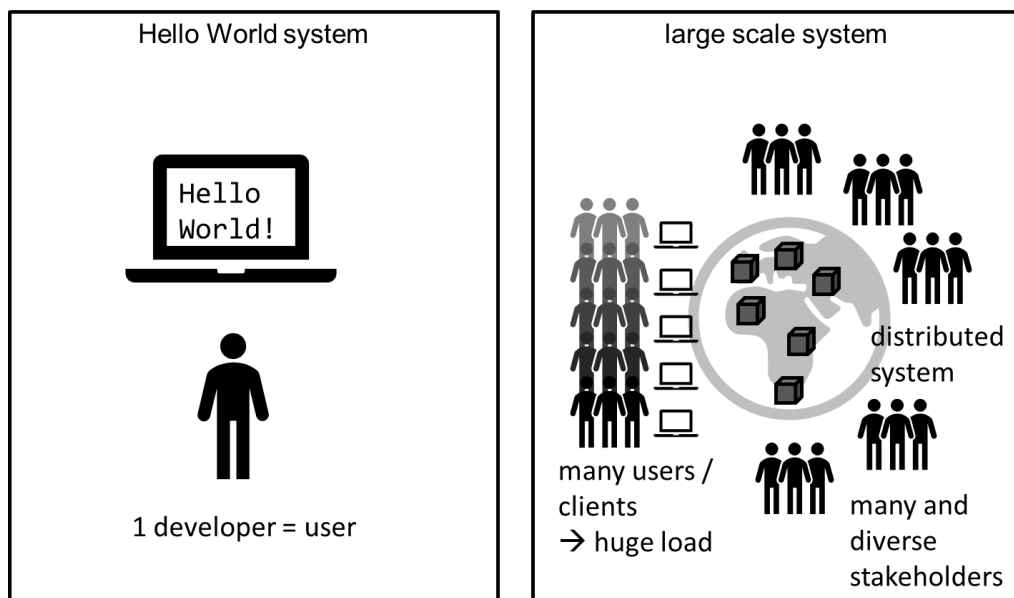


Figure 1: Hello World system vs. large-scale system

These lecture notes for the course "Large and Cloud-based Software Systems" focus on software systems at the complex end of the spectrum, especially with a view to operating them in the so-called cloud. Although the fact that our normal altitude is above introductory bachelor courses, we sometimes delve into technical details to make insights tangible.

¹ For the time being, we do not distinguish between a non-distributed monolithic system and a distributed system.

1.2 Topics and Structure

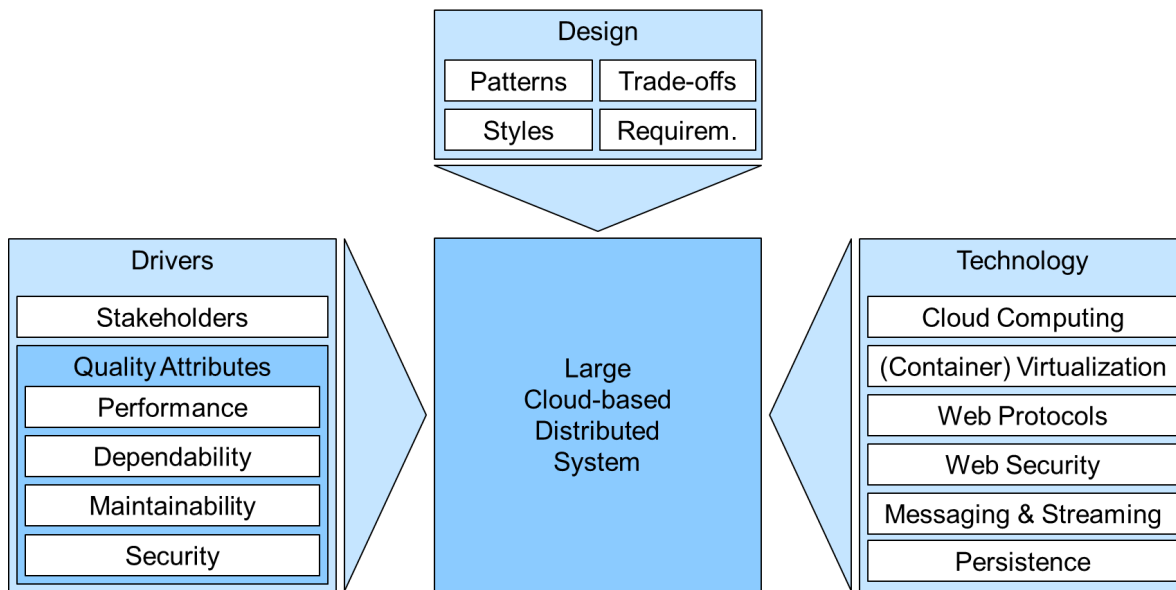


Figure 2: Course contents

The contents of this course are depicted in Figure 2 and seek answers to the following questions:

- **Drivers**
 - Software Systems: definitions and some basic terms
 - Stakeholders: The whole plethora of stakeholder groups and their numerous interests in large projects.
 - Quality Attributes
 - Performance: What makes my system slow? What workload do I have to deal with? Why is latency often a bigger problem than throughput? Why is it a bad idea to fully utilize my system? How can I scale my system?
 - Reliability: How can I increase my system's availability? How can I make my system resilient to faults? Why do I have to trade off availability against consistency and vice versa?
 - Quality Attribute Maintainability: What is maintainability? What fundamental cognitive mechanism drives the perception of a code base's maintainability?
 - Security: How can I describe security properties? What are the aspects of security? What are common threats for (web based) systems?
- **Design**
 - Trade-offs: Why is there no silver bullet?
 - Requirements: How can I formulate quality requirements?
 - Principles: Which fundamental principles govern architectural design?
 - Patterns: How should I distribute my system? How do systems and their parts communicate?
- **Technology**
 - Infrastructure and Middleware: What standard components constitute a contemporary large cloud-based web application? What are load balancers, inbound gateways, caching services, data warehouses, message queues, identity providers etc.?

- Cloud Computing: What kinds of cloud offerings do we have to deal with? How do we create an infrastructure in the cloud for a large-scale web application?
- Virtualization: What are virtual machines and containers, how do they differ and what are the trade-offs?
- Web: How do HTTP as the fundamental protocol of the world-wide web, its applications like GraphQL, and related protocols like WebSockets and gRPC work? How can web applications be secured by the Transport Layer Security (TLS) for encryption and server (and client) authentication, OAuth 2 for authorization, and OpenID Connect for user authentication?
- Messaging and Streaming: What do message queues, brokers and streaming platforms like Apache Kafka do?
- Web Security: How to make communication and interaction in web applications confidential, authentic, and integrous? What are common threats and methods? What are hash functions, symmetric and asymmetric ciphers? What is encryption and signing? How do Transport Layer Security (TLS), Open Authorization (OAuth2), and OpenID Connect (OIDC) work?
- Persistence: What kind of (No)SQL databases do we have and what are their trade-offs (relational, document, key-value, graph)? How does scaling work in the database world? How much consistency do I need?

1.3 Disclaimer

These lecture notes are work in progress and have not undergone any official review or proofreading process. I apologize for any errors or current inaccuracies. Any reader is invited to notify the author of typographical errors, obvious mistakes, or divergent opinions.

1.3.1 The State of Software Engineering

Speaking of "opinions": Because software engineering is still a young profession, there is an ongoing debate about which findings are reliable. Bossavit argues in [1, pp. 3–10] that many alleged facts about software engineering are mere opinions, beliefs, tacit agreements, folklore, or exaggerations of scientific evidence taken out of context and modality. In contrast, Martin points out in [2, p. 33] that opposing the currently accepted state of science sometimes leads to progress despite a lack of tangible evidence.

Both have a point: Software engineering is still in the age of alchemy, not chemistry. Software is intangible and not subject to physical laws. Instead, cognitive and sociological effects play an important role in the creation and maintenance of software. Disputes in the development community are often driven by the urge of opponents to save or increase learning investment. However, trading on the effects of new tools, frameworks, methods, techniques, or paradigms without hard evidence is the best we can do for now; yet more rigor would surely improve the state of software engineering.

1.3.2 Hype Cycles

Partly because of the lack of scientific rigor, software engineering is susceptible to trends, hypes, cargo cults, and fashions that often recur under different names which makes it hard to tell "really novel" and "just new" apart. Moreover, the belief in the (positive) effects of a novel paradigm, method or technology often reaches an early peak of "inflated expectations", fueled by success stories from developer conferences, blogs, and the like. These beliefs are less real evidence and more likely to contribute to a survivorship bias. Later, they fall into a "trough of disillusionment" before reaching a long-lasting "plateau of productivity", as the Gartner Model for Hype Cycles [3]

puts it. The one for application architecture and development for the year 2019 can be found in [4].

1.3.3 The Pendulum of Trends

In addition, some trends oscillate between extremes, such as

- whether computation and storage should reside on the client side or the server/cloud side,
- whether a software system is better be broken down into multiple microservices or better be kept in a structured monolith, or
- whether development projects should be carried out in a fully agile mode or in a series of phases,

to name but a few. Some of the methods and technologies are also the subject of these lecture notes. I do my best to focus on the facts and take a neutral stance on unproven positive or negative effects.

1.3.4 Metaphors

Another point worthy of a disclaimer is that software engineering makes extensive use of metaphors such as "containers" and models such as the relational data model. These are rather representations of abstract concepts, intended for reasoning about the subject. Please do not take a metaphor or model as "the" truth. A model cannot be true or false, but only helpful or not, or at worst misleading.

1.4 Teaching by Examples

I assume that it is easier for readers to abstract and extrapolate a concept from concrete examples than to interpret a purely abstract description. My own experience tells me that concrete examples are also very helpful to memorize a concept (cf. Section 10.1.2b) about Schema Building)

Therefore, I use many examples in these lecture notes which are typographically highlighted as the following (meta) example.

EXAMPLE 1: META-EXAMPLE

This is an example of an example in these lecture notes.

The main purpose of most examples in this document is not that you understand every detail of them. Instead, you are supposed to get an impression, how a certain technology, configuration language, modeling language etc. actually looks like and help to memorize it.

1.5 Recurring Example System ReWo-Rate

Some of the examples recur to a hypothetical system called ReWo-Rate that touches on many of the aspects that we deal with in this document.

EXAMPLE 2: RECURRING EXAMPLE SYSTEM "ReWo-RATE"

You are the Chief IT Architect at ReWo-IT, a medium-sized e-commerce company. Until now, this company has focused on building and operating web shops. Over the years, ReWo-IT has developed a product called ReWo-Shop, which is a customizable web shop system. Shop owners (sellers) can either buy a license for ReWo-Shop and run a customized instance, e.g., with their own product catalogue, corporate identity, and domain name, on their own infrastructure, in the cloud, or use the software-as-a-service offering for ReWo-Shop.

One day the CEO approaches you with the following idea: *"Many of our customers sell identical products. You know, we have this product rating component in ReWo-Shop. A visitor to any shop can rate a product after buying it. A single review is currently only visible within a single shop. Ratings for the same product are therefore scattered across shop instances. Buyers, i.e., the customers of our clients, do not put much faith in an overall rating of a product if it is backed up by only a few actual ratings. So, let's build a more centralized rating system, ReWo-Rate, where all ratings are stored, or at least those that are triggered by transactions of ReWo-Shop instances."*

Inevitably, the following questions come to your mind: *"How do we know which products are the same in different shops? How do we integrate the rating with ReWo-Shop or other web shop systems? How can we keep response times low and availability high? Is consistency important or can we live with slightly different ratings? What privacy constraints do we need to consider? Should we build for the cloud? NoSQL database, at last?"*

1.6 Some Technicalities

This document was initially written with Microsoft Word and later with LibreOffice Writer. References are managed with Zotero. The main font is "Harding Text Web Regular" made by webfontfree.com and licensed by CC 4.0 BY.

2 Software Systems

It is not easy to define what we talk about in these lecture notes: software systems. Therefore, we start by delineating and characterizing these sorts of systems.

2.1 The Nature of Software

Some say that programming is a kind of art, others that computer science is a branch of science or mathematics. Most people think of software as an engineering product.

Non-software engineering products, like cars, planes, drilling machines, houses, bridges etc are different from software. The following properties show inasmuch software is different from other engineering products:

- **Physical laws and limits:** Software is immaterial and therefore not affected by physical laws (cannot collapse under gravity, is not subject of mechanical attrition, etc.)
- **Complexity and Resilience:** Software systems are comparably complex as they can be in a vast number of states with even more cause-effect relationships between them and are therefore fragile. Every unforeseen input or fault may cause a complete failure. Moreover, software is always highly interdependent with other software that is developed independently. Such dependencies often rely on assumptions that are prone to break when software evolves.
- **Model and Plan:** Software is comparably hard to envision and to model.
- **Modifications:** Software is comparably easy to change.
- **Production and logistics costs:** Software is comparably easy and cheap to reproduce and store.

2.2 Software Characteristics

Software is ubiquitous. However, a PlayStation game is surely not from the same type as an SAP application. There are plenty of properties that shape the "character" of a software system.

EXAMPLE 3: ENTERTAINMENT SOFTWARE

A PlayStation game is an instance of entertainment software as opposed to business software.

There are many ways to characterize different types of software like Facebook, BitTorrent, MS Excel, Fortnite, Bitcoin, Netflix, Linux, DSL router software, weather simulation, or SAP applications. Table 1 provides one way to categorize these kinds of software. In that table, types of software systems are characterized.

- whether they are **embedded** or not into a hardware system (e.g., car). Embedded systems differ from non-embedded system typically in that
 - they have a very specific purpose, e.g., a user cannot easily install and run an arbitrary program,
 - that purpose is often not to interact with a user directly via a user interface but to monitor and control a surrounding system via sensors and actors, or to implement support functions like filtering, en- or decoding, or de- or encrypting data,
 - that the hardware resources are comparably limited in terms of computing power and memory due to costs, resilience against environmental stresses (temperature, humidity etc.), and limited power supply.

- if they have to respond in **real-time**, e.g., with a strict upper limit, which is often true for embedded system and which rules out normal operating systems but calls for a real-time operating system (RTOS)
- if and how they are **distributed**,
- if they focus on complex and vast **computation** tasks or **data**,
- if they can be directly used as **applications** by users or serve as bare **middleware**
- which **user groups** they target, and
- if or if not, their complexity mainly stems from a broad **functional scope** with complex and change-prone business logic, diverse and sometimes incoherent functions, entity types, user roles etc., which target different concerns.

Example	Embedding	Real-time	Distribution	Computation vs. data	Application middleware	User group	Functional scope
Facebook	non-embedded	non-realtime	client-server	data	application	private	medium
Bittorrent	non-embedded	non-realtime	peer-to-peer	data	application	private	narrow
MS Excel	non-embedded	non-realtime	local	data	application	business	narrow
Fortnite	non-embedded	non-realtime	local / client-server	-	application	private	narrow
Bitcoin	non-embedded	non-realtime	peer-to-peer	computation	application	private	narrow
Netflix	non-embedded	non-realtime	client-server	data	application	private	medium
Linux	all	non-realtime	local	-	middleware	all	-
DSL router Software	embedded	non-realtime	local	data	application	all	narrow
Engine control software	embedded	realtime	local	-	-	all	narrow
weather simulation	non-embedded	non-realtime	Local	computation	application	business	narrow
SAP application	non-embedded	non-realtime	client-server	data	application	business	often broad
ChatGPT	non-embedded	non-realtime	client-server	computation	Application	all	-
ReWo-Rate	non-embedded	non-realtime	client-server	data	application	business	medium

Table 1: Categories for software

In these lecture notes, we mainly focus on non-embedded, distributed and data-oriented applications like Netflix, Facebook, or our ReWo-Rate.

2.3 Basic Concepts

Of course, these lecture notes contain a lot of technical concepts and terminology. We introduce them as needed. However, there are some foundational concepts like “service” that are used with different meanings in literature and technology jargon. We therefore define and disambiguate these basic concepts in the following.

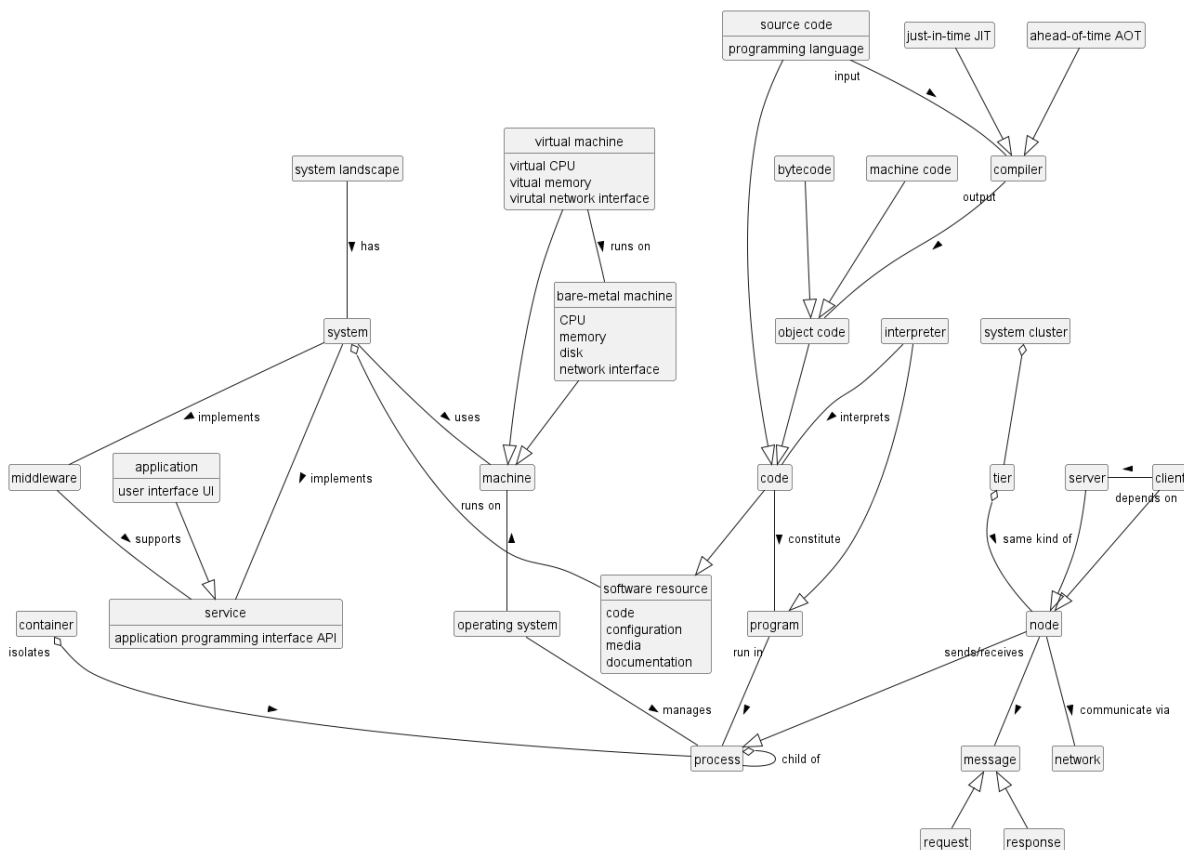


Figure 3: Map of basic concepts

2.3.1 Software

Software comprises **software resources** (a.k.a. **artifacts**). These software resources contain units of **code**, which constitute programs and libraries. A **program** has an entry point—often referred to as the **main method**—that is used by an operating system to start a program. A **library** lacks such an entry point but has entry points that can be invoked by other libraries and programs. Software resources also may contain configuration files, documentation etc.

2.3.2 Code and Compilers

Human developers edit code in its human-readable **source code** form normally written in a **high-level programming language** like Java, C, Python or Rust. A program called **compiler** translates code from higher to lower-level languages. **Source code** get translated by a **compiler frontend** to **intermediate languages** like Java Virtual Machine instructions (bytecodes) [5] or LLVM intermediate representation [6], which is still independent of a concrete CPU architecture. In another compilation step, a **compiler backend** translates intermediate representations to **object**

code, which is machine (architecture) dependent [machine code](#)². An [ahead-of-time \(AOT\) compiler](#) translates all code units of a program before its execution. A [just-in-time \(JIT\) compiler](#) translates code units right before their respective execution.

Only machine code can be directly executed on a machine. Source code or bytecode need a program called [interpreter](#) to be executed as-is. A [hotspot compiler](#) is both an interpreter and compiler. It either interprets or just-in-time compiles and then executes code units. The latter is faster for frequently executed code units, so-called hotspots, e.g., statements in loops.

[Object code files](#) usually contain references to other object code files from, e.g., external [libraries](#). In case of machine code, a program called [linker](#) cares for resolving these static references at build-time and produces another library object file or an [executable object file](#)³, also known as program [image](#). In case of bytecode, a program called (class) [loader](#) cares for resolving these references at runtime.

2.3.3 Phases

Editing source code, compiling, and linking are (iterative) phases of software development. We also speak of [editing time](#), [compile time](#) or [link time](#) to denote these phases. With [build time](#), we subsume all these phases that might vary depending on the programming language⁴ but all take place before actually starting a program. With [runtime](#), we mean all phases that come after this event.

2.3.4 Machines

Programs run on machines, a.k.a. computers. A [machine](#) is an assembly of resources like central processing units (CPU), graphics processing units (GPU), memory, network interfaces, or storage which processes can access by means of the operating system. A machine might be [bare-metal hardware](#) containing real [hardware resources](#) or virtualized with [virtual resources](#), i.e., being a [virtual machine](#). A machine that runs virtual machines is also called [host](#) with the virtual machines being its [guests](#).

2.3.5 Operating Systems

Operating system kernels provide an application binary interface (ABI) for programs. The constituents of this interface are called [system calls](#) (short: [syscalls](#))⁵. System calls provides means for (1) file management, e.g., open/create and close files and directories and write to or read from files, (2) device management, e.g., accessing a device's buffer or attaching or releasing a device like a video card or network interface, (3) information management, e.g., providing the system time to

² Multi-pass compilers typically transform source code in a series of phases to object code. These phases belong to the frontend or backend of a compiler, where the frontend produces code in an [intermediate representation](#), which independent of the source code programming language as well as the processor architecture on which a program is supposed to run. Given n different programming languages and m processor architectures, one has to implement $n + m$ front ends and back ends instead of $n * m$ different monolithic compilers. Moreover, since the final machine code consists of binary encoded instructions, so-called op(eration) codes, and is completely unreadable for humans, the compiler output is often presented in a processor architecture specific [assembly language](#), which provides some basic abstractions like mnemonics instead of numeric operation codes or symbolic names for memory addresses. The Compiler Explorer [7] is a free tool that shows the relationships between source code, assembly, and machine code.

³ This file then usually conforms to a file format like "Executable and Linking Format (ELF)" [8] for Linux or "Portable Executable (PE)" [9] for Windows.

⁴ For example, JavaScript need not be compiled but can be interpreted in its source code form.

⁵ As by the Linux manual page for syscalls, there are 467 such system calls.

programs, as well as (4) process control, e.g., creating processes and (5) communication between these processes.

2.3.6 Processes

A process is abstraction provided by an operating system kernel and serves as a **runtime environment** for running programs.

Each process allocates its own and isolated **virtual memory** address space. The virtual memory address space contains different **sections**: (1) a **text section** with the program (in compiled machine code), (2) a **data section** for global variables, (3) a **heap section** for dynamic data structures, and (4) a **stack section** for stack frames that store local variables and parameters for function calls [10, pp. 104–105].

Processes of a single machine are organized in a **process tree**: A program running in a process can create child processes via a system call, which copies⁶ of the parent process' data described above. The child process may then load another executable and run that and thereby reinitialize its memory sections.

If child processes just have their own program counter and stack section but share everything else with their parent process, these “lightweight processes” are more specifically called **native threads**⁷.

Operating systems kernels run their own **kernel processes** with privileged rights. Such **privileges** include accessing files on disk or devices, creation of processes etc. In contrast, programs such as word or spreadsheet applications or command line tools, which are either shipped as part of the **operating system distribution** or installed afterwards from third-party websites, run in non-privileged **user space processes**. These user space processes might do a **context switch** to the privileged **kernel mode** just during a system call in order to, e.g., obtain access to a file on a disk.

2.3.7 Systems

A **system** is the entirety of its **resources**, which might be software resources or hardware resources like machines, storage and network infrastructure that realize the respective application. If a system contains software resources, we call it more precisely a **software system** in order to distinguish it from other types of systems. The set of a single company's systems is called **system landscape**.

2.3.8 Networks

In a network, processes can communicate with processes on other machines via a network. In the following, we generally call a process that is addressable⁸ in a network and communicates with other nodes just by network protocols a **node** in that network. Please note that in the following we are just interested in application layer network protocols [11, p. 32], whose communicating participants, i.e., nodes, are always processes and not some low-level hardware network.

⁶ Actually, the virtual memory is not really copied on child process creation. Only if a child process writes to virtual memory the respective portion of that memory is copied and then modified. This is also called “**copy on write**”.

⁷ One could also say that every process has at least one “main” thread since you can consider the stack section and current activity the defining data structures of a thread and each process contains at least one of these.

⁸ In an internet protocol (IP) based network, processes are addressable by an IP address plus a target port number.

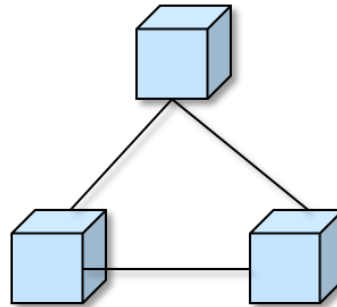


Figure 4: Nodes in a network

Nodes that run on the same (virtual) machine are **local** to each other. They can communicate via inter-process communication or network protocols via a virtual network interface called loopback device. If nodes are on different (virtual) machines, they are **remote** to each other.

In the following, we use the term node deliberately ambiguous: Sometimes, processes in a process tree interact in concert with one another and act as a single node to remote nodes. We then consider such a whole process tree to be a single node⁹. It turns out that such process trees are often isolated from each other using containers (cf. Section 25.1.2), in which call a whole container to be a node in a network.

If a system contains two or more nodes that communicate with each other, we have a **distributed system**. In a network, a node might be a **sender** which sends data in **messages**¹⁰ to a **receiver**. The flow of messages in a network is often referred to as (network) **traffic**. From the viewpoint of a single node, received messages constitute the **ingress** traffic and sent messages the **egress** traffic.



Figure 5: Sender, message, and receiver

2.3.9 Services

A **service** is a system that exposes an **application programming interface (API)** which can be called by other (remote) processes. If that API can be reached via the World Wide Web, e.g., via the HTTP protocol over the internet, we also speak of **web APIs**.

2.3.10 Applications

If a system provides a **user interface (UI)** and thus is useful for end-users, it is also called **application**. User interface might either be **graphical user interfaces (GUIs)** or **command line interfaces (CLIs)**. If a web service exposes a **web user interface (UI)** for browsers or a **web application programming interface (API)** for other user agents, e.g., native smartphone apps, it is called a **web application**.

⁹ This is the case for, e.g., certain database or web servers, where the root process spawns child processes for each client and for background jobs.

¹⁰ Messages are "meaningful and complete" data records for the receiver of arbitrary size, i.e., a program on the receiver is able to process it. Nonetheless, messages could be broken up in fixed-sized **packages** by the sender on the transport layer and reassembled on the receiver side.

2.3.11 Middleware and Infrastructure

Programs and resources that just support arbitrary other systems, e.g., by implementing certain technical and crosscutting functions like message queues, but provides no direct use for end-users by itself, are called **infrastructure or middleware**.

2.3.12 Client-server Systems

A **client-server system** is a distributed system, where a **client** node depends on a **server** node in order to work as specified but not the other way around. The dependency is usually of that form that the server node is able to look up or compute data that the client needs. If the client directly interacts with a user via a user interface (UI), it is also called a **user agent**.

Web applications are realized by client-server systems, where clients are user agents like browsers¹¹, smartphone apps or other external systems. If the server-side in a client-server system has two or more nodes, we call these nodes a **system cluster**. Please note that two nodes in such a server-side cluster might again form a client-server (sub-)system. Moreover, nodes that run the same programs might be scaled out thus forming a, e.g., **application server tier** or **database server tier**.

In a **client-server system** the **client** usually¹² sends a message called **request** to a node called **server**, which replies with a **response**.

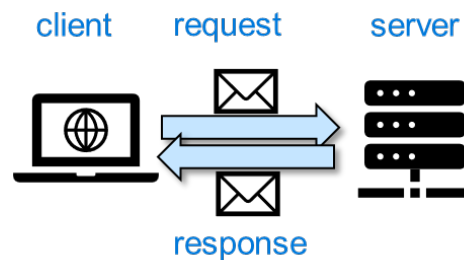


Figure 6: Client, request / response, and server

2.3.13 Messaging

Sometimes, sender and receiver communicate via **messaging system** (cf. Chapter 25). Since messages often contain data related to an **event** that took place in the sender and this event message might be received by many receivers, sender and receiver are usually called **publisher** and **subscriber**, respectively, in such scenarios.

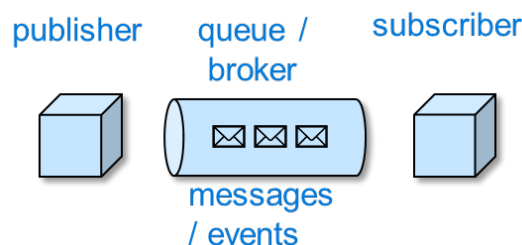


Figure 7: Publisher, queue, and subscriber

¹¹ In this case the "program" running on the client is the browser itself that interprets and renders web responses from the server or some JavaScript that has been transferred in a response from the server.

¹² The request-response communication pattern is not restricted to client-server systems nor the other way around. Yet, both often come together. Less common client-server communication patterns are discussed in Section 20.6.

I. Drivers

3 Stakeholders

Stakeholders in a software development process are people that have "stakes" in the process' execution or outcome. They have **interests** or, conversely, **concerns**. Stakeholders that share the same concerns due to common responsibilities or job profiles are in the same **stakeholder group**.

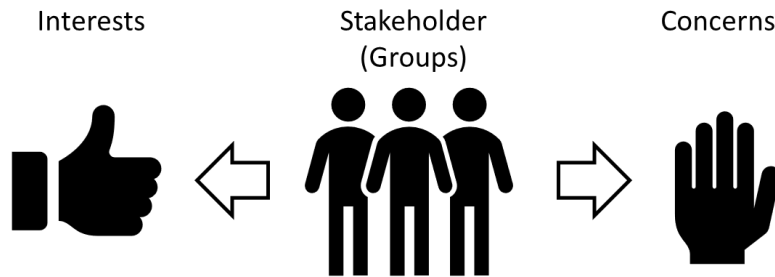


Figure 8: Stakeholders, interests and concerns

3.1 Stakeholder Groups

Developers and users are those stakeholder groups in software product and development processes which most people have in mind. However, they are by far not the only one.

EXAMPLE 4: STAKE HOLDER GROUP "FIRST LEVEL SUPPORT"

The stakeholder group "first level support" is concerned with whether a certain user interaction can easily be retraced in a support case.

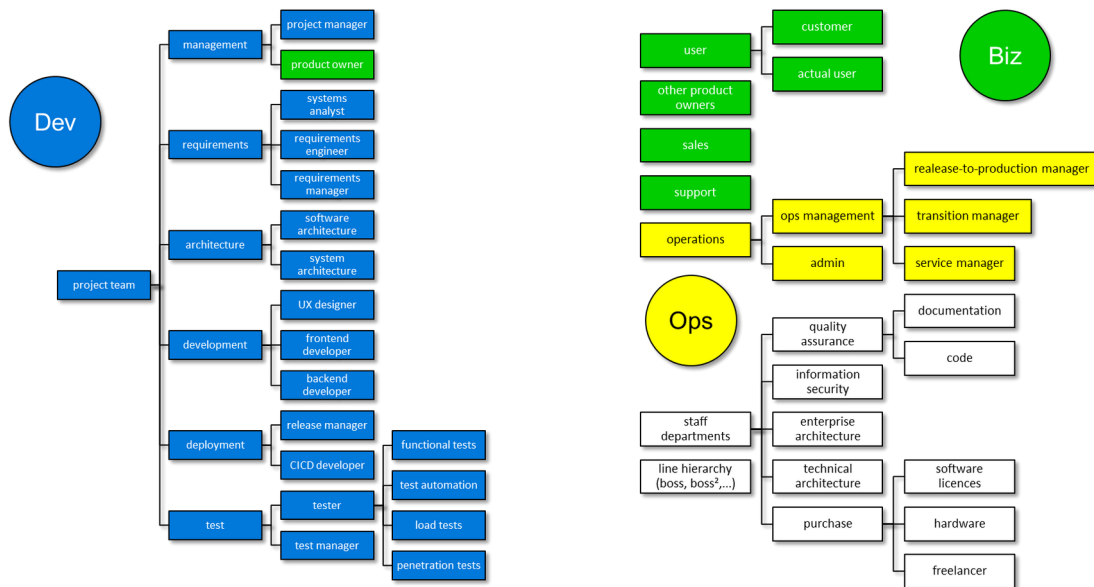


Figure 9: A taxonomy for roles in a development project

Figure 9 depicts stakeholder groups that you may encounter if you participate in a development project in a bigger enterprise. The leaves in the taxonomy are **roles** which are not necessarily carried out by exactly one person.

In agile development, teams act mainly autonomously. To fill out the roles, each team member typically assumes multiple roles, particularly those from Dev and Ops. This makes them **generalists** who are committed to fulfilling the promise "You build it, you run it", but in turn can

focus just on one development project. In classic development, there are **specialists**, who do not do anything but activities from a certain role, but for different development projects in parallel.

Anyway, roles are typically bound to certain interests and concerns and are therefore treated as equivalent with stakeholder groups in the following.

Stakeholder group	Responsibilities, interests, and concerns
project manager	The project manager is responsible for keeping a project in time, budget and quality, which are his major concerns.
product owner	The product owner represents the system under development towards the business. He or she is also responsible for maintaining the product's requirements. His or her main interests and concerns are about the final product and its success. He or she is often less concerned about time and budget.
application owner	An application owner is responsible for managing processes that revolve around a single application, which is normally implemented in a single system. These are, e.g., managing maintenance processes (like library updates). If a certain project mainly concerns a single application, he or she often also has the role of the project manager. The interest and concerns of an application owner are in a way all of those of Chapter 4 pertaining his or her application.
systems analyst	The systems analyst is mostly responsible for eliciting requirements and upholding the quality of requirements in terms of unambiguity, completeness, precision etc. Systems analysts are primarily interested in having all stakeholders finally agree on the contents in requirements even if their particular view on a business domain is not entirely reflected in these.
requirements engineer	Same as systems analyst.
requirements manager	Requirements in a development project are a moving target. They are in- or out-scoped, re-prioritized, elaborated, rephrased etc. A requirements manager keeps track of the current state of requirements, e.g., with the help of tools like JIRA or HP QC, but does not participate in actual requirements gathering nor elicitation (as opposed to a requirements engineer). Requirements managers are mainly interested in that all stakeholders agree on a current set of requirements, while they are a little less interested in the actual contents of requirements.
software architect	A software architect focuses on the architecture of software, which is mainly reflected in the structure of the source code and other software artifacts. He or she decides about the use of architectural styles and patterns like a 3-layered-architecture and governs the implementations of the design decisions. Software architects are concerned about architecturally relevant requirements that directly target their system, maintainability in particular. As opposed to system architects they are less concerned about the overall, e.g., performance of systems that contribute to a certain use case.
systems architect	A system architect mostly focuses on designing system clusters with resources of different kinds (cf. Chapter 23), and with integrating different systems, particularly negotiating interfaces between systems. He or she is concerned with whether these systems properly work together but a little less concerned about the inner architecture of software artifacts (as opposed to a software architect).
UX designer	A user experience (UX) designer is mostly concerned about the usability and look & feel of an application, e.g., if labels of input fields a better placed left to or above the input field.

Stakeholder group	Responsibilities, interests, and concerns
	He or she usually does not code but use mocking tools to design the user interface of a system.
frontend developer	A frontend developer focuses on the implementation of the user interface of a system. In case of a web application, the frontend developer has advanced skills in technologies like HTML, CSS, JavaScript and associated tools and frameworks.
backend developer	A backend developer focuses on the business logic and persistence in case of a web application. He or she normally has advanced skills in Java and SQL.
release manager	The release manager is part of the development team. He or she is responsible for the proper delivery of a software release, e.g., that exactly those changes are included in a release that have been ordered. Particularly, a release manager keeps an eye on the release state if last-minute changes have to be included in a release.
CICD developer	A CICD developer is responsible for the creation and maintenance of a continuous integration and delivery (CICD) pipeline. These pipelines are supposed to automate the repetitive tasks of software build, integration, test and delivery.
tester for functional tests	A tester for functional tests defines test plans based on requirements and manually executes them while recording if a software release behaves as expected or not. Nowadays, re-execution of test cases for functions introduced on prior releases, i.e., regression tests, is often automated. Therefore, testers for functional tests focus on progression tests, i.e., test for functions that were newly introduced in the current release.
test automation developer	Some specialists particularly focus on automation of tests, mostly regression tests. This can be done with unit test tools like Junit, which interfaces with the code, but also with tools like Selenium, which access the user interface.
load test tester	Load test testers focus on the non-functional quality attribute "performance". They use tools like Apache JMeter to generate workload for a system and measuring, e.g., response times.
penetration test tester	Penetration tests (or short: pentest) testers simulate attacks on a system thus testing its security.
test manager	Like a requirements manager, a test manager keeps track of existing and relevant test cases and their execution w.r.t. to a specific release. He or she is responsible for reporting testing progress to the project manager and endorses or advises against deploying a release to production.
customers	A customer might be a real person or a legal entity that pays for a certain software or services provided by a certain software system.
actual users	An actual user is a real person that uses a certain application. In a business setting actual users might be employees of a "customer", which is then a legal entity, e.g., a company.
other product owners	In a development project of software system x, product owners of those systems are important stakeholders, which have dependencies with x. Sometimes these systems have to implement minor changes because of x, so these product owners have at least an interest in getting a share of system x' budget.
sales department	The sales department is responsible for selling a software product or the services that are accessible by means of a software system. Bigger customers are assigned to dedicated members of the sales department, also known as key account managers.

Stakeholder group	Responsibilities, interests, and concerns
	These stakeholders are mainly concerned about customer satisfaction though not necessarily user satisfaction.
support department	The support department solves problems that arise on the customer's side. These stakeholders are interested in logging and tracing functions that allow for retracing user interactions with the system.
release-to-production manager	A release to production manager manages the transition of a release into production. He or she normally is employed in the company that is responsible for the operations of system and keeps track of availability of key resources (admins) during a release. His or her primary concerns are whether a release can smoothly be deployed into production.
transition manager	Same as release-to-production manager, yet he or she is usually employed at the company that owns the system.
administrator	An admin monitors the health of systems, installs new release and sets up environments.
documentation quality assurance	Bigger enterprises have a dedicated department that reviews system and project documentation. They mainly focus on the apparent completeness of documents and a little less on their inner quality because judging about the correctness of a document's contents would require too much insight into the respective system.
code quality assurance specialists	Another department in large companies cares for the quality of the source. These stakeholders use tools like Sonarqube in order to measure, record, report and archive performance of development teams in this regard with hindsight to maintainability of a system.
information security department	Members of an information security department usually have a high standing in bigger enterprises, which cannot afford reputation losses due to security incidents. These stakeholders demand extensive penetration test of systems before each release.
technical architecture group	Technical architects are concerned about which third-party resources (tools, libraries, and frameworks) are used in the overall system landscape. They make prescriptions or at least endorsements and keep track of the support of software versions, e.g., Java 8.
enterprise architecture	Enterprise architects deal with the architecture of the overall system landscape. They are concerned about which business capabilities are supposed to be supported by which (new) system but are not interested in technical details about each system's implementation.
purchase	The purchase department is responsible for acquiring resources, i.e., software licenses, hardware, or temporary human resources (freelancers). Their main concern is about the cost of these resources.
boss(es)	Almost every stakeholder has a boss. They often come into play if severe conflicts arise between stakeholders. Their major concern is whether their subordinates perform well.

Table 2: Stakeholder groups

Particularly in agile requirements engineering, one often talks about personas instead of stakeholder groups. **Personas** are concrete yet fictive and stereotypical representatives of some stakeholder group. Such personas are mostly used to represent subgroups of the user's stakeholder group, i.e., represent different kinds of users.

EXAMPLE 5: PERSONA FOR ReWo-RATE

A persona relevant to ReWo-Rate could look like this: "Bob is a 30-year-old single and tech enthusiast. He is an early adopter of new tech gadgets and eager in reading and writing his opinions about his latest purchases."

3.2 Stakeholder Analysis

Typically, actual stakeholders are classified according to the classification of Figure 10. Each classification is highly project specific, i.e., you cannot generally peg a certain stakeholder group as "apathetics" in every project.

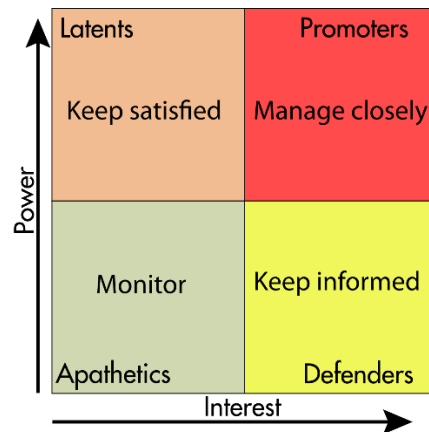


Figure 10: Stakeholder classification.¹³ (By Zirguez - Own work, CC0, <https://commons.wikimedia.org/w/index.php?curid=30927010>)

Table 3 depicts a (highly subjective) stakeholder analysis for ReWo-Rate.

Stakeholder group	Classification
project manager	Promoter. However, he or she is in charge manage other stakeholders
product owner	Promoter. Same as project manager
systems analyst	Defender. Often high identification with project but rather low power.
requirements engineer	Defender. Often high identification with project but rather low power.
requirements manager	Defender. Often high identification with project but rather low power.
software architect	Defender. Often high identification with project but rather low power.
systems architect	Defender. Often high identification with project but rather low power.
UX designer	Defender. Often high identification with project but rather low power.
frontend developer	Defender. Often high identification with project but rather low power.
backend developer	Defender. Often high identification with project but rather low power.
release manager	Apathetics. Often manages multiple projects, yet with low power.

¹³ German translations would be like apathetics = Gleichgültige, latents = Schläfer, defenders = Verteidiger, promoters = Förderer. Of course, conceptual models like these suggest dichotomic choice on each axis, which actually is a continuum, e.g., there might apathetics that are little less apathetic than other apathetics.

Stakeholder group	Classification
CICD developer	Apathetics. Often manages multiple projects, yet with low power.
functional tests	Apathetics. Often manages multiple projects, yet with low power.
test automation	Apathetics. Often manages multiple projects, yet with low power.
load tests	Apathetics. Often manages multiple projects, yet with low power.
penetration tests	Apathetics. Often manages multiple projects, yet with low power.
test manager	Apathetics. Often manages multiple projects, yet with low power.
customers	Latents. Their combined power is high, but interest has to be nurtured.
actual users	Latents. Their combined power is high, but interest has to be nurtured.
other product owners	Latents. Often somewhat reluctant to adapt their systems (interest) but with the power to block a rollout if their system is vital to the system under development.
sales	Promoters. Usually with high interest regarding the marketability of some software system (interest) but also putting high pressure on the product owner to suit customer's needs (power).
support	Apathetics. Often manages multiple projects, yet with low power.
release-to-production manager	Apathetics. Often manages multiple projects, yet with low power.
transition manager	Apathetics. Often manages multiple projects, yet with low power.
admin	Apathetics. Often manages multiple projects, yet with low power.
documentation quality assurance	Apathetics. Often manages multiple projects, yet with low power.
code quality assurance	Apathetics. Often manages multiple projects, yet with low power.
information security	Latents. Often manage multiple projects, yet with the power to block a rollout or even shutdown a system in case of a security incident.
technical architecture	Apathetics. Often detached from actual development projects. Provide mere guidelines.
enterprise architecture	Apathetics/Latents. Often detached from actual development projects. Might obstruct or stop a project if enterprise architecture goals are not met.
purchase	Latents. Often detached from actual development projects. Might defer project start due to difficulties in negotiations with subcontractors.
boss(es)	Latents/Promoters. Bosses of stakeholders have to power to overrule the stakeholders. Their interests range from outright opposition to high interest. It is vital for every project to have a promoter with high interest and power in the organization.

Table 3: Stakeholder analysis for ReWo-Rate

Note that the classification of stakeholders is dependent on the culture and politics of the organization and the particular development project as well. Generally, the interest in a particular

development project or running system is low if stakeholders are specialists that deal with a lot of projects or systems. Moreover, the power of stakeholders again depends on the company's culture and their standing within the company as well as their position in the organization's hierarchy.

Assessments in a stakeholder analysis often collide with the self-esteem of certain stakeholders. Thus, stakeholder analyzes are better kept confidential and accessible to just a few members of a development project team.

4 Quality Attributes

In the preceding chapter we have learnt that different stakeholders have different interest and concerns. Together with system analysts it is the job of an architect to transform the rather vague interest and concerns into proper requirements¹⁴ and identify constraints.

4.1 Requirements

Junior architects tend to narrow architecturally relevant requirements to something like "Keep the system modifiable by enforcing a 3-layered architecture". Though modifiability is key to a long-lasting system there are many other types of requirements that are important for architects. This chapter is meant to broaden the view for these types of requirements.

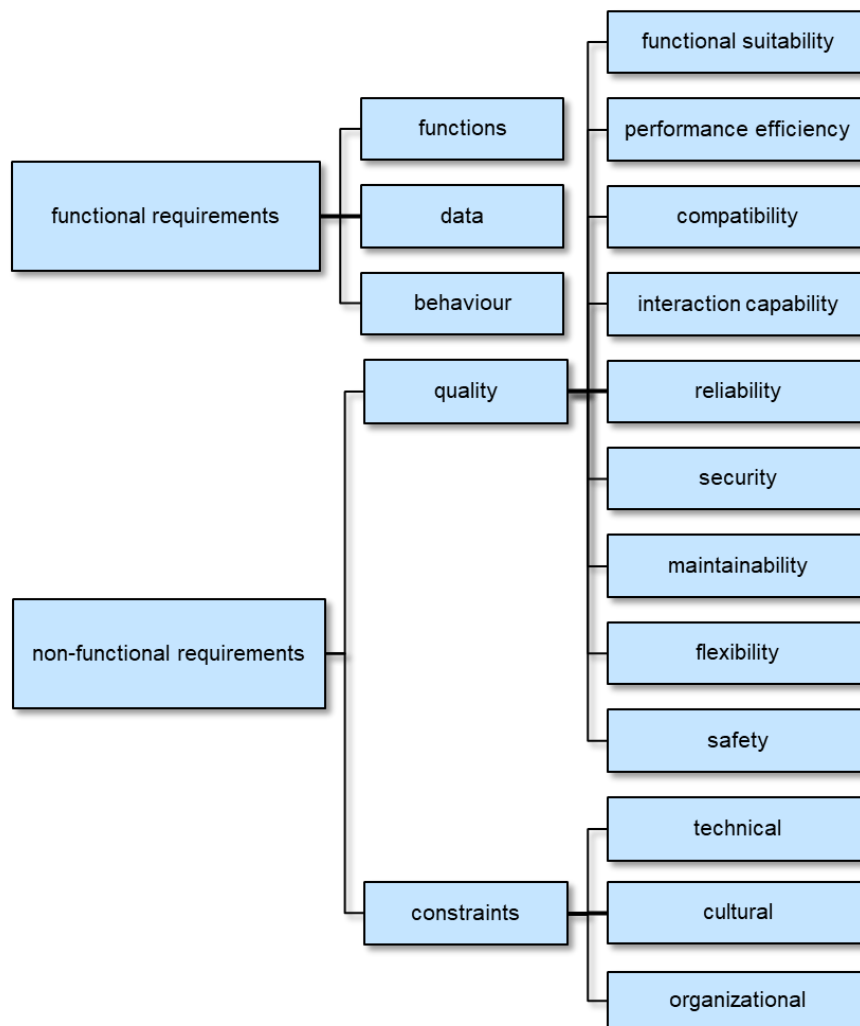


Figure 11: A partial taxonomy for requirements

¹⁴ According to [12] a proper requirement is complete, unambiguous, consistent, modifiable, verifiable, and traceable.

Requirements come along in very different flavors. Figure 11 depicts a taxonomy¹⁵ for requirements.

4.1.1 Functional Requirements

Functional requirements (FR) are those requirements that can be tested by functional tests. These tests describe the (stepwise) invocation of a certain **function** in a system, the client or user interaction with the system and the respective expected responses which are refined or influenced by the **behavior**, i.e., “rules”, of the system. Requirements about **data** describe the shape of the consumed, produced, processed and stored data.

Category	Example requirement
function	Sellers should trigger an email, which is sent to buyers and asks for a rating after a purchase has been completed.
data	Quantitative ratings, e.g., 1 to 5 stars, and a textual description are supposed to be persisted along with their associated transaction.
behavior	A rating can only be created by a seller. A buyer can edit, save, delete or commit it. After it has been deleted or committed, it cannot be further edited.

Table 4: Exemplary functional requirements for ReWo-Rate

Functional requirements have an influence on a system's architecture just inasmuch that their respective implementation need to be embodied in the source code in a reasonable, i.e., maintainable manner. The mapping of functional requirements to programs and program units is a virtue on its own and gave rise for methods like **domain-driven design** [14] for larger systems. Apart this mapping, functional requirements are not supposed to have a greater impact on a system's architecture.

4.1.2 Non-functional Requirements

Each requirement, which is no functional requirement, is called a **non-functional requirement (NFR)**. Such requirements require other test strategies like load tests for performance, code inspections and linting for maintainability, penetration tests for security etc. Non-functional requirements often contain so-called **architecturally significant requirements (ASR)**, which are those requirements that have an impact on the system's architecture. An architect rather focusses on these requirements whereas system analysts deal with sometime tedious details of how certain functions should look like or how a system should behave in certain situations.

EXAMPLE 6: NOT ARCHITECTURALLY SIGNIFICANT

"A displayed rating should be 200px * 100px in size" is not an architecturally significant requirement.

4.2 Quality Attributes / Ilities

There are non-functional requirements which can be negotiated and thus leave room for multiple design decisions. Such non-functional requirements correspond to **quality attributes** which are categorized into categories called **ilities** (since many of them end with -ility). There are many

¹⁵ This taxonomy is partial as it just includes those quality attributes that we are dealing with in the following. For example, dealing with "usability" would be a worth a book on its own and is therefore excluded. There are many taxonomies setting de factor or de jure standards yet each with just subtle differences, e.g., from [13].

possible ways to categorize quality attributes in a taxonomy. The most common taxonomy is that of the ISO/IEC 25010:2023 standard about software product quality [15] as shown in Figure 12 that refines the quality subtree of Figure 11.

SOFTWARE PRODUCT QUALITY								
FUNCTIONAL SUITABILITY	PERFORMANCE EFFICIENCY	COMPATIBILITY	INTERACTION CAPABILITY	RELIABILITY	SECURITY	MAINTAINABILITY	FLEXIBILITY	SAFETY
FUNCTIONAL COMPLETENESS	TIME BEHAVIOUR	CO-EXISTENCE	APPROPRIATENESS RECOGNIZABILITY	FAULTLESSNESS	CONFIDENTIALITY	MODULARITY	ADAPTABILITY	OPERATIONAL CONSTRAINT
FUNCTIONAL CORRECTNESS	RESOURCE UTILIZATION	INTEROPERABILITY	LEARNABILITY	AVAILABILITY	INTEGRITY	REUSABILITY	SCALABILITY	RISK IDENTIFICATION
FUNCTIONAL APPROPRIATENESS	CAPACITY		OPERABILITY	FAULT TOLERANCE	NON-REPUDIATION	ANALYSABILITY	INSTALLABILITY	FAIL SAFE
			USER ERROR PROTECTION	RECOVERABILITY	ACCOUNTABILITY	MODIFIABILITY	REPLACEABILITY	HAZARD WARNING
			USER ENGAGEMENT		AUTHENTICITY	TESTABILITY		SAFE INTEGRATION
			INCLUSIVITY		RESISTANCE			
			USER ASSISTANCE					
			SELF-DESCRIPTIVENESS					

Figure 12: ISO 25010 Software Product Quality Categories [15]

The following Table 5 provides the definition of each subcategory and an example for ReWo-Rate.

Category & Subcategory	ISO 25010 Definition	ReWo-Rate Example
1. Functional Suitability		
<i>Functional Completeness</i>	Degree to which the set of functions covers all the specified tasks and user objectives.	The system includes the full feedback loop: a widget for customers to submit ratings, an algorithm to aggregate them, and a dashboard for shop owners to moderate them.
<i>Functional Correctness</i>	Degree to which the system provides the correct results with the needed degree of precision.	The aggregation algorithm calculates the precise mean score. If there are three 5-star ratings and one 4-star rating, the display renders exactly 4.75 stars.
<i>Functional Appropriateness</i>	Degree to which the functions facilitate the accomplishment of specified tasks and objectives.	The "Verified Buyer" badge logic only activates if the review token matches a completed transaction ID, directly supporting the objective of establishing trust.
2. Performance Efficiency		
<i>Time Behavior</i>	Degree to which the response and processing times and throughput rates meet requirements.	The JavaScript widget loads asynchronously in under 200ms in the 99.99 th quantile, ensuring it does not block the "First Contentful Paint" of the host web shop.
<i>Resource Utilization</i>	Degree to which the amounts and types of resources used meet requirements.	The backend uses Redis caching to store aggregated scores, reducing database CPU load by 80% compared to calculating the average on every page load.
<i>Capacity</i>	Degree to which the maximum limits of a system parameter meet requirements.	The architecture supports a concurrent user load of 5,000 requests/second, sufficient to handle traffic spikes during Black Friday sales.
3. Compatibility		
<i>Co-existence</i>	Degree to which the product can perform its required functions efficiently while sharing a common	The ReWo-Rate widget is namespaced (e.g., ReWo.init) to ensure it does not conflict with other libraries like jQuery or Google Analytics running on the client's site.

Category & Subcategory	ISO 25010 Definition	ReWo-Rate Example
	environment and resources with other products.	
<i>Interoperability</i>	Degree to which two or more systems can exchange information and use the information that has been exchanged.	ReWo-Rate includes API connectors that automatically pull order details (JSON) from external platforms like Shopify and WooCommerce to trigger review emails.
4. Usability		
<i>Appropriateness Recognizability</i>	Degree to which users can recognize whether a product or system is appropriate for their needs.	The widget uses universally recognized symbols (★) and color coding (green for high, red for low), making its purpose immediately obvious to shoppers.
<i>Learnability</i>	Degree to which the product or system can be used by specified users to achieve specified goals of learning to use the product or system.	The shop owner dashboard features a "Quick Start" wizard that guides a new user through color customization and code generation in 3 steps.
<i>Operability</i>	Degree to which the product or system has attributes that make it easy to operate and control.	The dashboard allows bulk actions, enabling shop owners to "Archive" or "Approve" multiple reviews at once using checkboxes rather than processing them individually.
<i>User Error Protection</i>	Degree to which the system protects users against making errors.	The "Submit" button remains disabled until a star rating is selected, preventing users from submitting empty or invalid data sets.
<i>User Interface Aesthetics</i>	Degree to which a user interface enables pleasing and satisfying interaction for the user.	The widget supports "Dark Mode" and custom CSS injection to seamlessly blend with the visual branding of high-end fashion web shops.
<i>Accessibility</i>	Degree to which the product or system can be used by people with the widest range of characteristics and capabilities.	The widget complies with the Web Content Accessibility Guidelines [16]; star ratings are navigable via keyboard (Tab + Enter) and have ARIA labels for screen readers.
5. Reliability		
<i>Maturity</i>	Degree to which the system meets needs for reliability under normal operation.	The codebase has undergone 6 months of production usage with 0 critical defects reported, verified by a passing rate of 99% in automated regression tests covering 95% of all instructions.
<i>Availability</i>	Degree to which a system, product or component is operational and accessible when required for use.	ReWo-Rate is hosted on a redundant cloud infrastructure guaranteeing 99.99% uptime, ensuring the ratings never disappear from client shops.
<i>Fault Tolerance</i>	Degree to which the system operates as intended despite the presence of hardware or software faults.	If the main database fails, the widget switches to a "Read-Only" mode using cached data from a CDN, ensuring users still see ratings even if they cannot submit new ones.
<i>Recoverability</i>	Degree to which, in the event of an interruption or a failure, the product or system can recover the	Automated "Point-in-Time" backups are taken every 5 minutes, allowing the system to be restored to a state just prior to any data corruption incident.

Category & Subcategory	ISO 25010 Definition	ReWo-Rate Example
	data directly affected and re-establish the desired state of the system.	
6. Security		
<i>Confidentiality</i>	Degree to which the product or system ensures that data are accessible only to those granted to have access.	Personally Identifiable Information (PII) of reviewers (emails, names) is encrypted at rest and masked in the dashboard (e.g., j***@email.com) for support staff.
<i>Integrity</i>	Degree to which a system, product or component prevents unauthorized access to, or modification of, computer programs or data.	Cryptographic hashing prevents direct database manipulation. Even a database admin cannot alter a 1-star review to a 5-star review without breaking the hash signature.
<i>Non-repudiation</i>	Degree to which actions or events can be proven to have taken place.	Every moderation action (e.g., hiding a review) is logged in an immutable audit trail with the User ID, Timestamp, and IP address.
<i>Authenticity</i>	Degree to which the identity of a subject or resource can be proved to be the one claimed.	API requests from the widget are signed with JSON Web Tokens (JWT) to verify they originate from a valid, registered web shop domain.
7. Maintainability		
<i>Modularity</i>	Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.	The "Billing Service" is a microservice separate from the "Rating Widget." Developers can update payment logic without risking downtime for the widget display.
<i>Reusability</i>	Degree to which an asset can be used in more than one system, or in building other assets.	The "Star Rating Component" (UI) is a shared library used identically in the email templates, the web widget, and the admin dashboard.
<i>Analyzability</i>	Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change, to diagnose deficiencies or to identify parts to be modified.	The application uses structured logging (JSON). Errors include a Trace-ID that allows developers to instantly locate the specific line of code that failed.
<i>Modifiability</i>	Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.	Configuration for widget themes is stored in separate JSON files, allowing developers to add new visual themes without altering the core rendering logic.
<i>Testability</i>	Degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed.	The architecture uses Dependency Injection, making it easy to mock the database and external APIs for fast, automated unit testing.

Category & Subcategory	ISO 25010 Definition	ReWo-Rate Example
8. Portability		
<i>Adaptability</i>	Degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software or other operational or usage environments.	The widget is responsive: it renders as a horizontal row on desktop screens and automatically stacks vertically on mobile viewports smaller than 480px.
<i>Installability</i>	Degree of effectiveness and efficiency with which a specified product or system can be successfully installed and/or uninstalled in a specified environment.	Installation requires simply pasting a standardized <code><script></code> tag into the header. No server-side installation or binary compilation is needed.
<i>Replaceability</i>	Degree to which a product can replace another specified software product for the same purpose in the same environment.	ReWo-Rate offers a "Data Export" tool (CSV/JSON), allowing shop owners to easily migrate their data to a competitor if they choose to leave, preventing vendor lock-in.

Table 5: ISO 25010 Software Product Quality Categories Definitions and Examples

Table 6 exemplifies each quality requirement category.

Category	Example requirement
response time	When a user opens a pending rating, the response time should be below 1000 ms in the 99th percentile.
throughput	ReWo-Rate must be able to process 300 transaction per second. The term "transaction" is common in these kinds of requirement but calls for a specific definition. Since ReWo-Rate is a web application, it consumes HTTP requests (cf. Section 27.1), processes them, executes one or more transactions in its database, maybe calls external web services and finally produces a response. With a single "transaction", we mean this whole process: Producing a HTTP response for a HTTP requests with all intermediate processing steps.
Capacity	ReWo-Rate must store a maximum of about 100.000.000.000 ratings.
Scalability	ReWo-Rate resources are supposed to be utilized at an average of 60% while the workload peaks vary between 50 ratings/s (3 a.m. on a summer vacation day) a 500 ratings/s (6 p.m. Christmas holidays).
Availability	ReWo-Rate must be available for 99.99%, i.e., it may be down for at most 52 min. / year.
Faultlessness	ReWo-Rate must be 100% available on Black Friday.
Resilience	The core use cases, i.e., store and retrieve ratings must still work even if all partner systems are unavailable.
Security	URLs to ratings should be practically impossible to guess.
Maintainability	The technical debt should be below 100 days for repair according to a given SonarQube ruleset.

Table 6: Exemplary quality requirements for ReWo-Rate

4.3 Constraints

On the other side, **constraints** are not negotiable. They imply design decisions made externally. Constraints can be technical in nature, organizational, or imposed by legislation.

Constraint	Category	Explanation and rationale
Google Cloud	technical	Due to a partnership between ReWo-IT and Google, it's a matter of fact that ReWo-Rate will be operated in GCP. Therefore, GCP's strengths and limits must be considered right from the start.
CICD	organizational	Development processes at ReWo-IT follow a continuous delivery approach. So, the build of ReWo-Rate has to be automated and development processes to be set-up appropriately.
Enterprise Service Bus	technical	The ReWo-IT's reference architecture requires that ReWo-Rates uses an existing enterprise service bus for message-based communication with other systems of ReWo-IT.
GDPR	legislative	Users of ReWo-Rate have to consent to ReWo-IT's common cookie policy. Therefore, a respective pop up should be displayed to unknown users.
SSL only	technical	ReWo-Rate is exposed to the internet via https only. This has to be considered by external systems that are integrated with ReWo-Rate.
storage encryption	technical	Every data record that is stored by ReWo-Rate must be encrypted. So, if an attacker gets hold of raw data files, these must be useless for him or her.
rating scale	Cultural	The rating scale must adapt to the customs of a buyer's country. A rating scale of one to five stars is the default.

Table 7: Exemplary constraints for ReWo-Rate

In the following chapter, we focus on specific categories and subcategories of quality.

5 Performance

Performance is most noticeable by users. Keeping a large-scale system performant from the end-user's point of view is particularly challenging as many users compete for system resources and generate large cumulative workloads.

The ISO/IEC 25010:2023 defines not just performance but **performance efficiency** as the “capability of a product to perform its functions within specified time and throughput parameters and be efficient in the use of resources under specified conditions” [15, p. 2], thereby taking costs into account. However, we set the cost factor aside in the next sections and just define aspects of performance as such.

5.1 Workloads and Hardware Limits

Workload is a rather vaguely and non-technically defined term in literature. It refers to the computational complexity to process a job times the frequency, i.e., number of jobs per unit time.

EXAMPLE 7: TYPICAL LARGE SCALE SYSTEM WORKLOADS

The authors of Figure 13 have gathered some statistics and quantifies what we mean by actual jobs and their numbers for some of the largest applications in 2023.

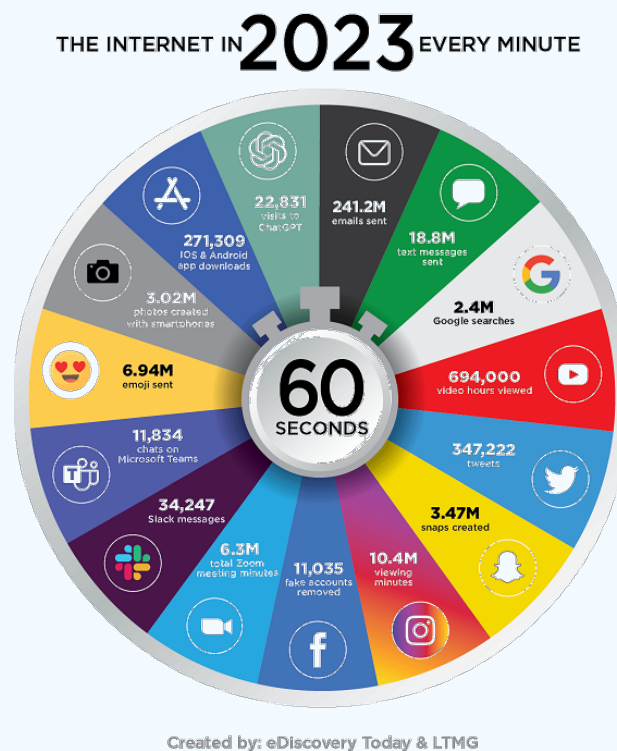


Figure 13: Workload for large scale systems in 2023 [17]

Each of these workloads has its specific demands for hardware resources like processing units, memory, storage devices and network interfaces. These hardware resources are of course finite in latency, throughput, and capacity and might therefore pose a bottleneck for certain **workload types**. In the following, we discuss that in detail.

5.1.1 Processing Units (CPU, GPU)

Processing units like the **central processing unit (CPU)** or the **graphics processing unit (GPU)** run machine code. They are basically giant assemblies of **transistors**, which are grouped into **logic gates** like AND, OR, NOT. These gates are grouped into **functional blocks** like adders, decoders, registers, or counters. Functional blocks are grouped into more complex **execution units** like the arithmetic logic unit (ALU) of a CPU. Execution units constitute **microarchitectures**. A microarchitecture implements an **instruction set architecture (ISA)** like x86-64, ARM, or RISC-V that defines the supported data types, memory model, operation codes (opcodes) etc. thus the machine code that programs must be compiled to.

A CPU or GPU can just execute a limited number of machine instructions in a given time. This number is limited by the frequency of the CPU **clock signal**, which typically ranges between 1.7 GHz and 4.0 GHz, the CPU **cache** sizes and speeds and the number of CPU **cores**, which can execute machine instructions in parallel.

Workloads that have processing units as their bottleneck, e.g., would be processed faster if the processing units were faster, are said to be **CPU-bound** or **GPU-bound**, respectively.

5.1.2 Memory

Memory¹⁶ differs in capacity, latency and throughput, i.e., the amount of data that can be transferred read and written in a given time span.

Workloads with memory as bottleneck are called **memory bound**. Since memory accesses are still typical machine instructions, CPU-bound and memory bound workloads are not easy to tell apart: CPU cores might be stalled most of the time, i.e., actively wait for data to be transferred from/to memory during the execution of such memory access instructions. If these instructions are frequent in comparison with other instructions like control flow or arithmetic instructions, then we can talk of a memory bound workload.

Fortunately, processor caches can significantly reduce such stall times as their latency is much lower (cf. Section 6.3). We recur to processor caches and their effects in Section 22.3.1a)

Please note that common monitoring tools like top(1) in Linux or the Task-Manager in Windows do not display stall times explicitly. So, if a CPU core is utilized by some percentage by a certain program, this might be due to stalls instead of “real” computation. So, if the overall CPU utilization is measured as too high given a certain workload, latency between CPU and memory might be the reason but not the CPU itself [18], which can be remedied programmatically to a certain extend by accessing memory such that cache misses are avoided.

Also note that GPUs have their own kind of memory called Video Random Access Memory (VRAM) that is often separate from the Random Access Memory of CPUs. Moreover, VRAM is dedicated to exactly one GPU and data between different VRAMs can only be shared via high-speed links. Since machine Learning model weights can only be reasonably stored in VRAM and since the model sizes often are in the hundreds of gigabytes, VRAM capacity is often a limiting factor. Sharding a model into different VRAM along layers or along matrix columns, called pipeline parallelism or tensor parallelism, respectively, is a way to cope with this limitation.

¹⁶ With memory, we mean what is also called random access memory (RAM) or main memory. All these designations have their weaknesses, so we have picked the shortest.

5.1.3 Storage

Durability is the difference between what we call storage and memory: Data in storage survives power outages, data in memory is erased¹⁷. Like memory, storage has a finite read-write throughput and non-zero latency. Often, storage performance is expressed by means of **input-output operations per second (IOPS)**. Single HDDs range from 20 IOPS to 200 IOPS and SSDs to 10,000,000 depending on the size of the blocks read or written in each operation [19], the interface type (fiber channel, serial attached SCSI (SAS), serial ATA (SATA)), the rotational speed in case of HDDs (from 5400 rpm to 15000 rpms), the distribution of blocks and thus the need for rotation and repositioning read/write heads in case of HDDs.

If processes spend significant time in a waiting state for an input-output operation to complete, the workload that is processed by these processes is called **(storage) IO bound**.

5.1.4 Network

Networks have a limited throughput and non-zero latency, too. Maximum throughput is typically limited by the link speed of network interfaces, which nowadays ranges from kilobytes (poor mobile connection) to gigabytes per second (fast LAN). Latency is inherent to networks as described in Section 5.2.2.

Again, if processes frequently wait for network input-output operations to be completed, we speak about **(network) IO bound** workloads.

5.1.5 Workload Bottlenecks

If a workload type is bound to a certain resource, that resource limits the processing performance of that workload type. 3D games, for example, mostly depend on GPUs.

The following Table 8 shows how much a certain workload type demands of what resource. For memory, storage, and network, we focus on throughput (TP) because that can be out-scaled to some extent while latency mostly depends on the underlying technology (HDD vs. SSD) or the proximity of communicating resources (network nodes in the same data center vs. on different continents).

Category	Workload Type	CPU	GPU	Memory TP	Storage TP	Network TP	Example Technology
Web Applications	Web Applications	●	●	●	●	●	Java, Java Spring, Postgres
Gaming	3D Gaming	●	●	●	●	●	Any 3D game
Analytics	Big Data / Analytics	●	●	●	●	●	Spark, Presto
Scientific Computing	Genomic Sequencing	●	●	●	●	●	BWA, GATK
	Scientific Computing	●	●	●	●	●	MATLAB, OpenFOAM
	Weather Forecasting	●	●	●	●	●	Global Forecast System (GFS)
	Financial Modeling	●	●	●	●	●	Excel, MATLAB, QuantLib

¹⁷ This boundary is blurred by the advent of technologies like NVRAM in the last years.

Category	Workload Type	CPU	GPU	Memory TP	Storage TP	Network TP	Example Technology
Machine Learning	Machine Learning Training	●	●	●	●	●	PyTorch, TensorFlow
	Machine Learning Inference	●	●	●	●	●	ONNX Runtime, TensorRT
IoT & Edge	IoT Data Aggregator	●	●	●	●	●	Apache Kafka/Spark/Flink
Media & Graphics	3D/Video Rendering	●	●	●	●	●	Blender
	Image Rendering	●	●	●	●	●	Blender
	Media Delivery	●	●	●	●	●	Youtube
	Media Processing	●	●	●	●	●	AMPP
Blockchain	Blockchain Node	●	●	●	●	●	Ethereum, Bitcoin Core
	Cryptocurrency Mining	●	●	●	●	●	NiceHash, PhoenixMiner
Software Development	Software Development	●	●	●	●	●	JDR, Visual Studio Code
	CICD-Pipelines	●	●	●	●	●	GitLab CICD
Supporting Workloads	(De-)Compression	●	●	●	●	●	gzip, zstd
	Content Delivery / Web Server	●	●	●	●	●	Cloudflare, Akamai, nginx
	Storage Services	●	●	●	●	●	Google Cloud Buckets
	Database OLTP	●	●	●	●	●	Postgres, Oracle CB
	Streaming	●	●	●	●	●	Apache Kafka
	Stream Processing	●	●	●	●	●	Apache Kafka/Spark/Flink

Table 8: Workload Types vs. Bounds

The colored circles denote one of three values “high”, “moderate”, and “low”. Table 9 clarifies, what these values actually mean for a respective resource type. (Workloads that make no use of a certain resource at all – like databases of GPUs – have a “low” in the respective cell.)

Resource Type	High-Threshold	Moderate-Threshold	Low-Threshold	Unit
CPU	200	50		Billion Floating Point Operations per Second (GFLOPS)
GPU	50,000	5,000		

Resource Type	High-Threshold	Moderate-Low-Threshold	Unit
Memory Throughput	800,000	200,000	Megabytes per Second (MB/s)
Storage Throughput	1,000	200	
Network Throughput	10,000	1,000	

Table 9: Resource Types, Units and Thresholds

5.2 Network Latency and Response Time

Large and distributed systems naturally tend to be network IO bound. Therefore, it is crucial to understand what latency and response times actually means particularly for network communication.

5.2.1 Network Metrics

Even for a single client-server system, there are many time spans that contribute to latency and response times as the server, client (browser) and even the user work in parallel. Figure 14 shows common time measures with respect to a single HTTP request.

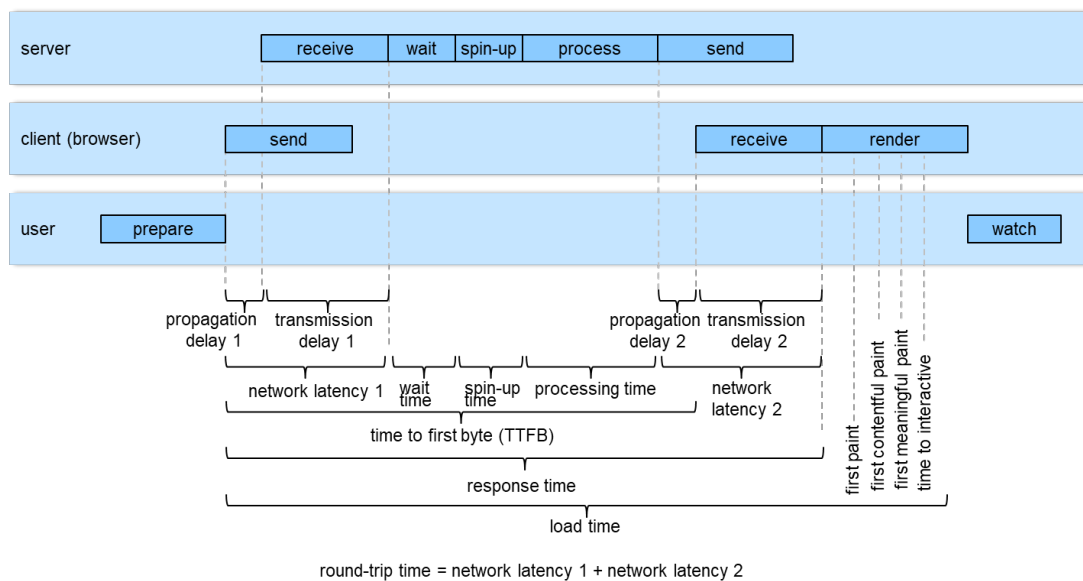


Figure 14: Common time measures

In network theory, **network latency** is the time a message needs to travel from a sender to a receiver. In a round-trip, the server replies to a request with a response and the **round-trip time (RTT)** the sum of both network latencies.

System architects are mostly concerned with **response times**¹⁸ which basically is round-trip time plus **wait time** on server plus **spinning-up time**¹⁹ for creating an appropriate process plus actual **processing time**, i.e., the time the process needs to process the request and produce a response.

¹⁸ Response time and latency are often used as synonyms in literature. We use the term response time to better distinguish it from network latency.

¹⁹ If corresponding processes have been created before and are just pooled, then this is neglectable. However, with the advent of serverless computing / Function-as-a-Service this time span again gains relevance.

Response times and particularly network latency can be kept small by locating copies of frequent responses in private local or shared remote caches that are (a) close to the expected locations of the clients and (b) do just a lookup of stored responses based on the request instead of actual computing a response (cf. web caching in Section 23.9). Processing time can be kept small by, e.g., parallelizing even the processing of a single request where possible.

Figure 14 simplifies things particularly inasmuch a typical web page consists of multiple responses. Moreover, **rendering time** in a browser is a complicated process that produces intermediate results that a user can view or interact with before every response has been received and completely rendered. If you are interested in browser networking: An excellent resource is <https://hpbn.co/> which is a free online version of [20].

5.2.2 Network Latency

A normal web application, where the client (browser) and the server are located anywhere on the globe, can suffer a lot from network latency. The ultimate threshold is the speed of light, which limits the **propagation delay** of a single bit of data.

EXAMPLE 8: ROUND-TRIP TIME BETWEEN COLOGNE AND NEW YORK

A round-trip between Cologne and New York, which are 6,000 km apart from each other, cannot be faster than $\frac{2 \cdot 6,000 \text{ km}}{300,000 \frac{\text{km}}{\text{s}}} = 40 \text{ ms}$.

Of course, in fact there are additional factors that slow down the **round-trip time (RTT)** [20, p. 15]:

- The propagation delay is typically lower: The speed of light in fiber is just about 200.000 km/s.
- The cables on the network route are in sum longer than the beeline.
- If the request or response are bigger, then **transmission delay** comes into play. This is the time it takes a sender to put a message on a link from the first to the last bit. From a receiver's viewpoint, it is the time between the arrivals of the first and the last bit. This delay depends on the data rate of the respective link, e.g., 100 Mbit/s.
- IP packets of a message might also wait in queues before being routed on a router (a.k.a. hop). Such **queuing delays** occur due to
 - **link congestion**, i.e., too many packets are routed from too many input links to too slow and few output links,
 - **serialization delay**, i.e., if a packet is unlucky because it hits a router just when another packet is being processed which can happen even also in very underutilized routers.
- **Packet drops**, which occur if an IP packet cannot be added to an already full queue and thus requires **retransmission**.
- TCP segments that have been created due to a write to a socket by some process might also remain in a buffer on the sender's side before being sent.
 - The **congestion window (CWND)** limits the number of unacknowledged TCP segments in flight. If the CWND is exceeded, TCP segments that are created due to socket writes of a process are withheld.
 - Even if CWND is not exceeded, senders might use **Nagle's algorithm** [21] to avoid network congestion: If there are unacknowledged TCP segments in flight, small writes, e.g., just some bytes in each write, are buffered on the sender's side. A new TCP segment is sent not before either the sender receives acknowledgements for the inflight data or the **maximum transfer**

unit (MTU) for a TCP segment is reached. Nagle's algorithm is a default setting, i.e., TCP_NODELAY is not set by default, neither in Windows [22] nor in Linux [23]. Therefore, even nodes behind network interfaces that are directly connected to each other can face network latencies around 40 milliseconds.

5.2.3 Impact of Response Time

Particularly in web applications, response time to user agents (browsers) is of major importance. With higher response times the **conversion rate**, i.e., the probability that a visitor becomes a paying customer, drops significantly.

Response time	User perception
0 – 100 ms	Instant
100 – 300 ms	Small perceptible delay
300 – 1,000 ms	Machine is working
> 1,000 ms	Likely mental context switch
> 10,000 ms	Task is abandoned

Table 10: User perceptions of response times (after [15, p. 170])

In internal communication within a distributed system, another negative effect is that a single node could be deemed to be faulty by, e.g., the health check of a load balancer, if the node does not respond within certain timeout periods.

5.2.4 Response Time Variance

Real-time systems guarantee certain fixed response times. However, in usual web applications we do not have such guarantees. Instead, response time varies for diverse reasons [24, p. 19]:

- **context switches** between processes consume processing time,
- **page faults** require reads from disk, which is why machines are usually equipped with enough memory to prevent page faults or even swapping at all,
- a **garbage collector** run could stop all Java threads in a JVM for up to 60 seconds,
- **retransmission** of IP packets, which have been dropped on their route, or
- **mechanical faults** like vibrations that slow down disk IO drastically.

5.2.5 Measuring Response Time

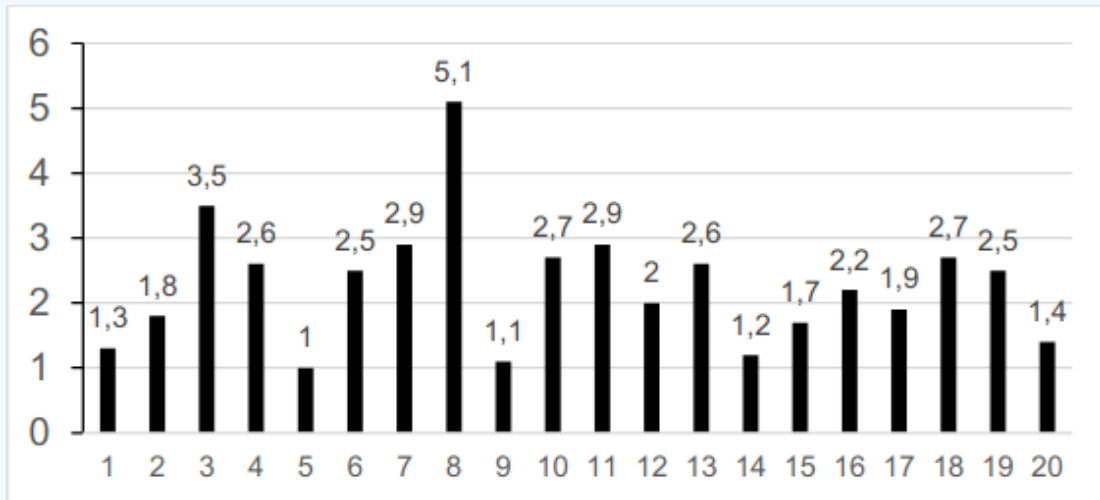
Due to the variation of response times, it would be pointless to require an upper bound for response time. Instead, response time requirements are often made with respect to the 0.99-**percentile** (P99) of response times. If the requirement says "The P99 of response times must be at most 1,000 ms", then 99% of all response times must be below 1,000 ms and just 1% may be above 1,000 ms.

Given a sequence $(x_1, x_2, \dots, x_n)$ of (measured) response times, which is sorted by size, i.e., $x_1 \leq x_2 \leq \dots \leq x_n$ the p -percentile x_p is defined by

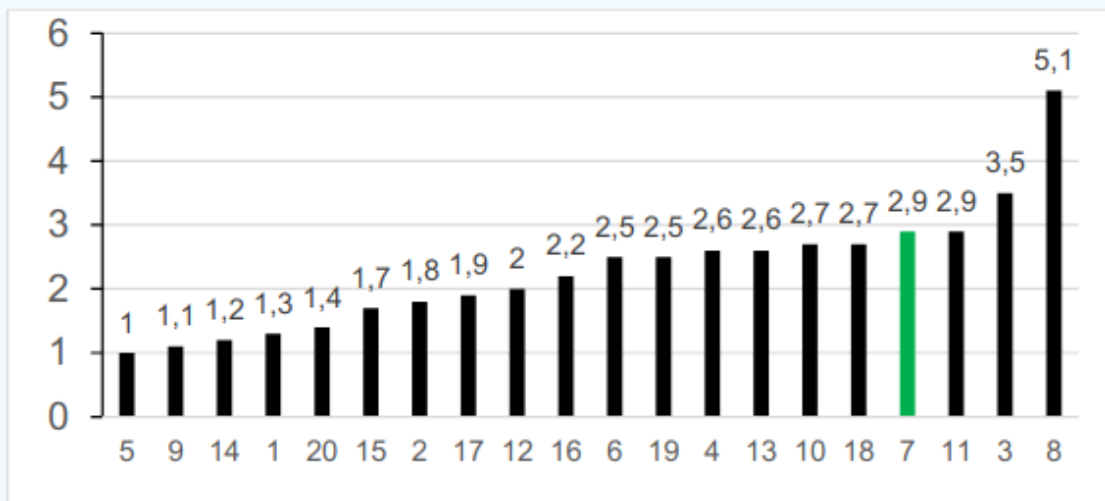
$$x_p = \begin{cases} \frac{1}{2}(x_{n \cdot p} + x_{n \cdot p + 1}), & \text{if } (n \cdot p) \in \mathbb{Z} \\ x_{\lfloor n \cdot p + 1 \rfloor} & \text{otherwise} \end{cases}$$

EXAMPLE 9: RESPONSE TIMES

Consider the sample response times from below.



We want to compute the 0.83-percentile (P83). First, we have to order the response times by size:



For n , we have 20 values in total. With $n \cdot p = 20 \cdot 0.83 = 16.6 \notin \mathbb{Z}$ we have

$$x_{0.83} = x_{\lfloor 20 \cdot 0.83 + 1 \rfloor} = x_{\lfloor 17.6 \rfloor} = x_{17} = 2.9$$

If we want to compute the 0.95-percentile (P99), we have $n \cdot p = 20 \cdot 0.95 = 19 \in \mathbb{Z}$ and therefore

$$x_{0.95} = \frac{1}{2}(x_{20 \cdot 0.95} + x_{20 \cdot 0.95 + 1}) = \frac{1}{2}(x_{19} + x_{20}) = \frac{1}{2}(3.5 + 5.1) = 4.3$$

5.3 Throughput

Response time might not be mixed up with **throughput**. Response time is measured from the viewpoint of a client. Throughput is a property of a server, which may have thousands of clients,

which are unaware of each other. With constant throughput, response times generally increase with the number of clients.

EXAMPLE 10: THROUGHPUT VS. RESPONSE TIME

Consider a system that processes 100,000,000 requests simultaneously with a processing time of 10 seconds for each request. This yields a high throughput of 100,000,00 requests per second. However, the response time of at least 10 seconds would be considered slow from a single client's perspective.

5.3.1 System Throughput

Throughput might not just refer to incoming requests per second. Instead, throughput²⁰ generally refers to "jobs"²¹ per second that have to be processed.

In queuing theory, a **M/M/1 queue** is most simple model that explains throughput:

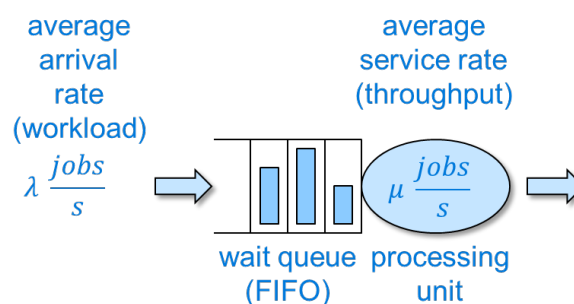


Figure 15: Simple M/M/1 queuing model

Jobs enter a system with a certain **arrival rate** and wait for some time in a wait queue. If a job becomes the oldest job in that queue, it is dequeued and is being processed by the processing unit. The time it takes the processing unit to process the job is the **service time**. Its reciprocal value is the **service rate** (a.k.a. departure rate). The **job size** is the required work of the **processing unit**, e.g., a CPU core, to process the job, e.g., generate a response to a request. The job size can be expressed in, e.g., CPU cycles or seconds of overall processing time spent for a single job.

Please note that “jobs”, “processing unit” and “wait queue” are just abstract concepts. In real systems they are hardware resource like those in the following table.

Job	Processing Unit	Wait Queue
OS Scheduler Time Slices	CPU core	Run queues
IO write (syscalls, interrupts)	Disk controller	Disk IO buffer

²⁰ In an undersaturated system, one has to distinguish between **actual throughput** and the **maximum throughput** -- also known as the **bandwidth** -- that would saturate the system. However, we use “throughput” homonymous for both actual and maximum throughput.

²¹ The terms requests, messages, transaction, and job are often used interchangeable when a system’s throughput is discussed. This is how they relate: An incoming message is called a request (message), if we talk about request-response oriented communication of a system with others (cf. Section 21.1.1). Such a message contains data that gives rise to new job which processes the data. Technically, such a job is nothing but a new process, thread, or cascade of function calls that begins in some kind of event loop, so in essence, a job is some kind of program execution. If such a job modifies data (in a database), jobs or also referred to as transactions as they include what we know by the concept of a database transaction (cf. Section 33.3.2c)).

Job	Processing Unit	Wait Queue
HTTP request	Server process listening on some port	Socket read buffer, Custom buffers
Message Broker Message	Subscriber	Message Broker Message Queue / Topic

Table 11: Actual jobs, processing units, and wait queues

In an M/M/1 queue, jobs arrive randomly and job sizes S vary. Therefore, we are just interested in the averages of both rates. Here, **workload** is another word for the **average arrival rate** λ , which is a quantity with a reciprocal time dimension, $\dim \lambda = T^{-1}$ (reciprocal time). **Throughput** is synonymous to the **average service rate** μ if no jobs are dropped from the wait queue and **average service time** t_s is

$$t_s := \frac{1}{\mu}$$

where, $\dim t_s = T$ (time). Although throughput and response time are different concepts, they correlate: Lowering network latency is still out of reach but using a faster processing unit increases throughput, thus decreasing the processing time for each job but also the time a job waits in the queue. Parallelizing processing units with dedicated work queues multiplies the throughput but also decreases the average number of jobs in a queue and therefore the waiting time in a queue for each job.

According to **Little's Law** [25], there is a simple and intuitive connection between

- The average **overall time** $W := W_Q + t_s$ a job (request) spends in a system with a **queue waiting time** W_Q ,
- the **average arrival rate** λ , and
- the **average number of jobs in the system** $L := L_Q + L_p$, i.e., those in queues L_Q plus those that are currently processed L_p :

$$L = \lambda W$$

The same holds true if we are just interested in waiting time W_Q and average number of jobs in queues L_Q :

$$L_Q = \lambda W_Q$$

5.3.2 Network Throughput

The term throughput is also applicable to single hardware resources like network devices or storage controllers. In network devices, processing basically boils down to manipulating and forwarding packets. Here, throughput is synonym to **transmission rate** (e.g., 1000 MB/s) and reciprocal to transmission delay.

Please note that in the case of TCP, network latency—a.k.a. roundtrip time (RTT)—limits network throughput depending on the size of the **TCP receive window (RWin)**, which is the amount of data a sender might send without getting an acknowledgement from the receiver.

$$\text{max. network throughput} \leq \frac{RWin}{RTT}$$

EXAMPLE 11: MAXIMAL NETWORK THROUGHPUT

With an RWin of 64 kB and an RTT of 150 ms the maximal network throughput is $427 \frac{kb}{s}$

5.4 Utilization

The **utilization**²² ρ of a system is defined by

$$\rho := \frac{\lambda}{\mu} = \lambda t_s.$$

- If $\rho = 1$, then a system is **saturated**.
- If $\rho > 1$, then a system is **oversaturated**, which inevitably leads to infinite growth of jobs in wait queues in theory and a congestion of wait queues and therefore to dropped jobs²³ in practice.
- If $\rho < 1$, the system is **undersaturated**, i.e., there are times when there is no job in the system and the processing unit (CPU core) switches to an idle state²⁴.

Of course, this model is a little simplistic as it omits the actual statistical distribution of job arrivals and job sizes as well as the possibility to have several parallel processing units each with a dedicated wait queue. [26] delves into these kinds of scenarios and their mathematical modeling.

Getting an undersaturated system close to $\rho = 1$ is not desirable with respect to necessary queue capacities and response times: If the arrival process is **stationary**, e.g., the average arrival rate and its variance remain constant²⁵, and the system is undersaturated²⁶, the average number of jobs L

in the system is $L = \frac{\rho}{1 - \rho}$

Figure 16 is a plot of this function.

²² Sometimes the term "**load**" is used synonymously with "utilization". In order to avoid confusion with the external "workload" of a system, we do not use this term. The term "saturation" is also used synonymously to "utilization".

²³ Before that, response times typically exceed the timeout set by clients, e.g., 300 seconds in case of a browser, so that even if the job is finally processed, the response is of no use.

²⁴ In case of an underutilization, modern CPUs can trade off higher utilization against lower power consumption.

²⁵ Figure 21 depicts an arrival process which is not stationary due to the long-term growth which particularly increases the average arrival rate over time.

²⁶ For saturated and oversaturated systems, the following formulae make no sense since the number of jobs and wait times become infinite and negative infinite, respectively.

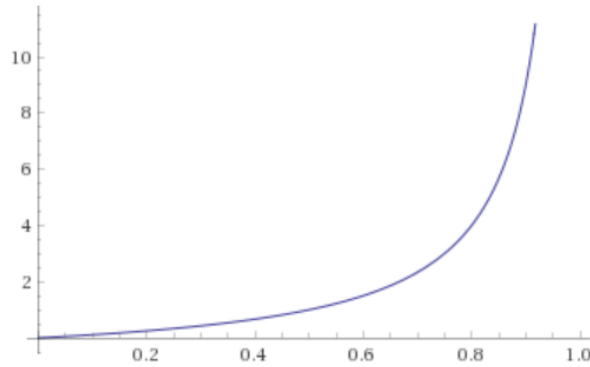


Figure 16: Utilization (x-axis) and avg. jobs in the system (y-axis)

EXAMPLE 12: UTILIZATION VS. AVERAGE JOBS IN SYSTEM

With a utilization $\rho = 0.4$ we have $\frac{2}{3}$ jobs in the system on average, with $\rho = 0.6$ it is 1.5, with $\rho = 0.8$ it is 4, $\rho = 0.9$ it is 9, with $\rho = 0.99$ it is 99, with $\rho = 0.999$ it is 999 and so forth.

With utilization $\rho := \frac{\lambda}{\mu} = \lambda t_s$ and $L = \frac{\rho}{1-\rho}$ and Little's Law $L = \lambda W$ we get:

$$W = \frac{1}{\mu(1-\rho)} = \frac{1}{\mu - \lambda} \Leftrightarrow W = \frac{t_s}{1-\rho}$$

which obviously is only meaningful if $\lambda < \mu$.

EXAMPLE 13: UTILIZATION VS. AVERAGE TIME IN SYSTEM

With the utilization of $\rho = 0.9$ and an average service time $t_s = 0.1$ s we get an overall time that a job (request) spends in a system of $W = 1$ s.

With $\rho = 0.9$ and $t_s = 1$ s we get $W = 10$ s.

With $\rho = 0.99$ and $t_s = 0.1$ s we also get $W = 10$ s.

So, one should keep that in mind when sending requests to a highly utilized system even if the expected average service time is short.

So, utilizations close to saturation have dramatic effects on response times, with implications as described in Section 5.2.3. Even worse, high utilization bears the risk of a **cascading failure** due to self-reinforcing feedback. A detailed discussion about cascading failures can be found in [27]

EXAMPLE 14: CASCADING FAILURE

Consider that all nodes in a horizontally scaled tier are highly utilized and thus all scratch at the configured timeout of the load balancer's health check. If one of these nodes actually misses the timeout, the load balancer rebalances the workload to the remaining nodes. These nodes then get even more utilized, will also miss the response time timeout, and thus will also be deemed unhealthy. The result is a complete failure of that server tier and therefore much likely the distributed system as a whole.

6 Capacity

Capacity²⁷ refers to the number of entities that are (transiently or persistently) stored in limited space. These include records in a database, files on a disk, bytes on a disk, and bytes in memory.

6.1 Database Capacity

Calculations about how much bytes a single entity allocates is not trivial. One must take the actual payload data into account but also

- if there is any **meta-data** (transaction identifier, version identifier etc.) to be stored,
- how information is **encoded**, i.e., German or French texts consume 1.1 byte per character since their special characters are encoded in two or more bytes,
- if any **padding**²⁸ takes place.

EXAMPLE 15: META-DATA AND PADDING IN POSTGRES TABLE ROW

Consider a chat web application where messages are retained. The messages are stored in a Postgres database table. The following SQL script creates such a table, stores a single message with 8 characters in a **table row** and computes the overall allocation in bytes:

```
CREATE TABLE Messages (
  content TEXT, -- 8 bytes: core data to store
  counter INT8, -- 8 bytes: counter for the messages in the chatroom
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(), -- 16 bytes: primary key
  chatroom UUID); -- 16 bytes: foreign key of the addressed chatroom
```

So, besides the core data of the message we have additional data like primary keys, foreign keys or the position of the message in the chronology of messages in the chatroom. These data adds up to 48 bytes. However, if you measure the size of a row with, e.g., <https://pgplayground.com/> and these statements:

```
INSERT INTO Messages (chatroom, counter, content)
VALUES ('b2cd12db-c473-435a-a745-465740f11e3a', 1, 'Message!');

SELECT pg_size_pretty(pg_column_size(Messages.*)::bigint)
AS size_pretty FROM Messages;
```

it gives you 80 bytes because the row is stored like so:

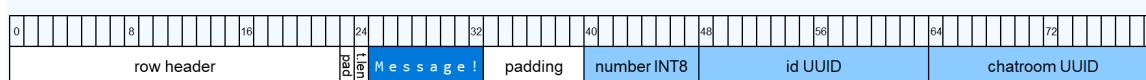


Figure 17: PostgreSQL row

Moreover, we have a **database index** for the id column of the table with 30 bytes for each entry. The following figure sums everything up:

²⁷ Sometimes capacity is also used synonymously with maximum throughput. However, we differentiate between both concepts in this document.

²⁸ Padding aligns data such that every elementary datum starts at a multiple of 2^n bytes. This is because data is addressed and transferred between storage, memory, CPU caches and CPU registers in chunks that have (multiples of) a word's size, e.g., 8 bytes for a CPU register, 64 bytes for a CPU cache line, 4096 bytes for a memory page or a file system block. If a datum overlaps these chunks, the CPU must transfer both chunks and reassemble the datum before it can be put properly in a CPU register. Alignment by means of padding avoids these costly additional steps.

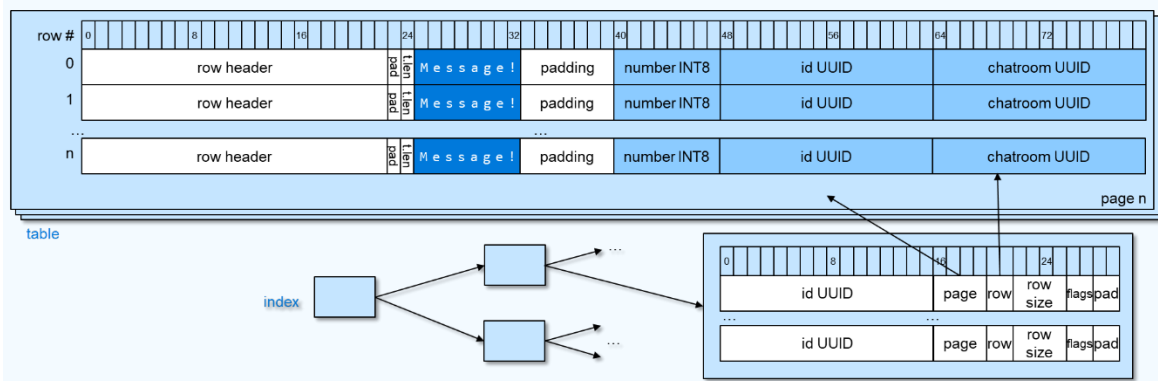


Figure 18: PostgreSQL table and index storage

Overall, we must allocate 110 bytes for an 8-character message. So, no less than 102 bytes are for counters and keys that relate this message to other data, meta-data like a row header and the text length and for padding.

6.2 Memory Capacity

In a broader sense, capacity also includes data structures managed by the operating system or some middleware like

- server-side sessions in an application server's memory,
- concurrent sessions in a load balancer, web server or database,
- threads or child processes spawned by a (middleware) processes, and
- TCP connections which are held by a (middleware) process.

These are artificially limited since their number must be traded off against performance degradation of the respective process.

EXAMPLE 16: MINIMAL PHYSICAL MEMORY PER PROCESS

Processes are abstractions provided by the operating system that have non-zero costs. Even processes for trivial programs have unique set size (USS)²⁹ of about 100 KiB at least. Convince yourself with the following C program `looping.c` that does nothing but looping:

```
int main() {
    while(1); // Loop indefinitely
    return 0;
}
```

Compile it via	<code>gcc -o looping looping.c</code>
make it executable via	<code>chmod +x looping</code>
run it in the background	<code>./looping &</code>
(install smem)	<code>sudo apt-get install smem</code>
and output its USS	<code>smem -P looping -a</code>
(and kill the process)	<code>kill \$(pgrep looping)</code>

²⁹ The unique set size (USS) is the private physical memory allocated to a process.

EXAMPLE 17: PostgreSQL MAX_CONNECTION

Of course, processes that are just pausing are still at the lower end of memory consumption. For example, in a PostgreSQL database, each client connection is handled by a dedicated child worker process, which has a unique set size (USS) of about 5 MB (idle) to 50 MB (processing complex query). That is why the `max_connections` property is not indefinite per default but 100.

6.3 Common Latencies, Throughputs and Capacities

Table 12 provides some rough numbers for fetching data from storage / memory of different types or other basic actions in computing. Please note that the numbers vary within their respective order of magnitude. The definitions of latency, response time, throughput, and capacity also slightly vary depending on the respective storage type or action.

Storage Type / Action	Latency	Latency / Response Time roughly scaled	Throughput	Capacity
CPU registers	0.3 ns	0.3 seconds	-	2 KiB
L1 cache memory	1 ns	1 seconds	1,000 GiB/s	32 KiB
execute typical instruction	1 ns	1 second	-	-
L2 cache memory	4 ns	4 seconds	shared 1,000 GiB/s	256 KiB
branch misprediction	5 ns	5 seconds	-	-
L3 cache memory	14 ns	10 seconds	400 GiB/s	8000 KiB
mutex lock/unlock	25 ns	25 seconds	-	-
main memory (DDR DIMM)	100 ns	1.5 minutes	100 GiB/s	4 GiB – 1 TiB
Non-Volatile Dual In-line Memory with NAND (NVDIMM- N)	100 ns	1.5 minutes	20 GiB/s	32 GiB
mode switch (syscall)	1,500 ns	0.5 hours	-	-
thread context switch	5,000 ns	1.5 hours	-	-
fork thread	20,000 ns	5.5 hours	-	-
process context switch ³⁰	30,000 ns	8 hours	-	-
I/O device on memory bus	50,000 ns	14 hours	25 GiB/s	6,000 GiB
Solid State Drive (SDD)	(random) 150,000 ns	2 days	1 GiB/s	2,000 GiB
fork process	200,000 ns	2.3 days	-	-

³⁰ The actual cost for context switches incurs direct costs for doing the actual switch like register save/restore, page table switch, security checks, scheduler runs, but more so due to indirect costs. The latter arise because of invalidation of or contention for buffers, caches, CPU pipelines, and branch predictor states and vary significantly whether scheduled threads/processes are pinned to certain CPU cores or not [28].

Storage Type / Action	Latency	Latency / Response Time roughly scaled	Throughput	Capacity
send packet within datacenter	3,000,000 ns	1 month	10 GiB/s	-
Hard Disk Drive (HDD)	(seek) 8,000,000 ns	3 months	0.05 GiB/s	4,000 GiB
send packet from US to Europe	75,000,000 ns	2.5 years	-	-

Table 12: Order of magnitudes for common latencies, throughputs and capacities (after [28], [29], [30], and [31, p. 472])

7 Scalability

ISO/IEC 25010:2023 defines scalability as the “capability of a product to handle growing or shrinking workloads or to adapt its capacity [cf. Section 6] to handle variability” [15, p. 8]. A way of dealing with variable workload could be to allocate many resources to the system right from the start. Having, e.g., Google's whole infrastructure for your own system surely would put you on the safe side for a very long time regarding growing workload. Though this would be an effective strategy, it would not be efficient because unused resources would cause costs anyway.

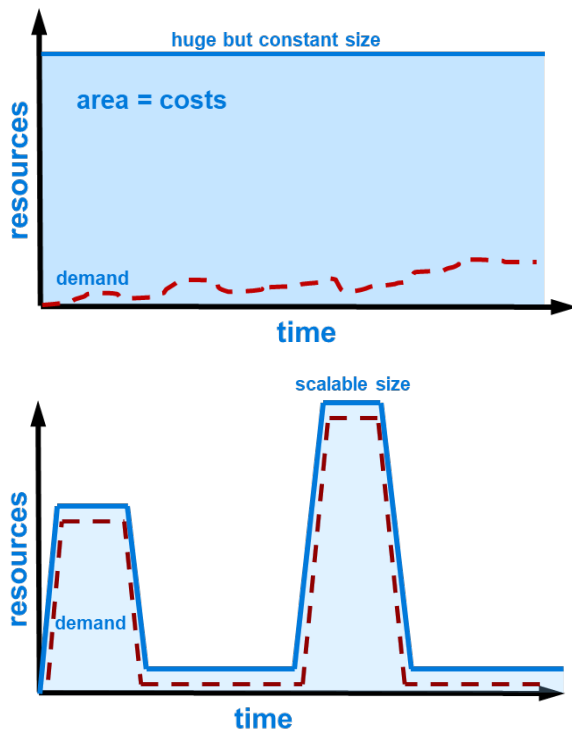


Figure 19: Huge but constant system size vs. scalable system size

If (a) system's resources can easily be allocated and deallocated and (b) performance grows proportionally with allocated resources, a system is said to be **scalable**.

Since allocated resources drive costs, scalability particularly contributes to the **efficiency**³¹ of a system.

7.1 Transactional Requests Workload Profiles

In an ideal world, the workload for a system would be constant. A system could be sized once for all, and the utilization of CPU and RAM resources would be constant for the entire system's lifecycle. However, in reality, workload is far from being evenly distributed at least for transactional request workload. In e-commerce applications like ReWo-Rate, the workload often follows certain patterns:

³¹ Scaling a system's resources in order to fit the workload is not the only contributor to efficiency. Profiling your system and removing bottlenecks in the code base is often a better way.

- **long-term growth:** An e-commerce start-up that rapidly gains new customers / users has to cope with a growing workload in the long run. Moreover, the workload caused by a single web page (re-)load also tends to grow due to technological advances.
- **seasonal peaks:** Christmas, Eastern or Black Friday come along with increased workload for e-commerce application segment. Parcel logistic systems face about twice as much load, web shops with a focus on gifts may have a seven times higher workload.
(Even more **exceptional peaks** occur, e.g., in web shops that sell sparse products like concert tickets from a given point in time in a first-come-first-serve manner. These systems have to cope with a burst of requests from clients of users who want to be among these first to serve.)
- **weekly cycles:** The workload follows a certain pattern throughout the week. For business-to-business e-commerce applications, weekends have much fewer workloads than workdays. Within the workdays, the workload decreases from Mondays to Fridays since Mondays also catch up what happened on the weekend before, e.g., web shop orders.
- **daily cycles:** For business-to-business e-commerce applications, the workload often peaks at 11 a.m. and drops after 6 a.m.

EXAMPLE 18: WORKLOAD PROFILES

Figure 20 and Figure 21 show a workload profile that is typical for a parcel logistics system. They contain all the patterns described above.

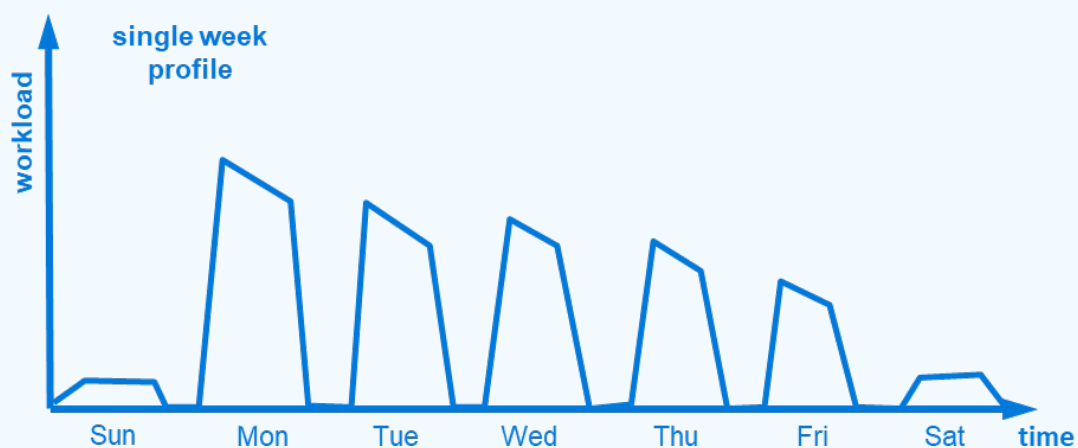


Figure 20: Workload profile for a single week

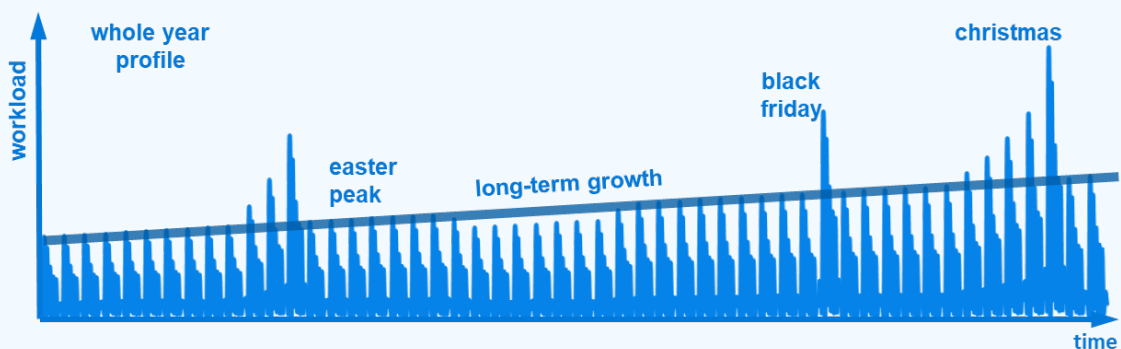


Figure 21: Workload profile for a whole year

7.2 Vertical and Horizontal Scaling

Basically, there are two options for scaling a system:

- One can allocate more resources to the processes (nodes) and the (virtual) machines or containers they run on, i.e., more CPU, RAM, disk space or network bandwidth. This is known as **vertical scaling** of a system or **scaling up/down** a system. Advantages of this sort of scaling include
 - that no additional complexities and problems arise, which come along with distributed systems, particularly with distributed databases, and
 - it works for non-distributed systems, too.
- One can also distribute the workload over processes that run the very same program. This is known as **horizontal scaling** or **scaling out/in** a system. Advantages of this sort of scaling include that
 - there are no natural boundaries like max. CPUs per machine that limit this kind of scaling,
 - availability goes up as described in Section 8.2, and
 - processes can be geographically distributed over different data centers in order to reduce latency or to deal with complete data center outages.

Of course, horizontal scaling works well if these processes run on different machines. A detailed discussion about that can be found in Chapter 18.

Due to virtualization techniques like virtual machines and containers, the boundaries of vertical and horizontal scaling are somewhat blurry: Sometimes, horizontal scaling may happen even within a single, yet heavily equipped bare-metal host machine, i.e., one that contains memory in the order of magnitude of a terabyte, hundreds of CPU cores, is connected to vast storage and to multiple network interfaces. This is because the actual allocation of processes, containers, and virtual machines to bare-metal machines is usually subject to automation conducted by some kind of scheduler processes.

The term **elasticity** refers to the capability of a system to horizontally scale in an automated way, i.e., to react automatically to changes in the current load.

Also note that performance and scalability are not synonyms. A system might have a very good performance for a certain workload but cannot be scaled at all if the workload exceeds a certain threshold.

8 Reliability

ISO/IEC 25010:2023 defines reliability as the “capability of a product to perform specified functions under specified conditions for a specified period of time without interruptions and failures” [15, p. 10]. So, reliability is about correct and timely responses of a system from the perspective of a client or user. There are many levels of how this quality can be violated. A system might always or sometimes respond not at all, not in a reasonable time span or with just an error message³². If it responds, the responses might be plain wrong, wrong by pure chance, or served outdated (stale) from a cache and thus inconsistent. In this chapter we focus on fault states where systems do not respond or with stale data.

8.1 Faults and Failures

Many stakeholders of web applications are concerned about whether a system is operational and produces correct responses all the time³³. However, **faults** in a system occur and lead to a complete or partial system **failure**. Note that a failure of a single system resource might be just a fault from the perspective of the entire system.

- **Hardware faults** are reason for 10% - 25% of data center outages [32]. These faults include faulty hard drives, memory or network resources.
- **Software faults** are another type of faults. These faults include excessive consumption of resources, e.g.,
 - CPU if a process is stuck in an endless loop,
 - memory if there are memory leaks, i.e., memory allocations permanently outweigh deallocations, or
 - network or disk I/O,
 - depletion of certain quantitatively limited abstractions³⁴ like descriptors for opened files, stack frames, or threads.
- **Human errors**³⁵ constitute another class of faults. According to [32] humans are the most likely source of faults. [33] contains some entertaining confessions about human errors with root causes like, e.g.,
 - being in the wrong environment (production), directory (root directory) or being the wrong user (root user) when recursively modifying or deleting files or data from databases,
 - forgetting about WHERE clauses in UPDATE or DELETE SQL statements,
 - locking oneself out of remote maintenance by closing certain ports (like SSH 22), and
 - overutilizing nodes by automatedly generated emails, too frequent expensive SQL queries etc.

³² For web applications with a fault behind the load balancer, the requested response is normally replaced by a response with a HTTP status code 5xx.

³³ Conversely, in high performance computing, a fault is escalated to a failure such that the ongoing computation stops. After repair, it just resumes from the last checkpoint [17]. Complaints from stakeholders in HPC applications are unlikely, nonetheless.

³⁴ Use `ulimit -Sa` to get an overview which limits apply to your current Linux user.

³⁵ Of course, one can consider software fault a subclass of human errors as they are made by human developers.

8.2 Recoverability

ISO/IEC 25010:2023 defines **recoverability** as the “capability of a product in the event of an interruption or a failure to recover the data directly affected and re-establish the desired state of the system” [15, p. 5].

Depending on the impact of the failure, recovering “the desired state” includes replacement of damaged hardware or reprovisioning of lost virtualized hardware, reinstallation of operating systems and programs, reconfiguration, recovery of data from backups, and a successful (re-)start of the programs that constitute the system in question.

A key metric for recoverability is the **mean time to recovery (MTTR)** defined as follows:

$$MTTR = \frac{\text{sum of downtimes}}{\text{number of failures}}$$

MTTR particularly plays a role in the definition of availability as we will see in Section 8.3.

8.3 Availability

ISO/IEC 25010:2023 defines **availability** as the “capability of a product to be operational and accessible when required for use [...] Externally, availability can be assessed by the proportion of total time during which the system, product or component is in an up state. Availability is therefore a combination of faultlessness (which governs the frequency of failure), fault tolerance [...] and recoverability [...] (which governs the length of down time following each failure).” [15, p. 5].

8.3.1 Mean Availability

For sake of simplicity, we assume that a system is either up or down, i.e., it either produces correct responses within required response times, or it does not respond at all. We call these states **uptime** and **downtime**, respectively. A transition from uptime to downtime is a **failure** and from downtime to uptime a **recovery**.

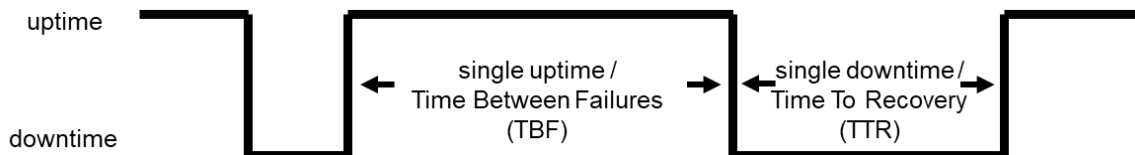


Figure 22: Up- and downtimes

The **availability** of a system is the probability that—for a given point in time—the system produces timely and correct responses.

Mostly, one refers to the **mean availability** A of a system in a given time span, which is

$$A = \frac{\text{sum of uptimes}}{\text{overall time}} = \frac{MTBF}{MTBF + MTTR}$$

Equation 1: Mean availability

where

overall time = sum of uptimes + sum of downtimes

and **MTBF** is the **mean time between failure** in that time span

$$MTBF = \frac{\text{sum of uptimes}}{\text{number of failures}}$$

We consider MTBF to be equal to **mean uptime** and MTTR to be equal to **mean downtime** though actual recovery is just a part of a downtime besides, e.g., detection and analysis of the failure, the time logistics take for getting a replacement part. Please note that the formulae above are not meaningful for edge cases like if the number of failures is zero. They are meant to converge to a meaningful value for long time spans and thus high number of failures.

8.3.2 Nines

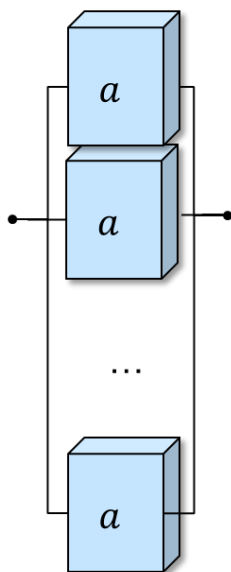
Usually, required availability is expressed as percentage values just containing nines, e.g., 99 % or 99.99 %.

8.3.3 Combined Availability

Often, a system is composed of resources, e.g., nodes, each of which has a certain availability a_i on its own. It is crucial how these resources are combined. If they work in **parallel** inasmuch the workload is distributed over the resources and each resource can serve as a failover for another one, then the combined availability is higher than a_i for each i .

k parallel components

$$A = 1 - (1 - a)^k$$



k serial components

$$A = a^k$$

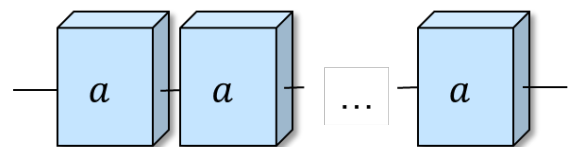


Figure 23: k parallel resources (left) and k serial resources (right)

In the following formula, we assume that each of the k resources have the same availability a . The converse probability $1 - a$ is the probability that a resource is not available, $(1 - a)^k$ is therefore the probability that all k are not available.

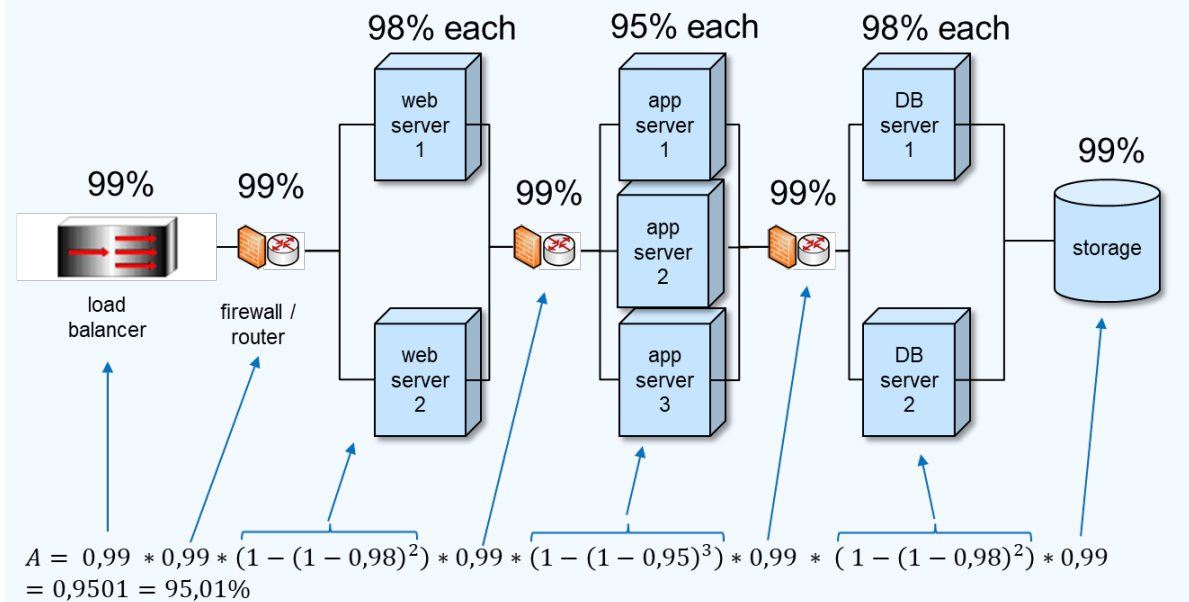
$$A = 1 - (1 - a)^k$$

If the connection of k resources is **serial**, each of the resources must be available if the system is available.

$$A = a^k$$

EXAMPLE 19: COMBINED AVAILABILITY OF A 3-TIER CLUSTER

A typical cluster consists of sequential tiers which contain out-scaled with parallel nodes. Thus, combined availability of the cluster is a mixture of the above formulae.



8.3.4 Planned Downtimes

Planned downtimes are less bad than unplanned downtimes. Planned downtimes of a system X can be scheduled for times of low system usage, e.g., night hours, and announced such that systems, people or processes that depend on X can be prepared appropriately. Such announcements are particularly important if the dependent systems or users are paying clients of a company that runs X.

Planned downtimes normally cannot be avoided entirely as certain maintenance actions like updates cause downtimes. However, in a distributed system updates can often be executed in a way that upholds the availability of the system, e.g., with **rolling updates** for k parallel components as explained in Section 18.7.

8.4 Faultlessness

ISO/IEC 25010:2023 defines **faultlessness** as the “capability of a product to perform specified functions without fault under normal operation” [15, p. 5]. In other words, the faultlessness of a system is the probability that—for the *uninterrupted* duration of a given *time span*—the system produces correct responses. Thus, faultlessness has a subtle different meaning than availability.

In systems where an incorrect response or downtime immediately leads to catastrophic results, faultlessness is more important than availability. Regarding downtimes, low frequency of downtime event is more important here than the average duration of a downtime, i.e., MTTR.

EXAMPLE 20: RELIABLY POWER-SUPPLY

The power supply of hardware is supposed to be reliable as even short outages lead to uncontrolled reboots. Only high availability is not helpful in this case.

Conversely, in systems where the impact corresponds to the duration of a downtime, i.e., MTTR, availability is more important.

EXAMPLE 21: HIGH AVAILABILITY DESPITE LOW FAULTLESSNESS

A client-server system (without transient server-side states) that has a downtime of one second after each 999999 seconds (i.e., a little more than a day), has an availability of $A = 99.999\%$. So, on average, just 0.001% of requests won't get a response.

8.5 Fault Tolerance / Resilience

ISO/IEC 25010:2023 defines **fault tolerance** as the “capability of a product to operate as intended despite the presence of hardware or software faults” [15, p. 5]. Resilience is often used synonymously to fault tolerance. Basically, fault tolerance means that

- certain internal faults within a system can be compensated or
- if system X depends on an external system Y, X can compensate for a failure of Y.

These compensations keep MTTR low and/or lead to a slightly degraded system operational state instead of a complete failure. Compensation methods include the following:

8.5.1 Exception Handling

Certain faults can be treated by additional branches of execution in programs. Many programming languages provide the concept of exception handling that is meant for dealing with such faults. The “degradation” in this case usually is that—though the program still runs normally—a certain function cannot be executed.

EXAMPLE 22: EXCEPTION HANDLING IN JAVA

A file might be opened by some process for reading. If a Java process tries to write to that file, a `java.io.IOException` is “thrown”, which could be “caught” elsewhere in the Java program and, e.g., inform the user by a message dialog that he or she should close the other program.

8.5.2 Failover for Parallel Resources

If a resource like a node fails, then there might be redundant parallel resources that serve as **failover** for each other. Here, one might distinguish two cases:

- **Active-Active:** Resources that are also active in normal uptimes take over the workload of a failed resource. This is usually the case for parallel web servers, application servers or database nodes with **multi-leader replication** in a system cluster. Failover usually works in a completely automated manner, i.e., a load balancer detects the failure of a web or application server and removes this particular server from load balancing. Then, we have a partial failure of the system. It affects just some of the requests between the failure of the server and the reaction of the load balancer, which is typically about 5 seconds. After the reaction of the load balancer, there is a performance degradation since fewer servers must handle the same workload.
- **Active-Passive:** In this case, the failover resources do not get workload in normal uptimes, i.e., they are passive. This is the case for database tiers with **single-leader replication**, in which just one leader node accepts write accesses and the replica nodes only read accesses. Again, failover might happen in an automated fashion. Between the failure of a resource and the completion of the failover, there might be a system downtime. In the case of databases, the MTTR is typically in the two-digit seconds.

Container orchestration middleware like Kubernetes can circumvent performance degradation for an active-active failover by moving processes (contained in so-called pods) running on unhealthy machines to healthy ones. If processes themselves go wild, e.g., by consuming more memory than specified, Kubernetes can get them straight again by just restarting the enclosing pod on the same machine.

8.5.3 Fallback for Serial Resources

Sometimes, a system X depends on another system Y in order to fulfil all or some of its functions. So, these systems are serial resources in the sense of Figure 23. If Y has a downtime, failover to X is pointless. Instead X might implement a so-called **fallback** which surrogates Y.

EXAMPLE 23: CACHE AS FALLBACK (CF. CHAPTER 22)

For cases in which X **pulls** data from Y, responses of Y can be cached by X in normal uptimes. Cached responses can then be used in case Y fails³⁶. Here, the quality degradation of X is the risk of cache misses, which result in no response at all, and hitting stale responses in that cache.

For cases in which X **pushes** data to Y, both systems can be decoupled by replacing a synchronous call with a messaging system (cf. Section 26.1). A quality degradation is that Y might reject messages without letting X know due to the lack of an acknowledging response like in a synchronous response.

³⁶ In case of HTTP, Y can set a `stale-if-error` header in responses as defined by [34]. A intermediate shared cache between X and Y then may reply with this response to X in case of an unavailability of Y even if the response is considered stale.

9 Consistency

ISO/IEC 25012:2008 [35] defines **consistency** as “[t]he degree to which data has attributes that are free from contradiction and are coherent with other data in a specific context of use. It can be either or both among data regarding one entity and across similar data for comparable entities.”

Consistency is particularly broken if clients get different, i.e., contradictory responses for the very same request even if no changes to any data have happened in the meantime. Reasons for inconsistencies are

- **caches** that serve **stale data** or
- **concurrent read and write accesses** to the same data,
- replicated database whose **replicas** have diverged due to network partitions.

In classical, mostly **relational database** systems, consistency is a sacred property and includes **ACID properties** of a database system. ACID stands for

- **atomicity**, i.e., all read/write access to potentially multiple data records either succeed or fail in a single transaction but nothing in between, which makes transactions abortable without the need for clients to compensate the writes that they have done so far in the transaction,
- **consistency**, i.e., every transaction enters and leaves a database in a consistent state, i.e., invariant constraints like that certain values in a data record must be set mandatorily or references (foreign keys) can always be dereferenced, must eventually hold at the commit (end) of a transaction,
- **isolation**, which is more like "consistency in face of concurrent accesses", means that concurrent transactions that overlap in the data they access do not interfere with each other and ideally produce results like if they have been executed serially,
- **durability**, i.e., once committed, a transaction remains committed even in the event of a database failure, i.e., is written on disk.

In many **NoSQL databases**, consistency is partially sacrificed in favor of availability. These databases merely adhere to the so-called **BASE** properties, which stands for

- **basically available**, i.e., a database is highly available,
- **soft state**, i.e., the state of a database might change even if there are no writes to it (due to ongoing synchronizations between database replicas) and,
- **eventually consistent**, i.e., the database finally becomes consistent, i.e., replicas get synchronized, if there are no additional writes.

Although the terms ACID and BASE are widely used, they are not formally defined and a little vague. A more detailed discussion about consistency in databases can be found in Section 33.

9.1 CAP Theorem

The **CAP theorem** summarizes the trade-off between availability and consistency: **Network partitions** that prevent communication between nodes cannot be entirely prevented in distributed systems. For example, in the event of a network partition in a multi-leader database tier, one has to decide between

- **consistency**, i.e., the node detects the partition and reject all requests or at least write requests (thus sacrificing availability) or

- **availability**, i.e., each node still accepts even write requests knowing that its replica cannot be kept in sync with the other (thus sacrificing consistency).

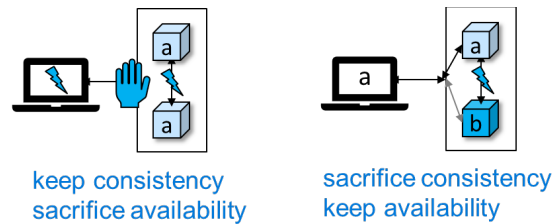


Figure 24: CAP theorem

This dilemma is stated by the CAP theorem in its original form in a somewhat awkward way: One can have at most two of the three: (1) availability, (2) consistency, or (3) partition tolerance [24].

9.2 PACELC Theorem

The **PACELC theorem** restates the CAP theorem and takes latency (in the sense of response times) into account: “if there is a partition (P), how does the system trade off availability and consistency (A and C); else (E), when the system is running normally in the absence of partitions, how does the system trade off latency (L) and consistency (C)?” [36, p. 41].

Scaling out a database tier with replicas results in a better performance since the workload can be better distributed. It also contributes to a higher availability since there are more servers available for failover. On the other hand, having multiple replicas might lead to inconsistency as stated before. So, performance and availability go hand in hand, but both compete with consistency.

10 Maintainability

In general, **maintainability** is the ability to operate a system at runtime or to change it at edit time. It is an umbrella term for more specific terms. ISO/IEC 25010:2023 defines it to be the “capability of a product to be modified by the intended maintainers with effectiveness and efficiency” [15, p. 6].

10.1 Analysability

10.1.1 Observability

Observability is the capability to keep track of a system’s current and past states on different levels of abstraction. **Monitoring** ensures keeping relevant stakeholders (e.g., admins or support) informed about the current state of the system and its event history. Monitoring can be subdivided into functions that are about metrics, logging and tracing, which overlap to a certain extent (cf. Figure 25).

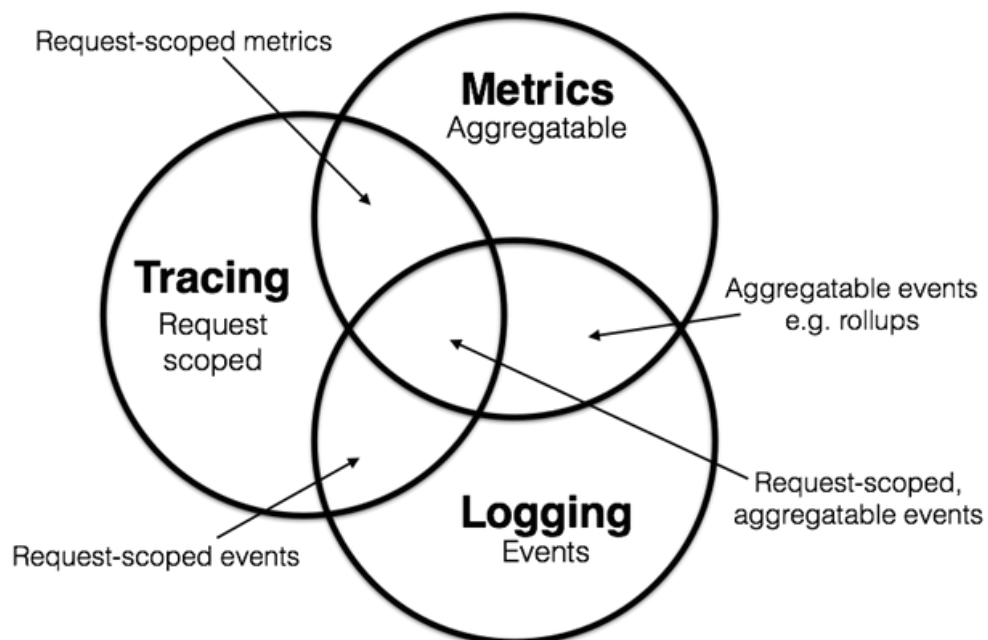


Figure 25: Metrics, tracing, and logging [37]

a) Metrics

Metrics focuses on the current or recent state of processes or machines, presented in an aggregated manner. Common metrics collect and present technical values, like CPU utilization at some point in time, the number of certain events like requests in a certain time span, utilization of some queue at some point in time, (mean) duration of a certain function call etc. **Business metrics** measure, e.g., the overall response times of a certain use case's page flow.

A monitoring tool notifies admins if certain thresholds of capacity or utilization are exceeded. Relevant metrics should be displayed in an aggregated view. Figure 26 shows some metrics as presented by the popular analytics and monitoring tool Grafana (<https://grafana.com/>).

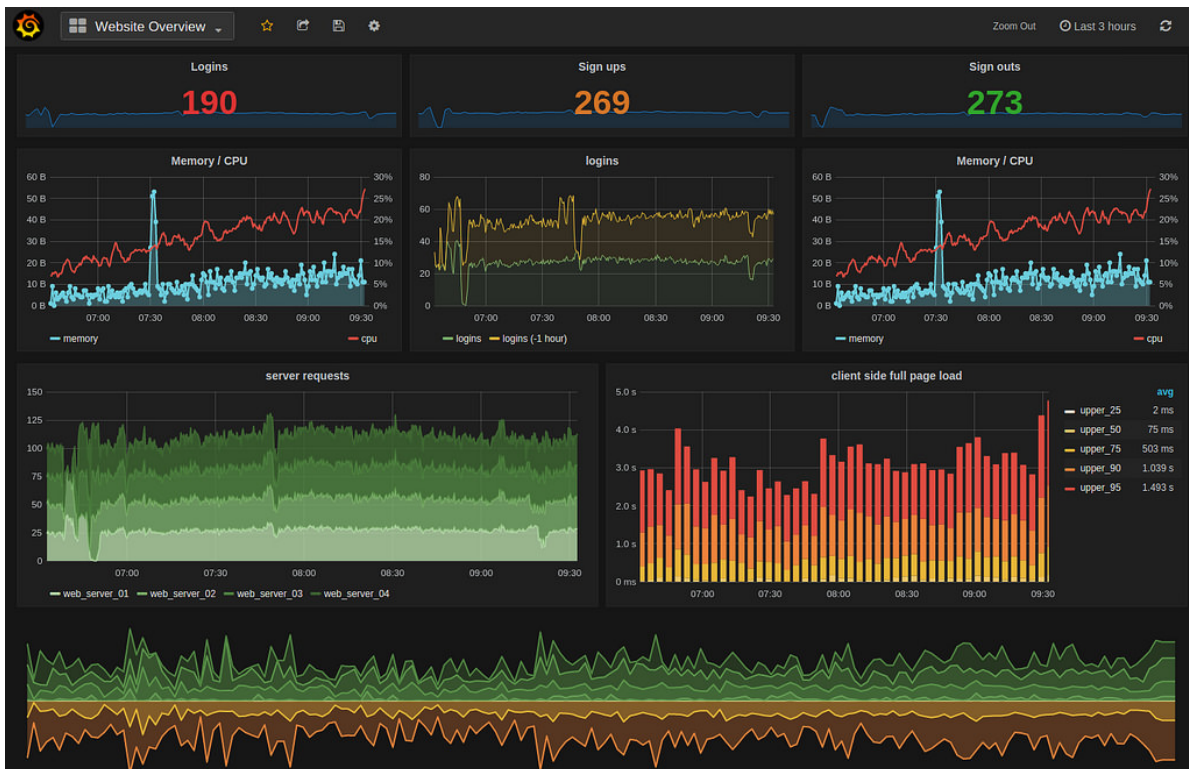


Figure 26: Grafana dashboard by Linux Screenshots (CC BY 2.0)

b) Tracing

A **trace** is the sequence of calls together with certain detailed metrics, e.g., duration of single call, input and output parameters, which are caused by a single request or during execution of one or more system's use case, a.k.a. **user's journey**. In a system landscape, a single trace might include processes of several systems, which call each other.

c) Logging

Logging focuses on the long-term **event history** of a single system containing, e.g., incoming requests, database accesses, outgoing requests to partner systems, completions of internal jobs etc. The aggregation of certain events again might serve as a metric. Logging tools usually provide functions for drilling down into certain **log entries** (i.e. events). Moreover, they allow for searching and filtering according to event attributes like severity or request id. This makes them even suitable for tracing. Figure 27 is a screenshot of the popular logging framework Logstash.

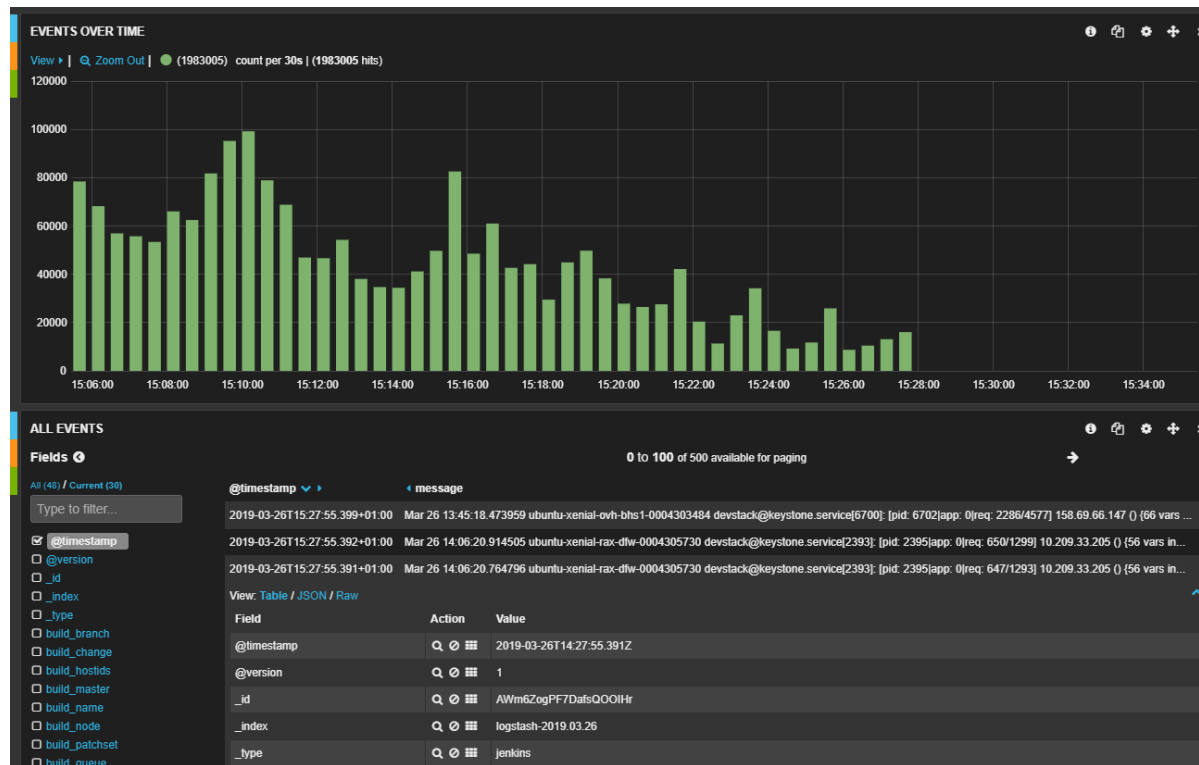


Figure 27: Logstash view

10.1.2 Simplicity

So far, we have dealt with quality attributes that are observable at a system's runtime. Simplicity and modifiability (cf. Section 10.2) are among those quality attributes that pertain to a system's edit time.

If a new developer for some system X needs comparably much time to understand how a certain change (bugfix, new requirement) can be implemented, we consider X **complex**, otherwise **simple**. Sometimes, the complexity of a system is quantified in **technical debt**, often expressed in developer working hours or days necessary to remove that debt from the system. Particularly systems with a broad functional scope (cf. Section 2.2) tend to become complex.

There are some basic mechanisms of human cognition that contribute to simplicity:

a) Chunking

A developer's **cognition** works best if information, e.g., source code, is suitable for **chunking**. This is the case if an inexperienced developer can iteratively group (source code) elements to reasonable abstractions in a bottom-up manner or if an experienced developer can top-down apply memorized chunks to a new situation [38, p. 70]. Moreover, humans can keep just about seven chunks in working memory at the same time [39]. So, to reason about a system as a whole, bigger meaningful chunks are needed.

EXAMPLE 24: CHUNKING OF WORDS

When pupils learn to read, they first recognize characters, then whole words and even sentences. A natural language supports chunking inasmuch there is a distinction between characters, words and sentences. Something like THISSENTENCEISNOTREALLYREADABLE is much harder to grasp than "This sentence is readable."

The following blocks of digits 3901 9411 5498 1049 are hard to memorize in comparison to 1949 1989 1648 2001, because the former is just a sequence of blocks with four random digits while the latter are real chunks which can be interpreted as identifiers for years with important events in history.

b) Schema Building

In cognitive science, a mental **schema** is a combination of abstract and concrete knowledge [38, p. 70].

EXAMPLE 25: MODEL-VIEW-CONTROLLER PATTERN AS SCHEMA

Architectural patterns like the "model-view-controller pattern" are such schemas. Developers normally memorize the abstract nature of a certain schema, e.g., the roles of a model, a controller and a view. They also memorize at least one concrete example, e.g., a concrete application they have developed in which the MVC pattern has been applied.

Of course, the size of an appropriate mental schema is reciprocal to simplicity.

EXAMPLE 26: IMPEDANCE MISMATCH

Impedance mismatch between the programming language / paradigm and the data model of a database in a way doubles the size of a mental schema a developer has to maintain in mind. In other words: He or she must think in two subtle different ways about the data of an application.

c) Hierarchization

Information like source code is easier to grasp and memorize if it is presented in **hierarchical structures** [38, pp. 84–88].

EXAMPLE 27: PROGRAM BLOCK STRUCTURE HIERARCHIZATION

A programming language's block structure, i.e., the possibility to nest blocks of statements into each other, is an example for hierarchization.

Programming languages, principles in software design, software architectures, data structures etc. provide additional examples, where these cognitive mechanisms are supported:

- Principles of Modularity, high cohesion and loose coupling support chunking
- Using a control structure like a for-loop in a proper manner is the application of a schema
- Inheritance hierarchies or aggregation hierarchies are examples for hierarchization.

10.2 Modifiability

Modifiability (a.k.a. evolvability) is about how easy it is to actually implement a change in a system without inadvertently introducing regressions. To anticipate future requirements and to build a system that could easily evolve towards these future requirements maybe is the hardest part of architecture work.

In some situations, simplicity and modifiability go together.

EXAMPLE 28: ENCAPSULATION FOR MODIFIABILITY

If complex data types are properly encapsulated in classes, this normally leads to simplicity and modifiability as changes to the data types (e.g., an additional attribute) have to be made just in a central place.

In other situations, simplicity and modifiability are opposing forces.

Internationalization (i18n), i.e., supporting several natural languages, is a common requirement for web applications. Having string literals in switch-statements with cases for each language all over the source code might be a design decision that promotes simplicity. However, it would certainly hamper modifiability, e.g., the adoption of an additional language. Fortunately, there are lots of frameworks that support modifiability with respect to internationalization for the price of an additional layer of indirection and thus a slightly higher complexity.

Conversely, intentionally adding complexity, e.g., by introducing unnecessary indirections, to cope with very hypothetical future requirements is a common antipattern in software engineering, also called **over-engineering**.

11 Security

11.1 Aspects

Security is the resilience of a system against potential harm deliberately caused by malicious adversaries. In contrast, **safety** is a system's resilience against accidental harm. The German term "Sicherheit" is ambiguous since it translates to both security and safety.

Those system parts that support information-related activities and are valuable resources for criminals are called **assets**. These assets could be confidential data records in databases, files in a file system, or critical soft- or hardware.

Assets are at risk in the presence of a system weakness, also called **vulnerability**. A vulnerability would be harmless if there were no associated threats.

A **threat** is the possible and undesirable outcome of an intentional attack or an accidental event that is facilitated by a vulnerability. Threats can be classified by **thread types** like the ones defined in **STRIDE model** [40]. The following table shows how the STRIDE threats impact quality in terms of ISO 25010.

STRIDE Threat	ISO 25010 Characteristic	Impact	ReWo-Rate Example
Spoofing	Authenticity	Spoofing is the act of falsifying identity. It directly attacks the system's ability to verify that a user is who they claim to be (Authenticity).	A malicious bot spoofs a "Verified Buyer" token to post 100 fake 5-star reviews on a product they never purchased.
Tampering	Integrity	Tampering involves unauthorized modification of data. It directly attacks the system's guarantee that data has not been altered (Integrity).	A shop owner uses an SQL injection to modify the database, changing a 1-star review from "Terrible" to "Terrific" without the reviewer's consent.
Repudiation	Non-repudiation & Accountability	Repudiation is when an actor denies having performed an action. It attacks the ability to prove an event happened (Non-repudiation) and the ability to trace that action to a specific user (Accountability).	A shop admin deletes a negative review and then claims, "I didn't do it; it must have been a system bug," because the system failed to log the deletion event.
Information Disclosure	Confidentiality	Information Disclosure is the exposure of data to unauthorized eyes. It is the direct opposite of keeping sensitive data accessible only to authorized users (Confidentiality).	An API bug reveals the email addresses and order history of all reviewers to anyone who views the "Page Source" of the widget.
Denial of Service (DoS)	Availability	DoS attacks exploit Security vulnerabilities (like a resource leak) to destroy Reliability (Availability)	A competitor floods the ReWo-Rate API with millions of fake rating requests. The security flaw is the lack of "Rate Limiting," but the result is that the widget fails to load for legitimate shoppers (Loss of Availability)
Elevation of Privilege	Integrity + Confidentiality	If a user elevates their privileges, they break authorization and gain access to	A standard "Shop Manager" finds a URL exploit that grants them "Super

STRIDE Threat	ISO 25010 Characteristic	Impact	ReWo-Rate Example
(EoP)		data they shouldn't see (Loss of Confidentiality) or can change data they shouldn't touch (Loss of Integrity).	Admin" rights. They use this new privilege to view the credit card details of the ReWo-Rate subscription owner (Loss of Confidentiality)

A mere threat might materialize in an actual **attack** or **incident**, which is an actual exploitation of a certain system's vulnerability.

EXAMPLE 29: STRIDE THREAT TYPES IN A WEB APPLICATION

Let X be a common web application where different users store confidential information, e.g., credit card numbers. Table 13 exemplifies security aspects from above.

Aspect	Type	Instance / Example
Asset	data	records in database table USERDATA
Vulnerability	injection of source code flaw	Existing source code line stmt.executeQuery("SELECT * FROM USERDATA WHERE userId = " + userId + " AND category = " + category)
Threat	information disclosure (also: spoofing and elevation of privilege depending on the columns in the USERDATA table)	disclosure of user data, e.g., employment status
Attack	SQL Injection	Submitting the value "'whatever" OR 1=1' for category in an unsanitized web input field of the respective system.

Table 13: Security aspects exemplified

Security is a broad field that comprises several overlapping subdisciplines. **Information security** mainly focuses on the asset "data", i.e., the protection of "information or information systems from unauthorized access, use disclosure, disruption, modification, or destruction" [41]. Its purpose is to uphold confidentiality, integrity as well as availability of information also known as the **CIA triad**. **Application security** focuses on the asset "application" with its vulnerabilities in each stage of its lifecycle (design, development, deployment, maintenance or retirement). In **network security** assets are network related.

11.2 Common Vulnerability Types and Attack Types

The example in Table 13 is just one of common threats to typical web applications. The following table lists the "OWASP Top 10 - 2017" risks for web applications [42], where "risk" stands for types of vulnerabilities and attacks.

Risks	Description
Injection	Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
Broken Authentication	Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws in order to assume other users' identities temporarily or permanently.
Sensitive Data Exposure	Many web applications and APIs do not properly protect sensitive data, such as financial and healthcare. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as network or storage encryption, and requires special precautions when exchanged with the browser.
XML External Entities (XXE)	Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.
Broken Access Control	Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.
Security Misconfiguration	Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched and upgraded in a timely fashion.
Insecure Deserialization	Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.
Using Components with Known Vulnerabilities	Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks.
Insufficient Logging & Monitoring	Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

Table 14: OWASP Top 10 Application Security Risks [42]

II. Design

12 Design Goals

In Table 6 we exemplified actual requirements per category. Section 13.1 (Quality Attribute Scenarios) showed us a way to formulate requirements which is particularly suitable for quality attributes.

In early discussions with stakeholders, it is not helpful to delve into requirement details, e.g., negotiating four nines availability. Instead, one should better outline which **goals**, i.e., stakeholder interests, must be met in general. Unfortunately, goals sometime conflict and must be traded with each other. Section 12.1 deals with such trade-offs. With such trade-offs in mind, it is sometimes useful to rate the importance of certain goals to set a focus early on.

EXAMPLE 30: CONTEXT-DEPENDENT DESIGN GOALS

In some situations, the most important goal is to be faster than the competitors, while other situations require a software system of very high quality.

12.1 Trading Goals

12.1.1 Interest Trade-offs

Certain interests tend to oppose each other. First, it is commonly known in product development that speed of development, quality, scope³⁷, and inexpensiveness oppose each other to some degree.

EXAMPLE 31: CHEAPNESS VS. QUALITY (PERFORMANCE)

Being stingy with CPUs might contribute to cheapness but leads to higher utilization of the remaining CPUs (given a constant workload). As we have learned in Section 5.4, this leads to a super-linear increase in response times.

The grey area in Figure 28 reflects the general performance of a development team or whole company: If you increase, e.g., the scope, while retaining the area of the quadrilateral, at least one of the other interests development velocity, product quality, or scope suffers.

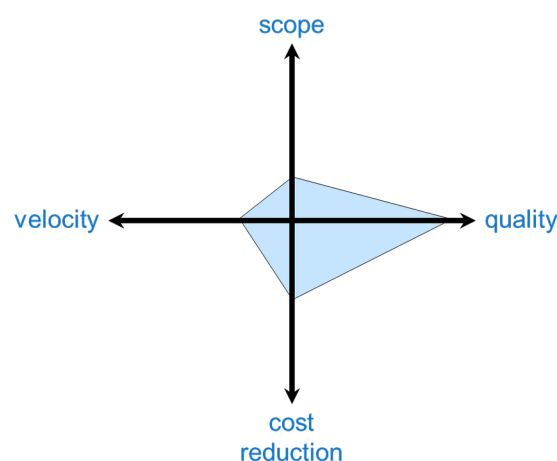


Figure 28: Opposing goals

³⁷ "Scope" refers to the number of implemented functional requirements, which is later on reflected by the number of supported functions in the navigation bar of an application, entity types in the database schema, supported user role etc.

12.1.2 Quality Attribute Trade-offs

The quality of a system is a complex goal that can be broken down to a hierarchy of quality attributes (cf. Figure 11) each of which constitutes a goal a stakeholder group might be interested in. Even with quality attributes, there are opposing forces. In Section 9, we have seen how performance and consistency do not go together. Finding the best compromise for each project and system is one of the most important tasks of an architect. Table 15 depicts some of the conflicts.

Conflicting pair	Rationale or example
performance vs. consistency	cf. Section 9
usability vs. security	A two-factor login increases the security of a system but lowers its usability as it requires a higher effort for each login. So, beware of the recurring requirement: "The system should be highly usable and secure".
performance vs. security	A TLS handshake increases the overall response time.
maintainability vs. speed of development	Often, maintainability of a software product is sacrificed in order to decrease the time-to-market of the first version, disregarding the fact that the speed of development for later versions is severely hampered. Keep that in mind if a stakeholder adds something like "The system must be highly configurable." on the list.
availability/performance vs. cheapness	Scaling out and replicate across different data centers increase availability and even performance but is also a costly thing to do. If a stakeholder requires that "The system must be 99.999% available.", a project manager should figure out beforehand what that means in terms of costs. Cloud pricing calculators like [43], [44], [45] can be helpful here.
performance vs. maintainability	Network messages can be self-descriptive, i.e., include some additional metadata, which improves maintainability but reduces response times due to additional processing time and longer transmission time.

Table 15: Conflicting quality attributes

EXAMPLE 32: TRADING QUALITIES IN ReWo-RATE

With respect to ReWo-Rate, one could trade off these conflicting requirements against each other as follows:

performance vs. consistency: It is not that bad, if not each and every rating is considered during a request for a product's average rating. Taking the average of a suitable, representative sample suffices.

usability vs. security: Ease of use is surely predominant for ReWo-Rate. The actual ratings are not that highly confidential. Possession of a hard-to-guess URL should suffice to be authorized to read or update that rating; two-factor-logins etc. are not necessary here.

maintainability vs. speed of development: This is something to discuss with the management. The case study suggests that ReWo-Rate is not a product that has to be pushed to market with high time pressure.

availability/performance vs. inexpensiveness: One should spend some money in making ReWo-Rate highly available and performing as ReWo-Rate is a kind of product that shines less through neat functions but through high performance and availability.

performance vs. maintainability: Since ReWo-Rate exposes public interfaces consumed by users' clients, both are equally important. However, typically such public systems would not sacrifice self-descriptiveness of their public interfaces for minor performance gains.

As an exercise one could compare that to how you would trade off qualities for a pandemic tracker app.

In stakeholder discussions, it is often helpful to point out such conflicts. Making the downside of a certain requirement explicit ensures that every stakeholder is aware of it.

12.2 Rating Goals

Before delving into requirement details it can be helpful to characterize a system by rating, i.e., prioritizing, common goals. Table 16 does so for ReWo-Rate.

Goal	Rating	Rationale
speed of development	low	ReWo-Rate faces no direct competitors, nor has its release been publicly announced. The time-to-market pressure is therefore below average.
cheapness	medium	Building ReWo-Rate: Due to the lack of time pressure, ReWo-Rate can be developed by a small team with just a small communication and management overhead. Also, there are no special requirements with respect to developer skills which would increase costs. Operating ReWo-Rate: To achieve performance goals, enough money should be spent on ReWo-Rates cloud-based production infrastructure.
size of scope	low	ReWo-Rate is deliberately restricted to a very limited set of functions, entity types, roles etc.
quality		
performance		
response time	very high	Response time directly affects the user experience of systems that are integrated with ReWo-Rate.
throughput	high	In the long run, we expect a high workload (rating requests) that must be processed (stored, aggregated etc.)
capacity	high	Since some products have a long lifecycle, a lot of ratings must be stored.
scalability	high	We expect big variances in the workload (cf. Figure 21) and particularly a steep long-term growth. Therefore, ReWo-Rate should be highly scalable.
reliability		
availability	very high	Downtimes directly affect the user experience of systems that are integrated with ReWo-Rate.

Goal	Rating	Rationale
faultlessness	medium	Frequent downtimes are acceptable if they are very short, so they do not affect availability.
resilience	high	ReWo-Rates is integrated with partner systems on the internet whose availability cannot be controlled. Failures of such partner systems should not lead to a complete failure of ReWo-Rate but at worst to a partial failure or stale data.
consistency	low	At least for transactional data, i.e., ratings, ReWo-Rate may be temporarily inconsistent. For example, it is okay if most recent ratings of a product are not displayed or considered in the computation of an overall rating.
maintainability		The maintainability of ReWo-Rate should be that of comparable systems. There is no particular deviation from a "medium" rating in each of the subcategories.
observability	medium	
simplicity	medium	
modifiability	medium	
security	low	The security requirements for ReWo-Rate can be low (for sake of usability) since stored data are not very confidential except user master data.
usability	high	Since users give ratings mostly as a courtesy, the usability of ReWo-Rate is crucial. A suboptimal usability would drastically lower the conversion rate, i.e., the transformation of a mere visitor to someone who actually does a rating.

Table 16: Rating goals

Such ratings admittedly simplify things since, of course, different system parts could have different goals.

EXAMPLE 33: PERFORMANCE VS. CONSISTENCY

For master data including user passwords, consistency is crucial, while for certain transactional data like ratings in ReWo-Rate the performance prevails over consistency.

Of course, such ratings are susceptible to being highly subjective. The best ratings are done by architects who have enough experience to benchmark a new system against comparable systems. For example, a "high" rating should be interpreted as "high in comparison with similar systems".

13 Design Requirements

13.1 Quality Attribute Scenarios

In this chapter we describe practice for specifying architecturally significant requirements.

Quality attribute scenarios (QAS) as defined by [46, p. 61 ff.] is a common text scheme that particularly suits quality attributes. (Conversely, **user stories** [47] rather suit functional requirements.)

A QAS is a sentence or short paragraph containing the following parts:

- **Stimulus:** An event on which the system (or its developers) must react somehow. Note that "event" is meant in a very generic way. It can be a mouse-click but also a new data protection law which requires modifications in backup processes of the system.
- **Stimulus source:** The emitter of that event. Again, this could be anything or anyone, e.g., some piece of hardware or a product owner or the legislation.
- **Response:** How the system should react. If the stimulus is a HTTP-GET-request for some web resource, the response should contain that respective web resource. If the stimulus is the modification request of a product catalogue in some shop system, the response is the correct implementation of this modification.
- **Response measure:** An observable criteria or event that determines if a response is satisfactory. For latency this might be a value for the 0.99-percentile, for a modification request this might be a test case which must succeed after the modification.
- **Environment:** Additional constraints that apply to the scenario. For latency this might be the current workload on the system.
- **Artifact:** The system or the system's resource to which the requirement applies. If in doubt, the whole system can be referenced here.

The downside of this generic pattern is that each part must be interpreted depending on the addressed quality attribute. In [46, p. 61 ff.] the authors present a **generic quality attribute scenario** for each quality attribute with valid keywords for each of the parts above.

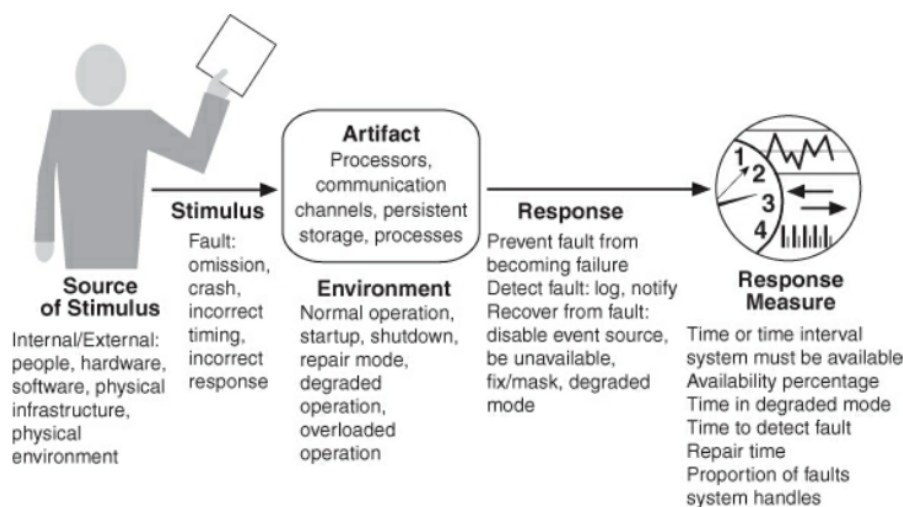


Figure 29: Generic quality attribute scenario for "availability" [46, p. 80]

For the quality attributes of Section 4, the generic QAS would look like this

Quality attribute	Portion of scenario	Possible values
performance	source of stimulus	internal (batch process) or external (clients)
	stimulus	arrival of periodic, sporadic, or stochastic events (like incoming requests)
	artifact	system (or system components)
	environment	operational mode: normal, degraded, peak load, overload
	response	process events, auto scale, notify for manual scaling
	response measure	response time, response time variation (jitter), throughput, dropped events (miss rate)
reliability	source of stimulus	internal/external: people, hardware (servers and network), software, physical infrastructure (e.g., electricity), physical environment (e.g., data center)
	stimulus	Fault: omission, crash, incorrect timing, incorrect response
	artifact	processors, communication channels, persistent storage, processes
	environment	normal operation, start-up, shut-down, repair mode, degraded operation, overload operation
	response	resilience: prevent fault from becoming failure detect fault: log, notify recover from fault: disable event source, be unavailable, fis/mask, degraded mode
	response measure	acceptable availability or faultlessness mean time to detect fault, mean repair time
maintainability	source of stimulus	end-user, product owner, developer, administrator, external system
	stimulus	request for adding/changing/deleting functionality, changing a quality attribute, adapting to new constraints (e.g., new data center)
	artifact	code, data, configurations, misc. resources, logs, processes, hardware
	environment	runtime (normal operation, non-normal operation), edit time
	response	implement, test, deploy change monitor, log, notify
	response measure	estimates (including time and money). how are artifacts affected
security	source of stimulus	internal/external human attacker or system
	stimulus	arbitrary attack vectors, e.g., brute-force password-guessing

Quality attribute	Portion of scenario	Possible values
	artifact	system functions or data (persisted or transient in memory, network buffers)
	environment	operation mode: normal or non-normal connectivity of system: online or offline accessibility: accessible from internet, behind firewall
	response	protect assets from unauthorized access, track/log activities, lock clients, notify administrators
	response measure	duration of attack, number of (resisted) attacks, amount of compromised data, duration of counter measure (e.g., client blacklisting)

Table 17: Rating goals for ReWo-Rate

A **specific quality attribute scenario** can take the suggestions of the generic QAS to formulate a requirement for a specific system.

EXAMPLE 34: SPECIFIC QUALITY ATTRIBUTE SCENARIO

In the following, we exemplify one specific quality attribute scenario each for performance, reliability, maintainability and security that applies for ReWo-Rate.

Performance

(Stimulus source) If a client
(Stimulus) sends a request for the average rating of a particular product
(Artifact) to ReWo-Rate's RESTful web API
(Environment) while ReWo-Rate is in normal operation mode with average workload
(Response) then a HTTP-response containing the average rating and the number of ratings is returned
(Response Measure) with a processing time of less than 50ms in the 0.99-percentile.

Reliability (Resilience fallback)

(Stimulus source) If the SMTP-server for outgoing emails
(Stimulus) is not available for ReWo-Rate
(Artifact) then ReWo-Rate
(Environment) while being in normal operation mode with average workload
(Response) buffers outgoing message and
(Response Measure) and resends them within 10 minutes after the SMTP-server is available again.

Maintainability (Observability)

(Stimulus source) If the SMTP-server for outgoing emails
(Stimulus) is not available for ReWo-Rate
(Artifact) then the logging of ReWo-Rate
(Environment) while ReWo-Rate is in normal operation
(Response) generates a log entry
(Response Measure) instantly with log level ERROR.

Security

(Stimulus source) If an external client
(Stimulus) repeatedly sends invalid login passwords for different users
(Artifact) to ReWo-Rate
(Environment) while ReWo-Rate is in normal operation
(Response) the current IP of the client is blacklisted
(Response Measure) for 1 day and the incident is logged in ReWo-Rate's logging.

13.2 Risks

Risks are pivotal in every development process. Budgets might be cut, key members might get sick, a competitor might scoop on a product under construction and so on. Yet, some of these risks are particularly connected to the system's architecture.

Risk: (condition) Given that ReWo-Rate synchronously calls an external system to verify the validity of an associated transaction / purchase during a rating and this external system is either slow or down

(consequence) then committing a rating might also appear to be slow or even unavailable thus having a negative impact on the reputation of ReWo-Rate for the particular buyer.

Probability: high, since partner systems are accessed via internet which increases the risk for a partner system to appear down or slow

Impact: medium, just those ratings are affected that are associated with transactions in a slow / unavailable partner system

Countermeasure: Defer validation after the commit of the rating. If the transaction turns out to be invalid, so will the rating.

Note the distinction between condition and consequence in the example above. Do not mix them up. To quantify a risk, estimate the probability that the condition actually comes true and the impact of the consequence. Risk is usually defined as:

$$\text{Risk} := \text{Probability} \cdot \text{Impact}$$

If we talk about mere subjective estimates, **probability** and **impact** normally have values from {low, medium, high} instead of numbers.

Countermeasures can be classified be

- **mitigating** the risk like the proposed one,
- **transfer** the risk, e.g., by covenanting a service level agreement (SLA) with each partner system, which is enforced by penalties that would compensate a reputation loss, or
- just **accepting** the risk as-is.

The opposite of a risk is a **chance**: Given a certain condition something good might happen.

EXAMPLE 35: CHANCES IN REWO-RATE

If implemented in Java, ReWo-Rate can be back-ported in ReWo-Rate, therefore avoiding redundant code bases.

So, while thinking about risks also think about chances.

14 Design Principles

Architectural principles, though mostly not directly applicable, lay foundations for good design decisions. They are rational for most architectural patterns and styles. The following architectural principles primarily aim at improving simplicity (cf. Section 10.1.2) and modifiability (cf. Section 10.2).

14.1 Abstraction

The **principle of abstraction** is ubiquitous in computer science: To reason about a system from a particular viewpoint, it is often helpful to leave out irrelevant details.

The best abstractions allow for neglecting the abstracted details not just temporarily but permanently without losing any control because of some automated mechanism that cares about these details.

EXAMPLE 36: ABSTRACTION IN HIGH-LEVEL PROGRAMMING LANGUAGES

A variable in a program written in a high-level programming language "abstracts away" the actual bit-wise representation of a value in memory as well as its actual location (memory address). The compiler and/or runtime environment cares about accommodating variable values in the right form and place in the memory.

The ISO/OSI stack [11] allows for thinking of communicating layers even though actual communication takes place between the bottommost layers. Nonetheless, one can reason about application layer protocols without considering what happens in the layers beneath.

Abstraction is also in the heart of any kind of **modeling**. Here, those attributes that are existent in the modeled original but left out in the model are called "**disregarded attributes**" (in German: präterierte Attribute), those which are only in the model are called "**abundant attributes**".

14.2 Structuring

In information technology, everything is put in some kind of structure. We often encounter **relational structures** which consist of elements and relationships. **Graphs** are special relational structures, which have just binary relationships, called **edges**, among elements, called **nodes** (also: **vertices**). If there is an edge $B \rightarrow A$ for each edge $A \rightarrow B$, edges form a symmetric relationship and graph is called **undirected**, otherwise **directed**.

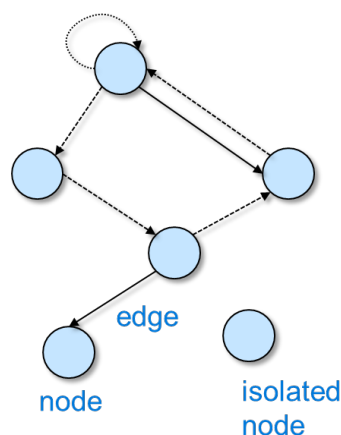


Figure 30: Directed graph with nodes (vertices) and edges

Directed Acyclic Graphs (DAG) are graphs, which are free of **paths** of edges, which form a cycle (like the dotted ones in Figure 30). A node with just outgoing edges is a **source** and a node with just incoming edges a **sink**.

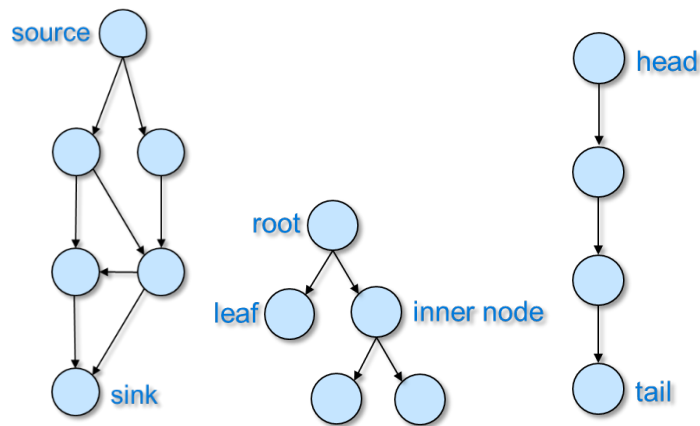


Figure 31: Acyclic graph (left), tree (middle), and list (right)

A subtype of a directed acyclic graph is a **tree** where each node has exactly one incoming edge except the only **source** (**root**) which has none. A node with an incoming and outgoing edges is called **inner node** and a node with just one incoming edge is a **leaf**.

A **linear list** is a tree with just one leaf called **tail**. (The root is usually called **head**. Depending on the definition both might switch positions.)

EXAMPLE 37: DIRECTED ACYCLIC GRAPHS

A Gantt chart in project planning is an acyclic graph. A file system with directories and files is a tree or an acyclic graph if symbolic links are considered as edges as well. The class `ArrayList` in Java implements a linear list.

14.3 Locality

The **principle of locality** demands that coherent units of a system are close to each other. Here, "units" also includes documentation, lines of code etc. "Close to each" other means, that the units are not scattered around different resources (files) or machines.

EXAMPLE 38: LOCALITY IN PROGRAMMING LANGUAGES

In program C, a self-developed method might rely on another one from a ready-to-use framework B, which again depends on another framework A. The developer must jump between three different programs and frameworks each with their own documentation on their own sites.

Violations of the principle of locality due to using many frameworks which introduce many indirections are a commonly accepted downside of contemporary (open source) programming ecosystems. Back-tracing indirections is a recurring task for a developer.

14.4 Strong Cohesion

Cohesion is a measure for the "common ground" of a program's unit (module, component, subsystem etc.). Generally, strong cohesion is desirable.

EXAMPLE 39: STRONG COHESION IN JAVA CLASSES

In a Java program, one could organize attributes and methods by alphabet in classes, e.g., a class A for all attributes and methods that begin with an A. However, these methods and attributes are likely to have nothing in common. For example, it is unlikely that the methods access the same attributes. This is an example for weak cohesion:

```
class MethodsWithA {
    void add(int i) {...}
    void alterName(String newName) {...}
    void acceptCreditCard(String id) {...}
}
```

A class that exhibits strong cohesion would rather look like this:

```
class Set<T> {
    Set unionWith(Set<T> other) {...}
    boolean hasElement(T element) {...}
    ...
}
```

The principles [separation-of-concern](#) and [single-responsibility](#) mandate to achieve strong cohesion by designing program units in a way that they do just exactly one “thing”.

14.5 Loose Coupling

If most modifications of program unit A require modifications of program unit B (or vice versa) then A and B have a [tight coupling](#), otherwise a [loose coupling](#).

EXAMPLE 40: LOOSE COUPLING BETWEEN JAVA CLASSES

Consider the left-hand side of Figure 32 to be a model of a Java program, which consists of six public methods (white boxes), 10 package-protected boxes (blue boxes) and four attributes (black boxes). The outer boxes represent classes, and the arrows represent (call or access) dependencies. If the type or signature of the dependent-on program unit (i.e. the target of an arrow) is changed, the dependent program unit (source of an arrow) needs to be adapted, too. Structuring a program like this leads to a tight coupling between classes. This tight coupling is reflected by the many arrows in Figure 32 which cross class boundaries. Moreover, the cohesion is weak as the units within the classes apparently have no commonalities.

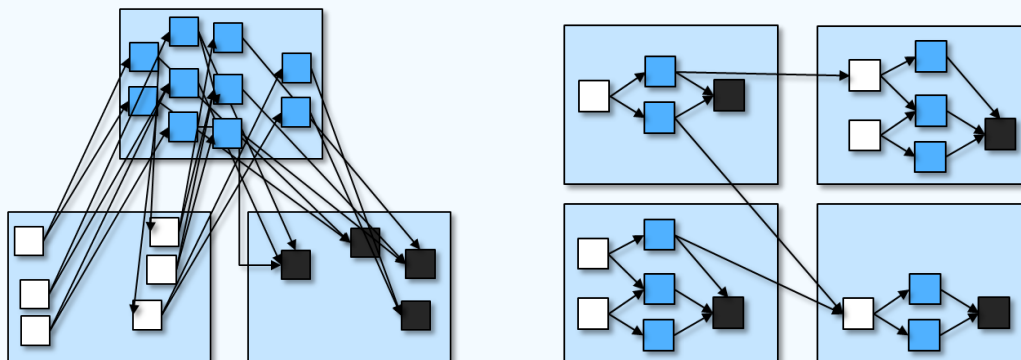


Figure 32: Badly modularized (left) and well modularized system (right)

In Figure 32 on the right-hand side, we have the very same program with the same dependencies just with a different assignment of the methods and attributes to classes. Here, most dependencies are contained within classes and just a few are between classes so in comparison, we have a strong cohesion and loose coupling here.

14.5.1 Dependencies

Loose coupling is characterized by

1. a low number of **dependencies** between program units,
2. a high **locality** of the dependencies (this is what makes the left-hand side different from the right-hand side in Figure 32),
3. how **explicit** such dependencies are modeled, which particularly eases the detection of violations,
4. if violations are **automatically checked** (by a compiler, linter, runtime environment) and
5. if they are **checked early**, e.g., during editing or build-time, so before runtime.

The properties 2 to 5 make a certain dependency in a way **benign**, their absence **malignant**.

EXAMPLE 41: BENIGN DEPENDENCY BETWEEN PROGRAM UNITS

A benign dependency in that sense would be the static invocation of a method by another one:

```
class A {
    static List<Animal> search(String criteria) {
        List<Animal> result = db.lookup(criteria);
        return result == null ? Collection.emptySet() : result;
    }
}

class B {
    void printMammals() {
        for (Animal animal : A.search("mammal", 23)) {
            System.out.println(animal.toString());
        }
    }
}
```

Here, the Java compiler can detect the mismatch between the declaration of the search method and its invocation with one more parameter. Since modern IDEs have built-in programming language support or make use of IDE agnostic language servers [48], developers are notified even earlier about this mismatch that is during editing time.

A less benign dependency stems from the assumption that search always returns a (potentially empty) List-object, so the caller can skip null-checks here. This dependency could be violated by changing the body of the search method, i.e., by actually dispatching the null value. This would be typically detected during tests of the respective program where the Java runtime environment would throw a NullPointerException (NPE).

The most malignant dependencies are those whose violations are left undetected for a long time even if they made their way to the production system. Mutual assumptions about data schemas and formats are among these dependencies.

EXAMPLE 42: MALIGNANT DEPENDENCY DUE TO WEAK TYPING

Consider a database column for sums of money in dollars where the values are strings (VARCHARs in SQL speech). Readers and writers to that column have particularly to agree on what character is a decimal separator and what is for grouping digits (comma or decimal point). Undetected disagreements can lead to financially disastrous consequences in this case.

One way to make such dependencies less malignant is to make such mutual assumptions explicit. This can be done by [explicit database schemas](#) (cf. Section 32.1.1 and [strong programming languages types](#) whose violations by readers or writers can be detected early, e.g., compile time.

Please note that the “runtime usage” of a dependency does not account for tight coupling. For example, if a method call is wrapped in a loop with one billion iterations, this coupling does not get stronger. So, the maintainability of a system is not affected. However, the performance would be affected particularly if the call is a remote method call.

14.5.2 Connascence

A more sophisticated way to categorize dependencies and dependent program units is the notion of [connascence](#) [49]. In [50], the authors identify three properties that contribute to connascence:

a) Strength

Strongly connascent dependencies are harder to detect. Moreover, with respect to mere programming languages, the strength can be classified by the following categories, where “name connascence” is the weakest and “identity” the strongest form of connascence. Please note that this is just a rough ordering: Not every “algorithm connascence” must be considered stronger than every other “position connascence”.

▪ **Static connascences**

- **Name**: several program units must agree on the name of a program unit, e.g., if the name of a method is changed, callers must be adapted as well
- **Type**: several program units must agree on the type of a program unit, e.g., a caller and a callee must agree on the type of a passed parameter or return value
- **Meaning**, e.g., the meaning of status codes
- **Position**, e.g., in Java, callers adhere to the positions of the formal parameters when calling a method
- **Algorithm**, e.g., a caller and a callee have to agree on, e.g., a certain encoding like UTF-8 or encryption, hashing or compression algorithm for certain exchanged data

▪ **Dynamic connascences**

- **Execution**, e.g., if the invocation order of different methods of a callee is important. The following example would cause undesired behavior, where the set last name gets lost.

EXAMPLE 43: DYNAMIC EXECUTION CONNASCENCE

```
SaveableEntity se = new SaveableEntity();
se.setFirstname("John");
se.save();
se.setLastname("Doe");
```

- **Timing**, e.g., if two program units concurrently write to a single value and must be synchronized in order to avoid race conditions
- **Value**, i.e., if several values must be changed together. In the following example, the area of a rectangle is not derived from the height and width but must be set explicitly, if height and width are changed.

EXAMPLE 44: DYNAMIC VALUE CONNASCENCE

```
Rectangle rec = new Rectangle(3.0, 4.0, 12.0);  
rec.setHeight(5.0)  
rec.setArea(20.0)
```

- **Identity**, i.e., if several program units have to reference the same entity.

Code refactoring is supposed to weaken connascent. For example, a HTTP status codes like "200" can be defined in Java by

EXAMPLE 45: WEAKENING CONNASCENCE

```
enum HttpStatus {  
    OK("200"),  
    CREATED("201"),  
    ...  
}
```

b) Degree

The degree of connascent is a metric for the number of dependencies a program unit has.

c) Locality

The locality of connascent program units, where dependent units that are close to each other are easier to refactor. The following list exemplifies what we mean by "locality" of program units (statements, methods, classes) from close to distant:

- statements in the same block
- methods in the same class
- class in same package / namespace
- program units in the same program
- program units in different programs but the same code base (mono repository)
- program units in different code bases

d) Strength and Degree vs Locality

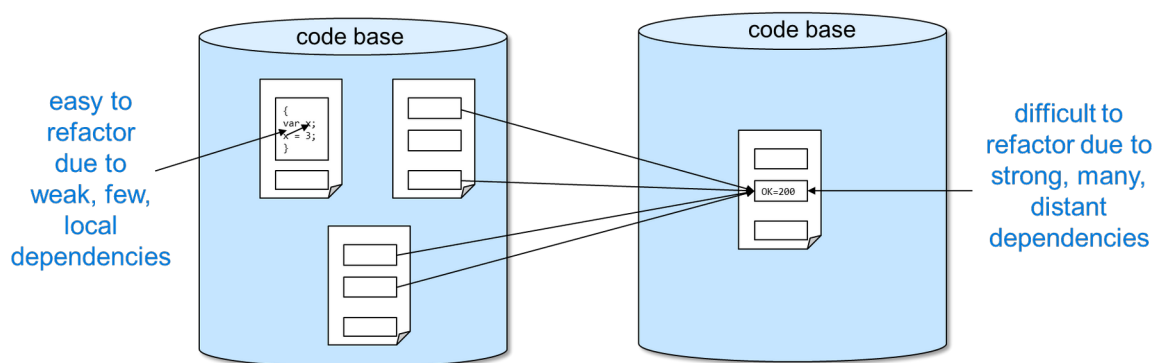


Figure 33: Strength and Degree vs Locality

Refactoring of strong connascent dependencies is easier if locality is high. For example, changing the name of a local variable in a block is easy to do; the impact is limited by the scope of the variable which is the block.

EXAMPLE 46: LOCALITY AND REFACTORING

```
{  
    int tpyo;  
    tpyo = 3;  
    tpyo = tpyo + 2;  
}
```

Conversely, if the developers of a Java Spring framework would decide that an OK response now should be represented by status code "0200" instead of "200" then a lot of other dependent code bases would be in trouble. In fact, such a refactoring would not be feasible since all dependent code bases cannot be determined at all.

14.6 Information Hiding

Modularizing a system like in Figure 32 (right) not only keeps coupling loose but also follows the principle of information hiding. A class may only depend on public methods (white boxes) of other classes. The other members are hidden and if changed, might only break members of the same class. This is also known as [encapsulation](#).

15 Design Patterns

This chapter provides a very high-level overview on design patterns.

15.1 Patterns on Different Levels

Principles like those of Chapter 14 lead to certain recurring patterns (a.k.a. strategies, tactics, methods, practices), which are applied when software is built. Patterns are self-imposed constraints about how to build a software system. They improve qualities like performance, consistency and maintainability as they support systematics. Patterns can be classified according to the level of granularity on which they are found:

- **Architectural patterns** and **styles** are about how a system or system landscape is distributed and how the distributed processes interact with each other, e.g., in a peer-to-peer manner via messaging. Sometimes they refer to a profound design decision in a single system which is hard to change afterwards, e.g., organizing the code base in a three-layer software architecture and implementing the frontend in a model-view-controller variant.
- **Design patterns** are about proven solutions for detailed software design. They describe how to organize the source code of a single program to tackle a certain recurring problem. For example, leveraging information hiding by using the design pattern "façade" is common practice [51, p. 185 ff.]. We use this term "design pattern" in this narrow meaning; in literature, the meaning of "design patterns" sometimes also covers what we mean by "architectural patterns". For example, "Cloud Design Patterns" of [52] are rather architectural patterns in our sense.
- **Idioms** are at the level of constructs of a concrete programming language. They provide solutions for common programming tasks like filtering a list according to a certain criterium and include programming best practices like yielding an empty list instead of null.

Table 18 further characterizes these levels.

	Architectural patterns and styles	Design patterns	Idioms
characteristic	fundamental organization	proven solutions for recurring design tasks	proven solutions for recurring implementation tasks
Subject	- distributed system - system of systems	- single program - component in a program	- single program unit, e.g. - classes - bodies of methods
specific for	information technology	programming paradigm (e.g., object oriented)	programming language
Examples	- 3-layered architecture - client/server model - monolith or microservice - messaging - MVC pattern	- observer pattern - factory pattern - singleton pattern	- "Hello World" - iteration through map - reverse string (http://www.programming-idioms.org)

Table 18: Patterns on different levels

15.2 Catalogues and Languages of Patterns

Patterns in computer science came up as a research subject since the 1990ies and since then found their way into industrial practice in one way or another. Unfortunately, each work about patterns comes along with its—sometimes tacit—focus area and taxonomy. Moreover, there is a tendency in these works to reframe every topic as patterns like, e.g., "caching", "sharding" or "materialized views". Since such topics will be touched in these lecture notes anyway without referring to them as patterns, we refrain from delving into each pattern in this chapter but discuss some very fundamental patterns in the following sections and an overview of some important works in the pattern field in Table 19.

Work	Pattern Level	Focus Area
"Design Patterns – Elements of Reusable Object-Oriented Software" by Erich Gamma et al. [51]	Design patterns	Patterns in programs written in an object-oriented programming language.
"Domain Driven Design" by Eric Evans [14]	Design patterns / architectural patterns	Patterns for decomposition of large systems with complicated business domain. Mostly used in conjunction with microservice architectures nowadays.
"Patterns of Enterprise Application Architecture" by Martin Fowler [53]	Design patterns / architectural patterns	Challenges of data-driven, multi-user enterprise applications like state management, concurrency and interplay between programs and databases.
"Enterprise Integration Patterns" by Gregor Hohpe [54]	Design patterns / architectural patterns	Messaging between nodes in distributed systems
"Cloud Design Patterns" [52]	Architectural patterns	Distributed systems, cloud computing
"Designing Distributed Systems" by Brendan Burns [55]	Architectural patterns	Distributed systems, cloud computing
"Microservice Patterns" [56] and "A pattern language for microservices" [57] by Chris Richardson	Architectural patterns	Distributed systems, microservice systems
"Fundamentals of Software Architecture" [58]	Architectural styles	Software architecture styles

Table 19: Literature about patterns

16 Modeling Systems

Large and distributed software systems quickly become too complicated to learn and reason about them directly. This is why modeling is a crucial technique in every day's software engineering.

16.1 Models and Originals

A **model** helps to understand its so-called **original**. An original can be anything; however, in this context we are only interested in originals that are software systems. More precisely, we are interested in those parts of a software system that people actually work on and reason about. This includes source code in some high-level language, the runtime behavior of a program in terms of an abstract machine, the deployment of artifacts (files) to file system of nodes etc. Conversely, models of machine code that is automatically generated by a compiler and its low-level execution in conjunction with an operating system kernel are uncommon. Such parts of a software system are usually abstracted away from developers (let alone end-users), anyway.

If the system in question is still under construction, **prescriptive models** show how the later system should look like and behave. Conversely, **descriptive models** have an already existent original, in this case the software system in question. They are used to make that original more comprehensible.

In practice, modeling is neither entirely prescriptive nor descriptive. Usually, modeling and implementation of the original goes together: One starts with a rather imprecise prescriptive model, then implements a single use case to get familiar with constraints that come along with middleware and programming languages and frameworks that are used for the system. Learnings are fed back into the more descriptive model which are fleshed out for other not yet implemented use cases in prescriptive parts of the model.

16.2 Model Properties and Qualities

A model has certain properties or qualities which range along four dimensions as depicted in Figure 34. In that figure, we see variants of models of different ambiguity, precision, and correctness for a single and simple Java class.

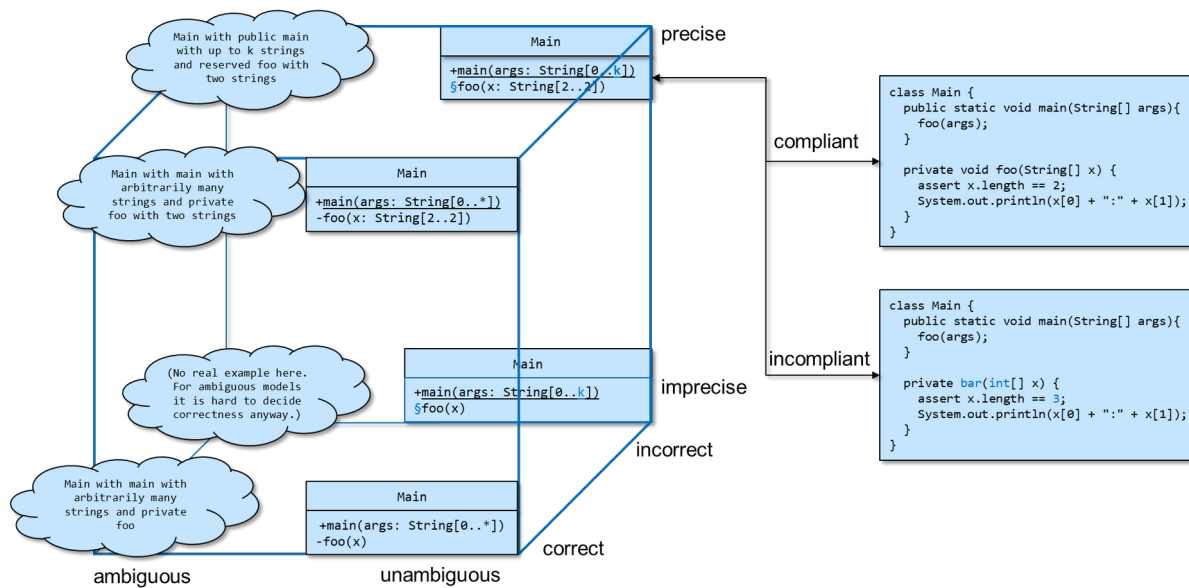


Figure 34: Quality dimensions for models

The quality dimensions are described further in the following.

16.2.1 Precision of a Model

Usually, a model has not just one compliant original. Usually, there is an (infinite) set of originals that (would) comply with a certain model. This set becomes smaller if the model is of high **precision**. Precision increases if the model includes more details that further constrain compliant originals.

EXAMPLE 47: MODEL PRECISION

The following UML classes model a Java class `Main`. However, the left one leaves out constraints about the type of parameter `x`, which is correct UML yet not precise. Thus, the right model is more precise.



Figure 35: Precision in models

Please note that higher precision is not always desirable. Depending on the current understanding of the original, lower precision is often more appropriate.

16.2.2 Scope of a Model

While precision delimits the “depth” of a model, the **scope** sets boundaries for its breadth. Defining a scope is important as modern software systems are normally embedded in a larger system landscape of systems they communicate with if not the whole internet. The scope delineates what parts of the overall original should be included in a single model.

16.2.3 Unambiguousness of a Model

If a model is **ambiguous**, it remains unclear how a complying original may look like and if an existing original actually complies with the model. In other terms, the set of complying originals

has blurry boundaries. Often, so-called marchitecture diagrams, a portmanteau of architecture and marketing, are ambiguous as their main purpose is to impress their readers by mere aesthetics to support the marketing of, e.g., some kind of middleware product.

Note that unambiguousness is different from precision although the difference is often subtle: With lower precision, the set of complying original increases yet is still well-defined. Contrary, with ambiguous models there are originals for which it is not clear if they comply with the model or not.

EXAMPLE 48: MODEL UNAMIGUOUSNESS

In Figure 36 we again see two variants of a model that are supposed to capture the same original. The left uses natural language, while the right is a UML class. The left is more ambiguous as it is not even clear what Main actually refers to. Even if we assume that Main is a Java class, it is not clear if foo is a method or an instance member variable.

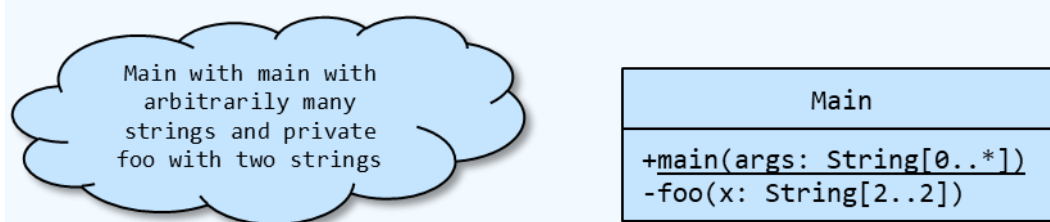


Figure 36: Unambiguousness in models

Unambiguity can be established by thorough descriptions of how elements in a model (will) map to the original.

16.2.4 Correctness within a Model

Irrespective of an original, models can be incorrect in themselves, i.e., they can include errors. If a model is supposed to comply with a (semi-)formal modeling language, syntax errors are possible. Other errors range from inconsistent names to very subtle errors in the model that make the set of matching originals empty.

EXAMPLE 49: MODEL CORRECTNESS

The following UML class is incorrect by itself as the multiplicity $0..k$ is incorrect (should be $0..*$), the visibility modifier $\$$ is not defined by the UML.

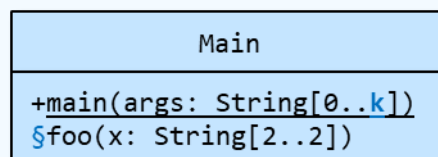


Figure 37: Incorrect UML class

16.2.5 Compliance between an Original and a Model

A descriptive model is supposed to include the existent original in its set of modeled originals. If that is not the case, we say that the model is not compliant with the original. If a prescriptive model comes first, and the original fails to be in the set of modeled originals, the original is not compliant with the model. In Figure 34 you can see both cases. The lower original deviates in the method name and parameter type and does therefore model and lower original do not comply with each other.

Unfortunately, it is all too common to leave a model intentionally ambiguous in order to circumvent noncompliance. It is very much up to the own discretion of individuals to determine compliance. No common understanding³⁸ can be achieved this way.

16.3 Decomposition

Decomposition is a common technique in system modeling. For example, modeling might start with a context diagram, where the system in question is just one model element which is surrounded by all systems and users it communicates with. So, a context diagram is deliberately imprecise: Many compliant originals are possible that implement the system in one or another way.

On the next layer of decomposition, we “zoom into” that model element and create another diagram which shows the inner components of that system yet still on a coarse-grained level. Here – say – each model element corresponds to a microservice that contributes to the system in question.

The same can be done for other viewpoints as well: On the most coarse-grained level one might model the system in question and partner systems each as a single UML lifeline in a UML sequence diagram and show how the systems communicate with each other. On a lower level, one can show how the system’s microservices interact. Even deeper, one can depict how the program units, e.g., classes, within a single microservice call each other.

EXAMPLE 50: MODEL DECOMPOSITION OF ReWo-RATE

In Figure 38, we have a UML component diagram in the upper left corner that is a context diagram. Here, every system is modeled as a single component including the system in question ReWo-Rate. Moreover, just those interfaces are included which are accessible by other systems. A context diagram is on decomposition level 0. We assume that ReWo-Rate by itself is comprised of two microservices. Decomposition level 1 exactly shows these microservices as stereotyped components. The next decomposition level 2 details one of these microservices and depicts the tiers (cf. Section 18.2) with their mutually independent deployment artifacts. Note that we have switched to a more appropriate UML deployment diagram on level 2. On decomposition level 3, we arrive at a diagram that models the code which can be found compiled in the artifact.

Accordingly, we can also decompose models of other viewpoints like a UML sequence diagram that models the runtime behavior of ReWo-Rate. In Figure 38, we break down the ReWo-Rate lifeline on decomposition level 0 and show how the microservices interact on decomposition level 1.

³⁸ Pseudo-compliance via ambiguity plagues other fields as well. For example, a project plan in form of a Gantt chart (model) is ambiguous, if the individual tasks are ambiguous. For example, it is useless to have a definite end date for a task T if there are not objective criteria exist if T has been completed in the project (original). A mere label like “test software”, which is typical does not provide such an objective criterion.

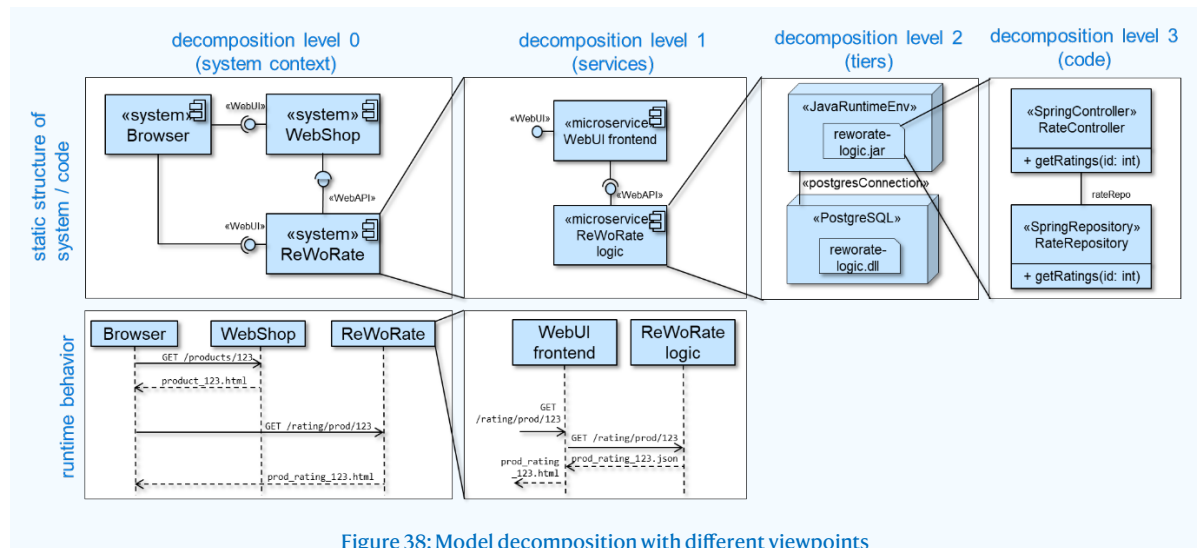


Figure 38: Model decomposition with different viewpoints

The granularities of the decomposition levels should be balanced and chosen with respect to stakeholders. For example, members of the development team (cf. Figure 9) of a partner system (like “WebShop”) are mostly interested in decomposition level 0 (system context) as they do not care of the inner workings of ReWo-Rate. System and enterprise architects may be interested in the distribution of a system into microservices on decomposition level 1. The Ops team for ReWo-Rate should be aware of what tiers need to run for ReWo-Rate, so they should also know ReWo-Rate on decomposition level 2. Actual developers of ReWo-Rate are interested in the structure of the programs that constitute microservices, so they need to know decomposition level 2.

16.4 Viewpoints

In models, details of the original are left out intentionally. Models often take a certain **aspect** or **viewpoint**, which prescribes what kinds of details that are relevant for a certain view onto an original and which can be left out.

EXAMPLE 51: VIEWPOINTS

A model of the structure of a certain program does not need to show how the program behaves on certain inputs, how the program is deployed nor how the program communicates with others in a distributed system.

Literature provides diverse compilations of viewpoints, like the 4+1 view model by Kruchten [59], or the viewpoints by Rozanski and Woods [60]. In the terminology of Rozanski and Woods [54] a **view** is the application of a viewpoint to a concrete original.

Viewpoint	Description	Notations
Functional	Focuses on the functionality a system provided, how the functions are related to each other and to external actors (users or partner systems) Typical stakeholders: product owner, teams of partner systems, architects	- UML use case diagrams - Prose text that refines a single use case. - UML activity diagrams that refine a single use case
System's Static Structure	Focuses on the logical building blocks (components) of a system including its context, their interfaces and their dependencies. The UML component diagrams of Figure 38	- UML component diagram together with a detailed explanation of how a certain component is manifested in the modeled original.

Viewpoint	Description	Notations
	supports this viewpoint. Typical stakeholders: developers, enterprise architects, system architects	- WebAPI specifications that formally specify exposed APIs (like in OpenAPI, Web Service Description Language (WSDL), XML Schema Definition (XSD) etc.)
Code Base Structure	Focuses on the static structure of the system's code base, e.g., the program units (classes, methods, packages etc.) and how they are related. The UML class diagram of Figure 38 in the upper right corner provides an example. Typical stakeholders: developers, software architects	- UML component diagrams. (Again, additional information must be given here, how such components map onto the source code, i.e., what a component actually is (supposed to be) in the code base.) - UML class diagrams. (In case of an object-oriented programming language the mapping is mostly self-explaining.) - UML package diagrams, which show how namespaces in a code base are related.
Data Schema	Focuses on the shape of the data that the system processes and stores. Typical stakeholders: developers, software architects	- UML class diagrams - Entity Relationship diagrams - Tables Sometimes, files that are directly used in the implemented original (system) also count as suitable models like: - SQL Data Definition Files - XML Schema Definitions (XSD)
Runtime Behavior	Focuses on common and exemplary event sequences during the execution of some use case, call orderings of microservices or program units, allowed states and possible transitions etc. Typical stakeholders: developers, software architects	- UML sequence diagrams (cf. example at the bottom of Figure 38) - UML communication diagrams, which are basically equivalent to UML sequence diagrams but noted differently (i.e., UML sequence diagrams looked from "above") - UML state charts which are used to model states and possible transitions. - UML object diagrams that exemplify a system's memory state in a certain situation
Deployment	Focuses on - which programs and other software parts, identified by the systems static structure and code base structure, will be "manifested", i.e., compiled and bundled into which files or directories, - which of these programs run on which nodes and in which runtime environment, - how these nodes communicate (network protocols), - where data is stored and accessed by the programs Typical stakeholders: system administrators, system architects	- UML deployment diagrams

Table 20: Common System Architecture Viewpoints

16.5 The Unified Modeling Language (UML)

The Unified Modeling Language (UML) as specified in [61] defines the **abstract syntax** for diagrammatic models as well as the **concrete syntax**, e.g., visual representation of model elements.

Although it is possible to make use of any of the UML model elements in one model, it is common to stick to a certain UML diagram in which just certain UML model elements occur, and which support a certain viewpoint. For example, in UML class diagrams we find UML classes and UML associations but not UML artifacts, which can be found in UML deployment diagrams.

Figure 39 provides a taxonomy of all UML diagrams. The figure itself is a UML class diagram with classifiers and generalization relationships between them. Note that there are two main categories of diagrams:

Structure diagrams describe the static structure of a system, i.e.,

- the structure of program units in a program's source code,
- a snapshot of a process, i.e., a (part of a) single state of running program (process) at a given time expressed by the current values of variables,
- files (artifacts) that contain programs, configuration files etc. and by which processes they are used and where they are being deployed etc.

Conversely, the **behavior** of a software system can be understood as the possible chronologies of events at runtime, which are input messages fed into the system, output messages that are produced by the system, timer events within the system and the like which all cause transitions from one system state to the next.

Even in trivial systems, the set of possible states and atomic transitions is enormous. It is therefore infeasible to model these as-is in a transition system. This is still true, if

- only parts of a system are regarded, e.g., the lifecycle of a certain data object and
- states are composite, e.g., stand for several states and transactions that are actually being traversed like a statement in a high-level language is broken down to several in machine code.

Behavior diagrams therefore not only focus on parts and aggregate but also take a certain viewpoint to model the behavior.

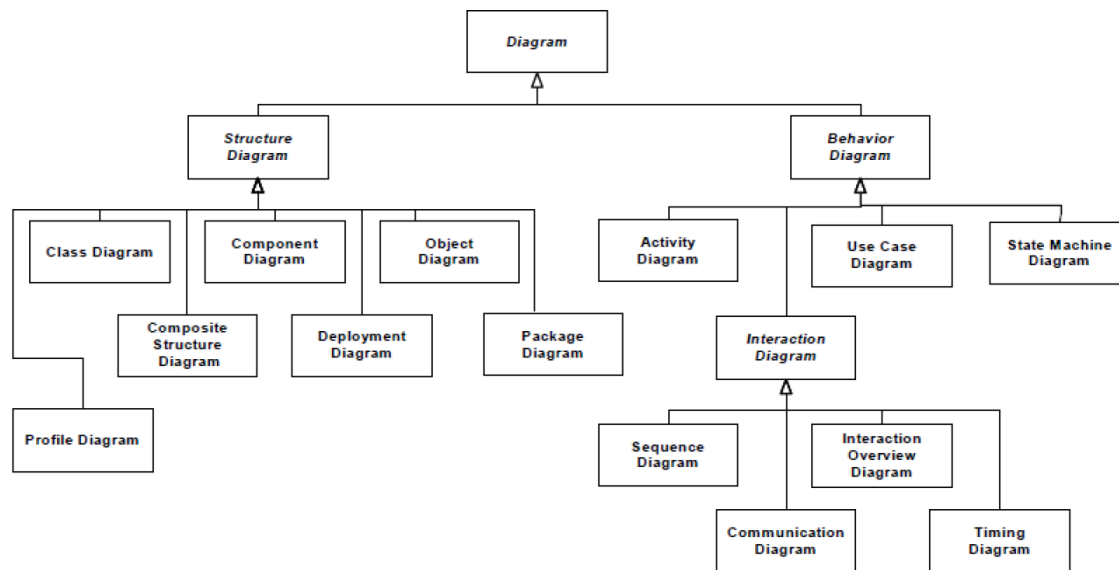


Figure 39: The taxonomy of UML diagrams [61, p. 685]

UML models are primarily meant for human-to-human communication and can therefore arbitrarily vary in precision and ambiguity without rendering them incorrect. In particular, there are only few rules what kinds of model elements are really mandatory in which UML diagrams. This supports an iterative modeling approach, where models are continuously detailed out as the understanding of the system in question grows.

In the following subsections, we exemplify the most common UML diagrams.

16.5.1 UML Use Case Diagram

UML use case diagrams model functional requirements of a system on a coarse-grained level. The main model elements in a UML use case diagram are actors (human users or other systems), subjects (systems or parts of them) and use cases that stand for some kind of behavior of the subject, which are possible sequences of event occurrences, i.e., sequences of inputs, outputs, and state changes of that system.

A UML use case diagram is a good starting point for identifying work packages for the detailed design and particularly for the implementation of a system.

EXAMPLE 52: EXTENDS AND INCLUDES IN UML USE CASE DIAGRAMS

The <<extends>> relationship between use cases gives you a hint which use cases will lead to source code that is coupled to some degree and thus should be implemented by the same developer. An <<includes>> relationship hints to some kind of interface on which different developers need to agree on.

UML model diagrams deliberately do not provide a technical view of the system in question. In particular, they are agnostic of interfaces and components that constitute a system. Such aspects are subject of other sorts of UML diagrams. Use cases are usually labelled with a verb plus an object. The behavior of a certain use case is not modeled in a use case diagram. Instead, the behavior is modeled a refining UML diagram of another type like a UML state machine or activity diagram (cf. Sections 16.5.2f) or some kind of textual description.

EXAMPLE 53: ReWo-RATE UML Use Cases

The following example shows a simple UML use case diagram which exemplifies the most important model elements:

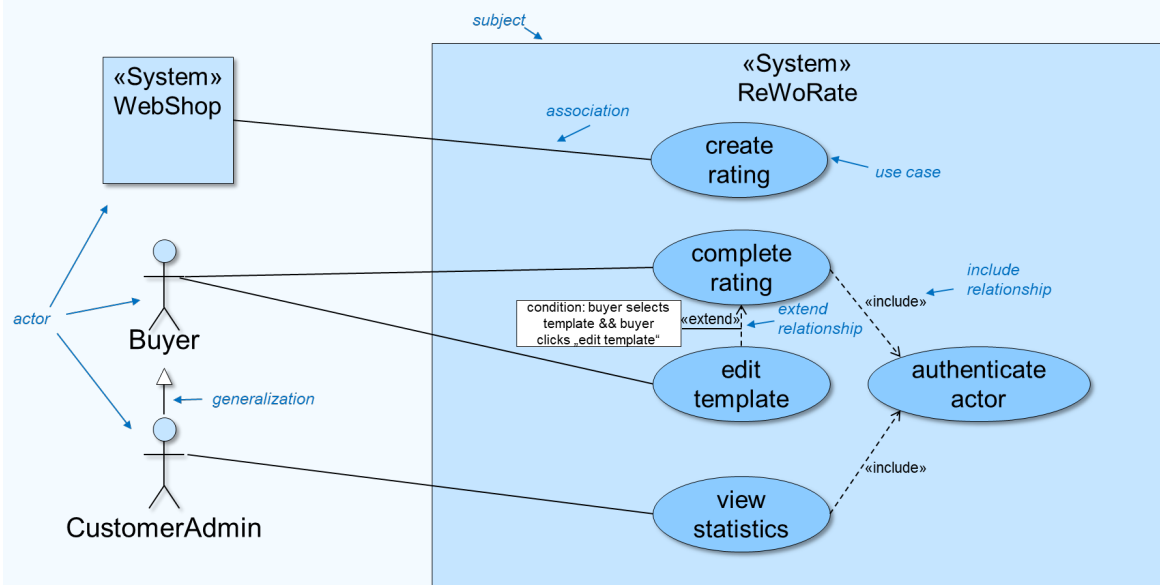


Figure 40: A UML use case diagram

The meanings of the model elements are described in the following Table 21:

Model element	Mapping / meaning
Subject	A subject can be anything that has behavior, i.e., that has (complex) state which changes due to events, and which emits events. This includes systems, subsystems, services, or even classes. Oftentimes, the subject is the whole system in question and stereotyped as such.
Actor	An actor represents a human user captioned by his or her role with respect to the subject. It is also possible that an actor is an external system that interacts with the modeled system.
Use case	A use case represents a function of a system that an external actor can interact with and that implements some kind of behavior, i.e., certain sequences of events and state changes. A use case is normally labelled with a verb plus an object like in the example. It is also admissible to summarize recurring patterns of use cases into one if you explain how. In particular, you can summarize use cases create X, read X, update X, and delete X as manage X or crud X.
Association	An association relates an actor with a use case. It particularly is used to model which actor is authorized to do what with the system. In the example above, ratings are created by external WebShop systems but completed by Buyer actors.
Generalization	A generalization between actors denotes that the more specific actor inherits all associations of the more generic actor. In the example, a CustomerAdmin can act as a Buyer and complete ratings on its own.
Extend relationship	An «extend» relationship means that certain events during the execution of a use case are optional and thus externalized into another use case. In the example above, the Buyer may use a template during completion of a rating and thereby open and edit this template.
Include relationship	An «include» relationship expresses that a certain use case must be executed during the execution of another one. In the example, the authentication of the actor must take place during the completion of the rating or the view of statistics. Of course, one could abstract from this sub step and model the authentication step in the refining description of the use cases complete rating and view statistics. When multiple use cases share common sub use cases, it is beneficial to model them as shown.

Table 21: UML use case diagram model elements

16.5.2 UML Activity Diagram

UML activity diagrams core element are actions and control flows. Actions are typically labeled with verbs like “review rating” that denote some kind of process—executed manually by a human or automatically as part of a program—instead of adjectives like “complete”. Current states and their transitions are not explicitly modeled in UML activity diagrams.

UML activity diagrams can be used to describe business processes as well as the inner workings of some (concurrent) algorithm. Therefore, it is again crucial to explain the actual meanings of the model elements and the meaning of the action model elements in particular.

EXAMPLE 54: UML ACTIVITIES IN ReWo-RATE

Figure 41 depicts an example of an UML activity diagram.

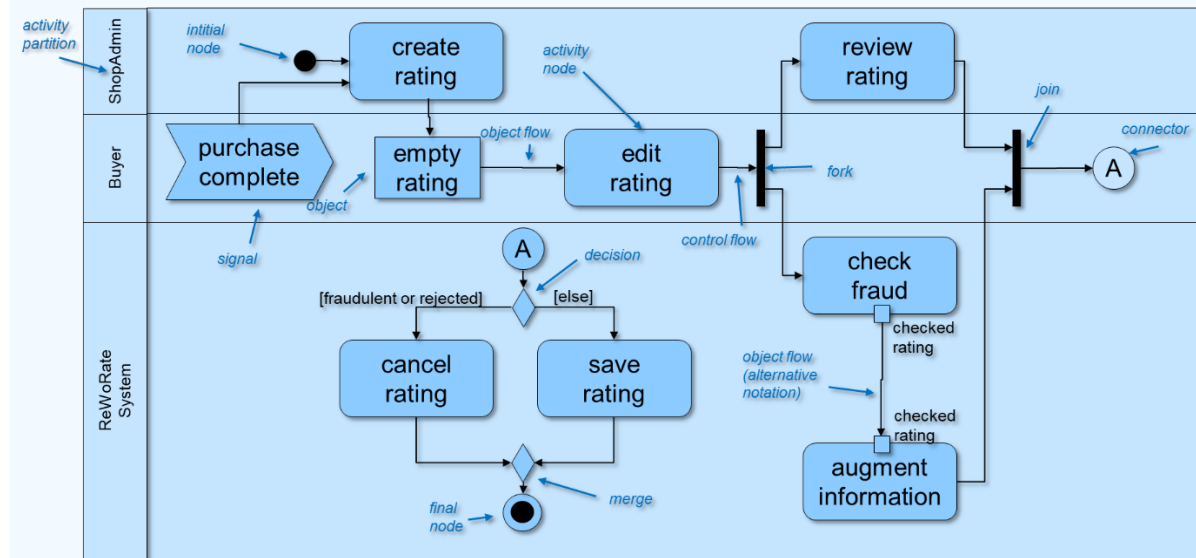


Figure 41: An UML activity diagram

The meanings of the model elements are described by Table 22.

Model element Meaning / Mapping

Activity node	An activity node encapsulates some kind of behavior, i.e., converts some input to output and/or changes the state of the overall system. In the example, the activity node “create rating” creates a new rating, so gets no input but produces an “empty rating” as output. The overall state change is therefore that there is a new rating, e.g., reflected in a new row in a database table of the ReWo-Rate system. Note, that “behavior” can be implemented in completely different ways, e.g., be some program invocation or a completely manual human task. If in doubt, one should use stereotypes to indicate what kind of activity we speak about.
Control flow	Control flows connect activity nodes and thereby determine their execution order. An activity node can only be executed when all activity nodes at the source of incoming control flows are completed.
Object	An object is a passive entity that represents some kind of data that is produced by an output of some activity node and/or used as input of activity nodes. In the original system, the object might actually be passed by value as the body of a HTTP message, as a message in a message queue, as a parameter in an in-process function call or by reference as a record in a database (where actually just a pointer, e.g., a key is passed around), or a pointer in a function call. Moreover, if the activity nodes are executed by humans, the objects can also be emails or an actual paper document that is passed around etc.
Object flow	An object flow is a special kind of control flow that carries an object. The object is either notated between the activity nodes or on the output and input pins (parameters).
Signal	A signal is a special kind of object that is supposed to originate from outside the modeled part of the system or is sent outwards. So, as opposed to normal objects, signals are either just the input of activity nodes or just the output but not both. In the original system, a signal might be an operating system interrupt, an incoming or outgoing http message, email etc.
Initial node	The initial node is a pseudo activity node that marks the beginning of the overall execution.

Model element	Meaning / Mapping
Final node	The final node is the counterpart for the initial node and marks the end of the overall execution.
Fork / join	A fork is a pseudo activity node that has no behavior on its own. It has a single incoming and two or more outgoing control flows and thus spawns a parallel execution. A join node is the counterpart that has two or more incoming and just one outgoing control flow. Mixtures with two or more incoming and outgoing control flows are also possible. Depending on the nature of activity nodes in an original system, parallel execution can be implemented by forking additional processes or threads or calling other (remote) processes which run independent of the caller by nature.
Decision / merge	A decision node is also a pseudo activity node and has one incoming and two or more outgoing control flows. The outgoing control flows are labelled with mutually exclusive conditions. Therefore, just one successor activity node becomes active while the others are left unexecuted. A merge node is the counterpart that collects all alternative paths. Within a program, a decision is typically implemented with constructs like if or switch.
Connector	A connector node is just a placeholder node with no semantics that can be used to enhance the layout of a diagram. A connector with some label X occurs exactly twice per diagram where one occurrence has one incoming and the other one outgoing control or object flow, which are to be considered one single control or object flow.
Partition	Activity nodes can be grouped in partitions, where each partition indicates some commonality (like stereotypes). Typically, a partition refers to the executor of the activity, e.g., a human (role) or some system. Partitions are often notated as so-called swim lanes.

Table 22: UML activity diagram model elements

16.5.3 UML State Machine Diagram

UML state machine diagrams model finite automata that contain states and the effects and conditions of state changes. Roughly speaking, UML state machine diagrams flip the meaning of boxes and lines in comparison to UML activity diagrams.

Again, the meaning and mapping of a UML state machine diagram to the system in question is often non-obvious and needs additional explanation. In principle, the current state of a system or the state of system parts can be reflected anywhere in accessible memory and even storage. However, it is good programming practice to encapsulate an automaton in, e.g., a class in an object-oriented language, where the current state is stored in an instance member variable. This way, a UML state machine diagram can be used to detail the behavior of some class in a UML class diagram.

EXAMPLE 55: UML STATE MACHINE FOR RATINGS IN ReWo-RATE

In the example of Figure 42, we see the UML state machine for the state of a single rating ReWo-Rate. Note that each of the current ratings in ReWo-Rate would have a distinct state. This is common as the overall system state is often too complex to be modeled and not of any interest, too.

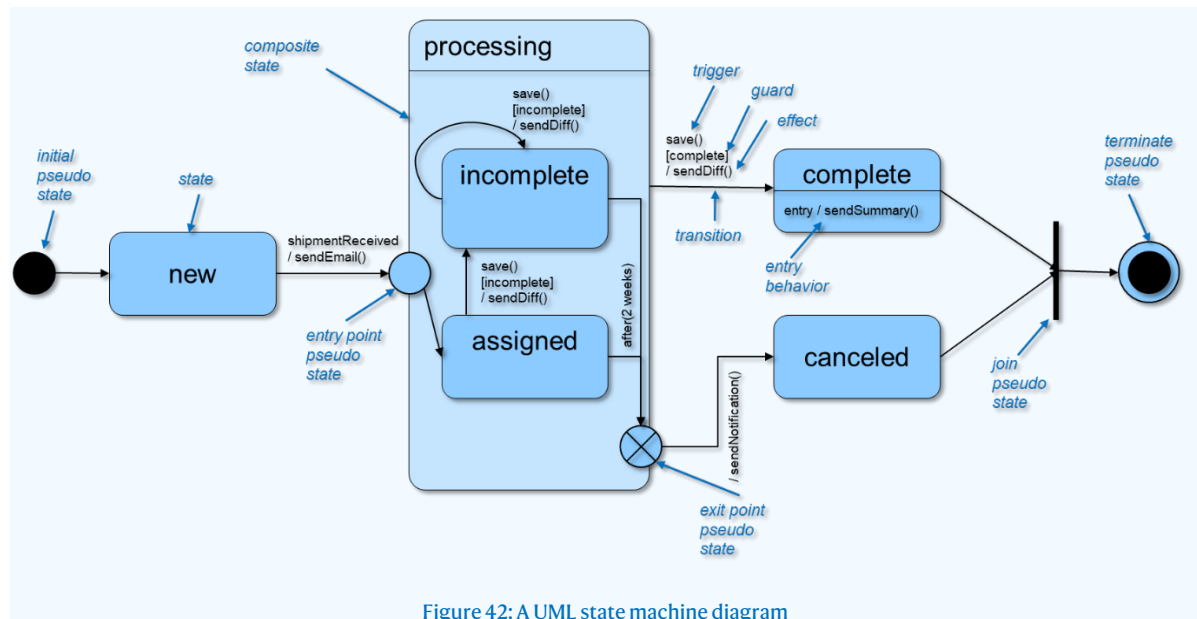


Figure 42: A UML state machine diagram

Figure 42 includes the most common model elements of UML state machine diagrams, which are further described in

Model element	Meaning / Mapping
State	A state of a (part of a) system. In the example, the state would be reflected as being the value of a certain member variable state of a Java class Rating. Note that the name of a state is supposed to sound like a state, like “closed” or “processing” but not like an activity like “order product”.
Entry behavior	The entry behavior is something that happens when a state is entered. What that something is, again depends on the nature of the modeled original. If we talk about the states of an object in an object-oriented language, this “something” usually is the invocation of a function, sendSummary() in the example.
Initial pseudo state	The initial pseudo state points to the very first actual state that the stateful original can be in.
Terminate pseudo state	The terminate pseudo state marks the end of all state transitions. If reached, the modeled stateful original sometimes also ceases to exist, e.g., a Java object instance might then be released to garbage collection.
Composite state	A composite state is a named state that contains other (composite) states. Its internals may be shown or hidden (e.g., be abstracted) in different UML state machine diagrams.
Entry point pseudo state	A composite state may have a dedicated entry point that bundles all incoming transitions.
Exit point pseudo state	Same for outgoing transitions.
Join pseudo state	Again, a bundling pseudo state, which however can be used independent of composite states.
Transition	A transition marks a possible change between two states. Conversely, the absence of a transition marks an impossibility. In our example, the state of Rating.state cannot change directly from new to completed.
Trigger	Transitions may be captioned by triggers. If that trigger, e.g., an event, occurs and the stateful object is in the source of the respective transition, the state may change to the

Model element	Meaning / Mapping
	target of the transition. What a trigger is, depends on the nature of the original. In the case of the example, triggers are function calls.
Guard	A guard is a condition that needs to be evaluated before a transition can be traversed.
Effect	Traversing a transition reflects a state change but can additionally cause effects (events) to happen. Effects are often the desired outcome, whereas the state are rather bookkeeping. Like with triggers, it depends on the system what an effect can be. In our case, it is again a function call.

Table 23: UML state machine diagram model elements

16.5.4 UML Component Diagram

The UML component diagram can be used for decomposition of the static structure of a certain system.

UML components are generic elements³⁹, which can stand for everything in a system that has an implementation and provides interfaces for or requires interfaces from other components. Components have no natural counterpart in the modeled system. Therefore, it is crucial to clarify how the original of a component actually looks like. Usually, there are several components that are all of the same kind and thus all represent the same concept in the modeled system. One can tag these with the same stereotype in such cases and define their meaning just once for the stereotype. Table 24 gives some examples for common component stereotypes as well as other model elements in UML component diagrams and their respective meaning.

³⁹ Due to the UML meta model, UML components are actually direct specializations of UML classes as defined in [61, p. 209]. So, every UML component is also a UML class. The difference between both according to the UML specification is that UML components do not need to be instantiated, can have required and provided interfaces and realizations.

EXAMPLE 56: UML COMPONENTS FOR ReWo-RATE

Figure 43 shows an example with all model elements.

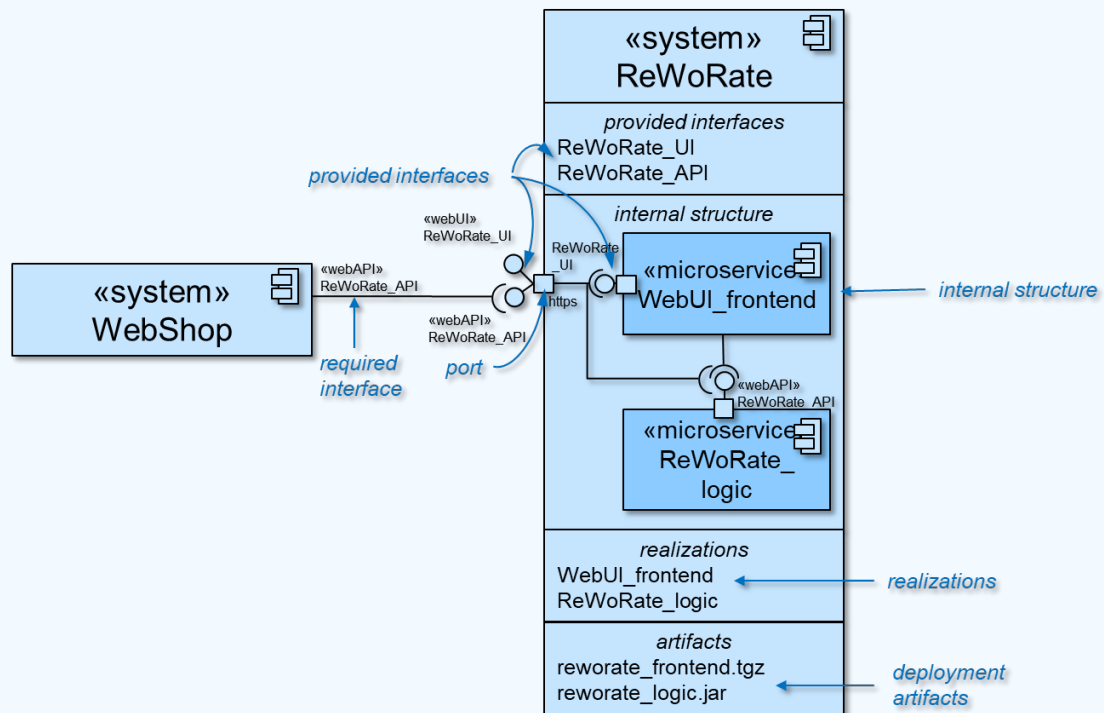


Figure 43: A UML component diagram

This example component diagram is on the same level of decomposition as the UML use case diagram in Figure 40 but shows other aspects like interfaces while leaving out others like external human actors.

Table 24 describes the meaning of these model elements.

Model element	Meaning / Mapping
«system» component	A «system» component represents a whole system; may it be the system in question or one of its partner systems. Note that the component with stereotype „system» need not have a direct counterpart in the modeled original (system). It may just be a conceptual aggregate of its internal structure.
«microservice» component «service» component	A «microservice» or «service» component represents a microservice or service, respectively. One should define what a microservice / service actually means in the model. In Sections 2.3.8 and 20.4.2, we give a common definition for these terms which can serve as a starting point for a more context-specific definition.
«FPGA» component	In microcontroller programming, a single FPGA on a board could be a component.
«program» component	«program» component represents a whole program, e.g., code with an entry point (main-Method) that can be started and run in a process
«JavaModule» component	«JavaModule» component represents a Java module in the sense of [62, p. 225]
«SpringComponent» component	A «SpringComponent» component could be defined as a pattern within a Java program that uses the Spring framework, and which consists of two Java classes and an Interface that implements a Spring Controller, a Spring Service and a Spring Repository. Developers are supposed to be familiar with this pattern, so it

Model element	Meaning / Mapping
	is reasonable to abstract it into a single model element.
«NextJSPage» component	A «NextJSPage» component represents a JavaScript or TypeScript function in a NextJS-based program that returns a ReactElement, which renders a whole route (page) in a single page web application.
«ReactElement» component	A «ReactElement» component represents a JavaScript or TypeScript function that returns a ReactElement JavaScript object, which renders a dynamic visual element within a web page.
Provided interface	A provided interface is an interface that can be accessed by other components to interact with the exposing component. The nature of a provided interface depends on the component. A provided interface of a «microservice» component can typically be remotely accessed via HTTP. A provided interface of a «ReactElement» component is just the signature of the respective JavaScript function and only be called from within the same program. A provided interface can either be noted as a lollipop or listed in the provided interfaces compartment of the component.
Required interface	Analogous to provided interface, yet a required interface R of component X means that X depends on another component Y which has I as required interface. A required interface can either be noted as a grab or listed in the required interfaces compartment of the component.
Port	The meaning of a port also depends on the nature of the component. For «system», «(micro)service», or «program» components, ports are typically TCP or UDP ports through which HTTP-based APIs are served (cf. Section 27.1). For a «FPGA» component, the ports are hardware ports that adhere to a certain standard. For, e.g., a «SpringComponent» component, ports make no sense at all.
Internal structure	Contains the next decomposition level. If that level is too complex, it should be modeled in a separate diagram.
Realization	A textual list of the components on the next decomposition level.
Deployment artifacts	The names of the artifacts (usually files) that manifest the respective component, e.g., contain program code, configuration file and the like. This relationship can also be modeled in a UML deployment diagram (cf. Section 16.5.6)

Table 24: UML component diagram elements

16.5.5 UML Class Diagram

UML class diagrams are particularly useful to model the structure of programs or parts of them. Other uses are to express data schemas, e.g., in a relational database, although not all kinds of model elements have a meaning in such contexts.

EXAMPLE 57: UML CLASS DIAGRAM FOR ReWo-RATE CONCEPTS

The following example shows the most common model elements of a UML class diagram.

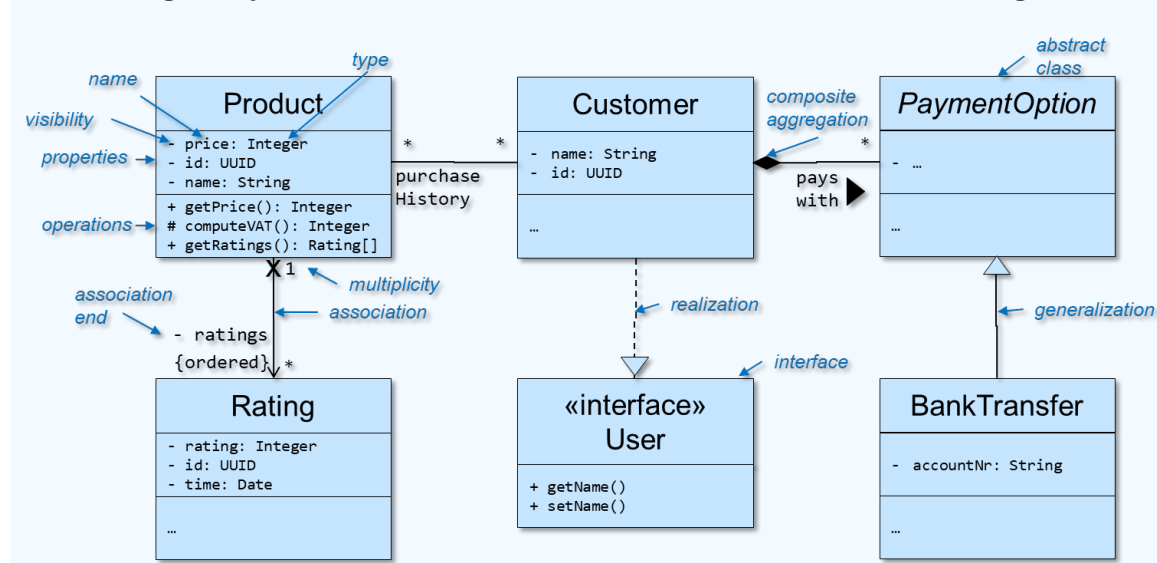


Figure 44: A UML class diagram

UML class diagrams are particularly geared towards programs written in an object-oriented programming language like Java, C# and the like. Here, the mapping of a UML class to program units in the respective program is rather obvious since these programming languages have a concept of a class for themselves. In Table 25, we describe the model elements used in Figure 44. We assume that the modeled original is a Java program; so, we describe the mappings too. Meanings of UML class diagram model elements would need a different mapping for other programming languages. In Rust for example, a UML class might stand for a trait together with a `struct` and `impl`.

Model element	Meaning	Mapping to Java
Class	A UML class stands for a program unit (or a set thereof) that typically encapsulates state (i.e., mutable data) via properties and behavior via operations, which might retrieve or mutate that data. UML classes are meant to be instantiated at program runtime into objects. A UML class has a name like <code>Product</code> , which denotes the type of instantiated objects.	If the modeled program is written in Java, a UML class will be mapped to a Java class with the respective name.
Properties	Properties are named variables with types that are either fundamentally built-in the programming languages like <code>char</code> or refer to other classes. Properties have visibilities which make them accessible from code within other classes or not.	In object-oriented languages properties are typically mapped to instance member variables, so each instance of the respective class has the same variables but with different values.
Operations	Operations are declarations of functions. Operations are meant to access properties, i.e., read and mutate them or compute a return value (output) from input parameters.	Operations are typically mapped to Java class methods.
Associations	If a class <code>Product</code> has a property <code>ratings</code> , whose type is or includes <code>Rating</code> (like the array type <code>Rating[]</code>), one can use an association between	Associations together with their association ends are again implemented as

Model element	Meaning	Mapping to Java
	both classes. The arrowhead at the association end indicates that <code>Rating</code> instances are navigable from <code>Product</code> instances (via <code>ratings</code>); the cross on the opposite side expresses non-navigability.	instance member variables in Java. In the example, we would have a declaration like <code>private Rating[] ratings;</code> or <code>private List<Rating> ratings;</code> In the Java code. A <code>Set<Rating></code> would not suffice due to the <code>{ordered}</code> modifier. Due to the non-navigability, there is no counterpart in the <code>Rating</code> Java class like an instance member variable declaration <code>private Product product;</code>
Association end	The association end at the <code>Rating</code> side is captioned with <code>rating</code> to indicate that there is a property with a type that is or includes <code>Rating</code> . The modifier <code>{ordered}</code> indicates that the ratings are ordered.	
Multiplicities	Multiplicities denote how many instances refer to how many instances of the associated class.	The mapping of multiplicities 1 and * are explained above.
Composite aggregation	A rhomb on an association end expresses a whole-part-relationship. In our example, every <code>PaymentOption</code> instance may exist if the respective <code>Customer</code> instance exists.	Developers must take care of the composite aggregation inasmuch a part cannot exist without its whole. For example, if the Java classes <code>Customer</code> and <code>PaymentOption</code> are backed by Tables in a relational database, the deletion of a customer must cascade to the deletion its “owned” payment options as well.
Abstract class	Declaring a UML class as abstract by setting its name in italics means that it cannot be instantiated.	Java provides a keyword <code>abstract</code> for that.
Interface	Stereotyping a UML class as «Interface» means that the class itself has not properties but only operations and cannot be instantiated itself.	Java provides a keyword <code>interface</code> (instead of <code>class</code>) for that.
Realization	A class can act as the realization of an interface if it implements a superset of the interface operations.	Java provides the keyword <code>implements</code> for that.
Generalization	A class can inherit from another class and thus get all its properties and operations.	Java provides the keyword <code>extends</code> for that.

Table 25: UML class diagram elements

16.5.6 UML Deployment Diagram

A UML deployment diagram bridges the gap between rather conceptual UML components and where these components actually “run” in the system.

EXAMPLE 58: UML DEPLOYMENT DIAGRAM FOR ReWo-RATE

Figure 45 exemplifies a UML deployment diagram with the most common model elements.

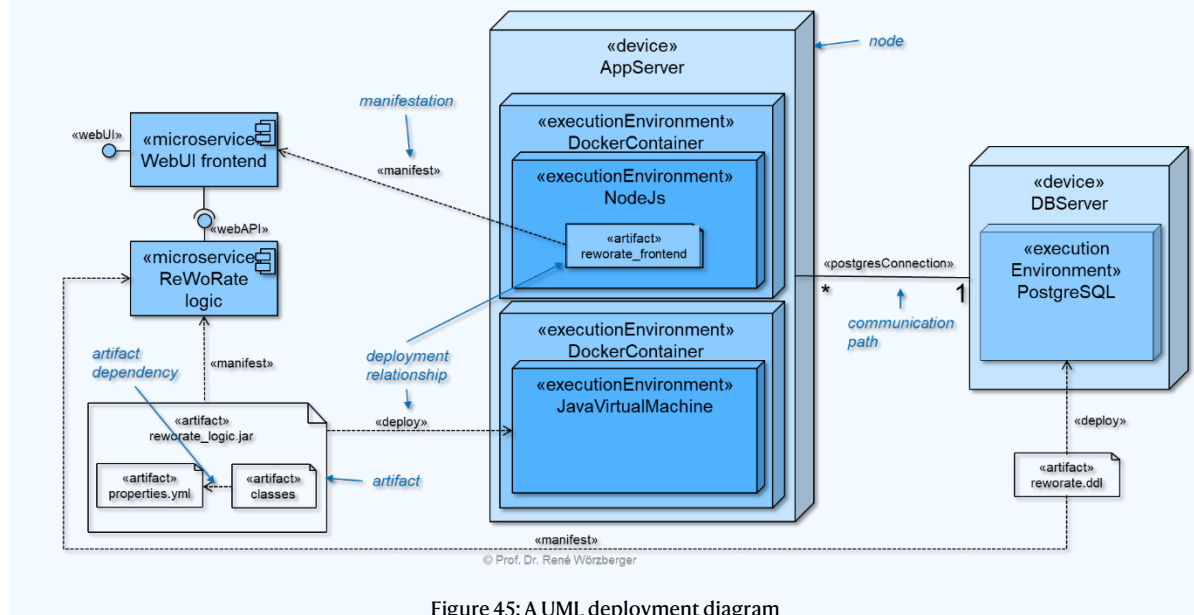


Figure 45: A UML deployment diagram

The meanings of these common model elements are described in Table 26.

Model element	Meaning / Mapping
«device» node	A «device» node is a physical computing resource, e.g., a machine in the sense of Section 2.3.4 or FPGA board etc. Virtual machines as described in Section 25.1.3 often also count as «device» nodes.
«executionEnvironment» node	«executionEnvironment» node can be nested into other nodes and stands for an execution environment for programs. Operating systems, containers or the Java Virtual Machine are examples for such «executionEnvironment» nodes.
Artifact	An artifact is something deployable like a file or a directory. Artifacts may be single executable files, like the output of a Go or Rust compiler, single archive file like a Java Archive (*.jar) file, or a whole directory, whose files are to be executed or served by a NodeJS server.
Manifestation	An artifact as a physical entity may embody (manifest) a rather conceptual UML component. In the example of Figure 45, the ReWo-Rate_logic.jar manifests the UML component «microservice» ReWo-Rate_logic. The ReWo-Rate_logic.jar is further refined into contained subartifact classes, which contain the bytecode of the program and properties.yml, which contains parameters to configure environment dependent values like network ports. The file ReWo-Rate.ddl also manifests this component as it contains SQL code that builds the database schema for the «microservice» ReWo-Rate_logic.
Deployment	Artifacts need to be deployed to nodes to become effective in a system. What deployment actually means in the modeled system depends on the artifact and might need further explanation. In Figure 45, the deployment of ReWo-Rate_frontend means that this artifact is a directory which contains some files (normally in a “build” subdirectory) that are served as-is to a browser, where they are interpreted and some files, which are interpreted in the NodeJs execution environment itself. The ReWo-Rate_logic.jar means that the file is copied into a file system mount that is accessible from the DockerContainer and then passed as parameter during start of the JavaVirtualMachine. The deployment of the ReWo-Rate.ddl means that this

Model element	Meaning / Mapping
---------------	-------------------

	SQL file is executed by a database client node, e.g., psql, against the PostgreSQL node in order to create the necessary database objects in the database.
--	--

	Deployment relationships can either be expressed by a dependency (arrow) with stereotype «manifest» or by nesting the artifact into the respective node.
--	--

Communication path	A communication path denotes a data connection between nodes. It abstracts from any intermediaries on that path like routers, switches, or load balancers. Its nature can be a network connection or inter-process communication between «executionEnvironment» nodes that are in the same operating systems. Since a node in UML is a subtype of class it can be instantiated multiple times. This can be used to abstract from scaling out nodes (cf. Chapter 18) in the deployment diagram. How many nodes communicate with nodes on the other side of the communication path is then expressed by multiplicities. (CommunicationPath is a specialization of Association in the UML meta model.) Of course, one may refine a communication path with a stereotype. In the example, «postgresConnection» indicates that we speak about a TCP connection on standard port for the Postgres protocol (normally over TCP port 5432).
--------------------	---

Table 26: UML deployment diagram elements

16.5.7 UML Sequence Diagram

A UML sequence diagram models possible executions between interacting systems or parts within a system. It models the chronology of message events within a system (of system) if read top to bottom. In particular, one can model traces of messages that are exchanged between system / parts in. UML sequence diagrams typically just exemplify such traces instead of specifying the set of all possible traces although so-called combined fragments can be used to model, e.g., alternative traces.

EXAMPLE 59: UML SEQUENCE DIAGRAM FOR ReWo-RATE

Figure 46 exemplifies a UML sequence diagram. Here, we see the inner interactions of class instances (objects) within the program that implements component ReWo-Rate_logic (cf. Figure 43) written in some object-oriented programming language.

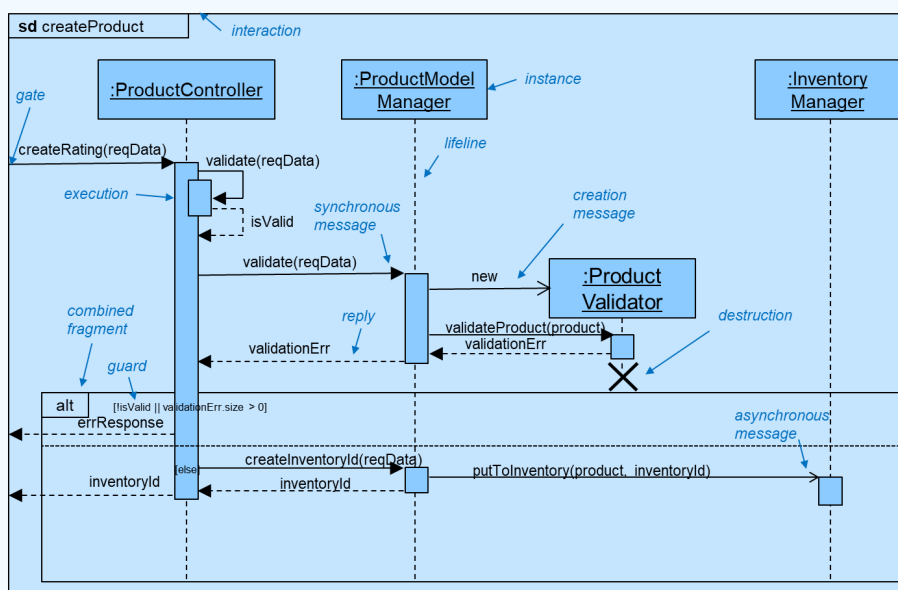


Figure 46: A UML sequence diagram

In Table 27, we describe the model elements of the UML sequence diagram in Figure 46.

Model element	Meaning / Mapping
Interaction	An interaction is the enclosing element of a UML sequence diagram.
Lifeline	A lifeline represents the instance of a class or a component. It can be the source or sink of a message.
Execution	An execution marks an instance to be active. What “active” means depends on the kind of the instance. If the instance is of a «microservice» or «program» component, “active” would mean that the respective thread is an active state. If the instance is that of a class, “active” means that a method of that class is part of a current call stack.
Message	A message denotes a data flow between lifelines. Depending on the instances of the lifeline, a message could be an in-process call between program units (e.g., functions) or represent a message over a network between different, e.g., microservices.
Synchronous message	A synchronous message (cf. Section 21.4.1) means that the execution of the sender’s lifeline is blocked until the reply message arrives. In-process calls are usually synchronous. A synchronous message has a triangular arrowhead
Reply	A reply comes after a synchronous message and is in opposite direction. In an in-process call it would model the return statement of a function. It is noted with a dashed arrow and a triangular arrowhead.
Asynchronous message	An asynchronous message is non-blocking for the sender (cf. Section 21.4.2). It is noted down with an open arrowhead.
Gate	A gate represents messages that enter or leave the current interaction. Like for interface is UML components diagram, a modeler must keep these messages consistent across different decomposition levels.
Creation message	A message can particularly lead to the creation of a new instance. For example, a Java object instance can have a method with a new statement that creates another object instance.
Destruction	Instances can be destroyed as well. In programming languages there are explicit destructions using keywords like <code>delete</code> or implicit destructions when instances become unreachable.
Combined fragment	A combined fragment represents a control flow structure like an alternative (keyword <code>alt</code>) or a loop (keyword <code>loop</code>). [55, p. 582] provides a list of all possible kinds of combined fragments.
Guard	A combined fragment may be augmented with a guard that reflects the condition that must hold such that the first alternative is pursued, or the loop keeps iterating, respectively.

Table 27: UML sequence diagram elements

16.6 The C4 Model

The UML lost adoption in industry in recent years. This may partly be attributed to the (mis)conception that rigor and exhaustive modeling itself yields not enough (short-term) value in agile development. However, UML as a modeling language itself has some weak spots:

- The UML 2.5.1 specification [61] comprises no less than 754 pages due to the vast number of different model elements with their respective inner syntactical structure and their interconnections. This makes it very hard to really master UML.
- The UML does not specify what to model at each decomposition level. (The decomposition levels in the examples in Section 16.3 are not part of the UML standard.)
- The meaning of model elements like “component” or “node” remains very unclear and requires additional explanation as mentioned in Section 16.5.4.
- The UML is unmistakably a child of the late 1990’s when object-orientation was very en vogue. Conversely, it partly lacks support for modeling large and cloud-based software systems. For example, horizontal scaling of a certain node can only be modeled indirectly via multiplicities on adjacent communication paths.

The C4 model as presented in [63] addresses these issues. The only model elements in C4 are **persons**, **software systems**, **containers** and **components** and arbitrary **relationships** between them which are defined by examples in [63].

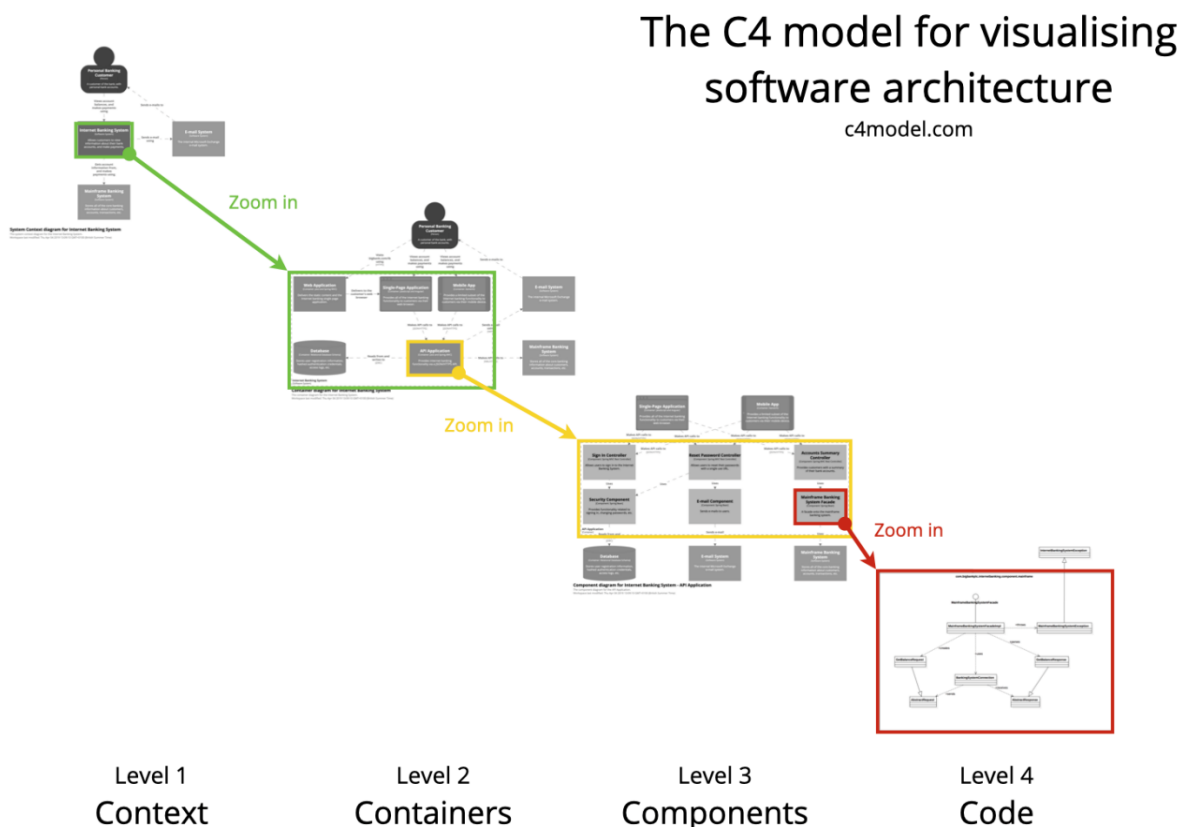


Figure 47: C4 decomposition levels [57]

The decomposition levels of C4 are depicted in Figure 47.

1. **Context diagram:** The context diagram just contains **software systems** and **persons**.
2. **Container diagram:** A software system is decomposed into containers. A **container** is independently deployable and thus roughly comparable to what we have defined as a tier in Section 18.2.
3. **Component diagram:** A container is further decomposed into **components**, which represent parts specific program.

4. **Code diagram**: A component can be further decomposed into its constituent program units, e.g., into classes.

Besides these diagrams, C4 defines

- a **system landscape diagram**, which is like on the decomposition level of a context diagram but without the focus on a specific system,
- a **dynamic diagram** that resembles the UML collaboration diagram, and
- a **deployment diagram**, which corresponds to its UML counterpart.

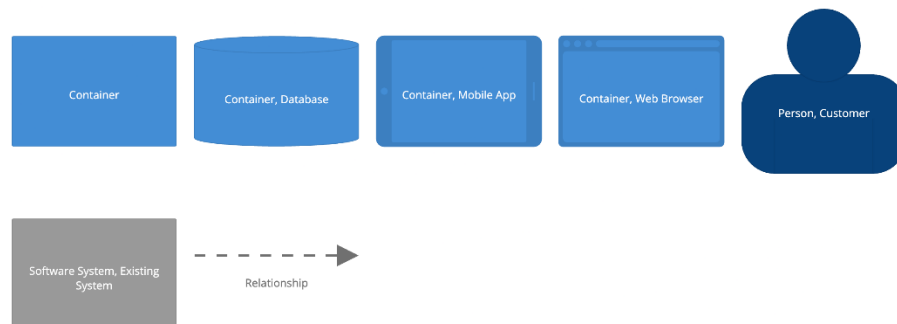


Figure 48: C4's box-and-lines notation [63]

C4 claims to be notation independent and is thus expressible in UML notation. However, it suggests a box-and-line notation on its own. In contrast to UML, the boxes contain more and non-formal text including (1) a name, (2) some description of the kind of technology the model element represents, e.g., a Spring Controller or a Docker Container in case of a component and container, respectively, and (3) a description of the responsibility of the respective model element. Some few recurring subtypes of containers like databases or web browsers are depicted in a visual variant. There is also just one dashed arrow for expressing arbitrary relationships. The respective meaning of each relationship is given as the arrows label.

17 Software Architecture

After we discussed fundamental principles and patterns of design in the previous chapters, we now talk about how to express design in a software system. For now, we focus on what can be expressed in the source code of a single program alone. We call the part of the system architecture which is embodied in the source code of a single program **software architecture**.

A plethora of volumes have been written about software architecture on as patterns on the levels we identified in Section 15.1. In this section, we just summarize the most architectures on the level of architectural styles.

17.1 Modularization in General

The process of structuring a program in the large is called decomposition or modularization. There are several patterns (or strategies) by which one can break down (or: modularize) a program into several modules.

Please note that modularization is different from distribution. Each part that results from a modularization, i.e., each "module" (a.k.a. component) might technically be a program unit like a (set of) packages or namespaces or whatever the respective programming languages offers for modularization or even a program on its own if the modules are distributed. It is crucial that all developers of a system agree on how a module / component is embodied in the actual source code and its file and folder structure.

Also note that we have already seen decomposition as a modeling technique in Section 16.3. Whether we do that in a model or not has no compulsory effect on the original, i.e., the program. In the following we talk about actually using programming language constructs to do modularization. (Although decomposition in the model and the original are separate things, it is of course advisable to align them, i.e., to embody actual decomposition decisions in the model in the source code.)

17.1.1 Modularize by Technical Responsibilities

A system can be broken down by technical responsibilities. For example, it is not uncommon in Java programs to put classes that care for rendering the UI into a package, all classes that care for the business logic into another one and all classes that implement persistence into a third one. Layered Architectures (Section 17.2) and Hexagonal Architectures (Section 17.3) are architectures that modularize by technical aspects.

17.1.2 Modularize by Functionality

Another approach is to modularize a program by cohesive functionality. Of course, the question remains, what cohesive means regarding functionality. Roughly speaking, we mean those functions of a program that support the same use case, business capability, business role, or domain concept. **Domain-Driven design** [14] helps to design a system in such a way. Microkernel Architectures (Section 17.5) also modularize by functionality.

EXAMPLE 60: MODULARIZE A PROGRAM BY FUNCTIONALITY

If you have attended the "Software-Praktikum" in "Technische Informatik / Informatik und Systems-Engineering (Bachelor)" you might remember the WaWi's component architecture of Figure 49:

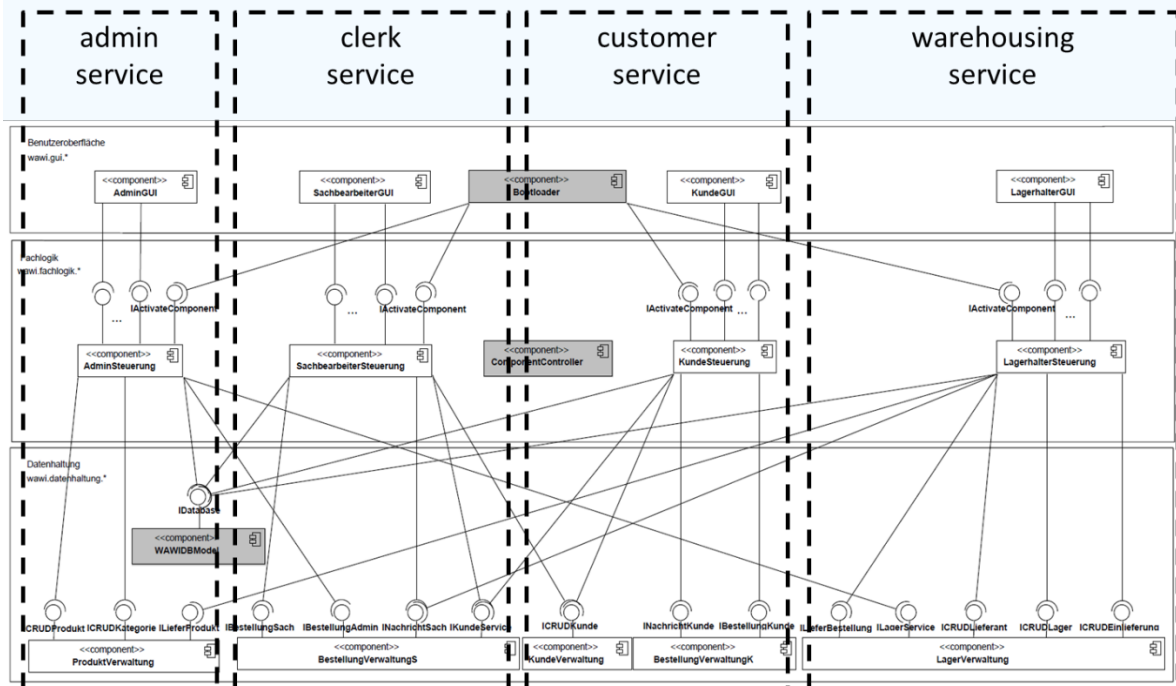


Figure 49: Component architecture of a "Warenwirtschaftssystem" as a UML component diagram (source: Software-Praktikum, Grobspezifikation WaWi)

It is better to modularize (and optionally distribute) this WaWi by functionality, e.g., having one „admin service“, one "clerk service", one for interaction with customers, and one for the warehousing would be better than to do it horizontally, i.e. having one module for GUI rendering, one for the business logic and one for persistence. Please note that the figure just models a program that typically also accesses a database to form a whole system. The database schema within the database is supposed to be modularized in the same way as well.

17.2 Layered Architectures

A layered architecture [58, Ch. 10] is an instance of “modularization by technical aspects”: Larger program units like classes in an object-oriented language are assigned to layers according to their responsibility in the system. Typically, there are the following layers: (1) a presentation layer, which receives input or requests from input devices or clients respectively and converts output to output devices or responses for clients, respectively. The presentation layer depends on (2) a (business) logic layer whose classes implement the logic and the define the data schema of a system and (3) a persistence layer, which translates between the business logic layer and a database or other kind of storage. In essence, the point of such architecture is to keep the business logic agnostic of aspects like storage or presentation. These concerns are separated.

In such a layered architecture the upper layers just call lower layers but not vice versa. The presentation layer sometime serves different flavors for different clients. For example, a UI package within the presentation layer might serve websites (HTML etc.) for browsers and an API package might serve JSONs for smartphone clients. Please note that in that case, we typically must add a client (browser) and a database to the overall system, which have been left out in Figure 50.

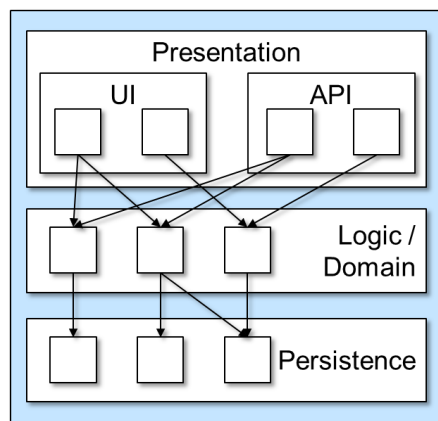


Figure 50: Layered architecture

17.3 Hexagonal Architectures

Layered architectures bear the downside that the logic layer (sometimes: the domain or model) has explicit dependencies to a specific implementation of the layer beneath it. Even if the interface of the persistence layer remains unchanged, substituting the persistence layer thus amounts to numerous changes in the logic layer due to changes of imported namespace and the like. This becomes worse if the logic layer depends on more than just the persistence layer, e.g., a layer that implements clients for communication with external systems.

In a **hexagonal architecture**, there is an outer ring that consists of **ports** incoming dependencies and **adapters** for outgoing dependencies. The inner core is comparable to the logic layer but mostly called domain. Dependencies may point from the ports and adapters inwards to domain but not out of it. More precisely, the domain defines classes and interfaces on which the outer ring depends on. Since the domain still call the adapters but the dependencies are now the other way around, hexagonal architectures make use of a pattern called **dependency injection**. The name of hexagonal architectures style stems from its typical depiction using hexagons.

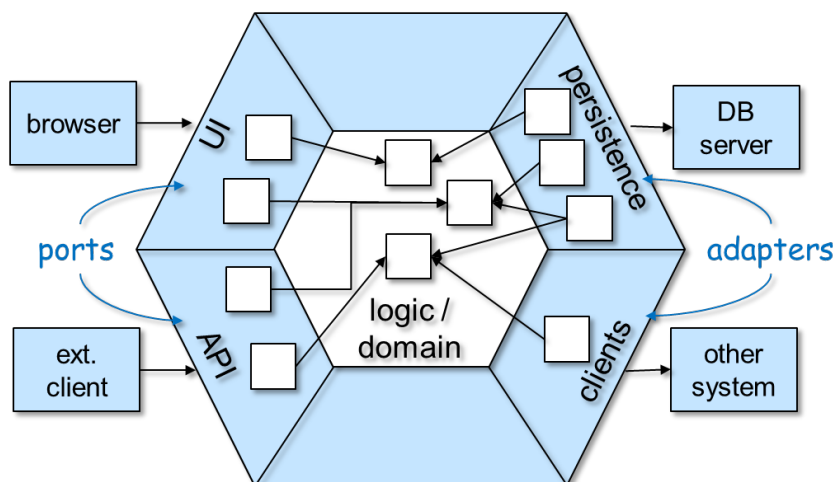


Figure 51: Hexagonal architecture

17.4 Pipes-and-Filters Architectures

Pipes-and-Filters architectures are particularly suitable for processing streams as workload (cf. Section 5.1). Different program units called **filters** produce, manipulate, augment, aggregate or consume such streams. They communicate with so-called **pipes**.

This kind of architecture makes sense only if the filters are executed concurrently or even in parallel. In a single program one can make use of threads or coroutines for filters and data structures like queues for pipes.

Pipes-and-Filters architectures are not restricted to single programs: Unix userland command line tools can be assembled to a more complex application using this architecture. In this case, the command line shell will spawn a subprocess per filter (except for producer and consumer, which are files) and uses inter-process communication for pipes.

The Pipes-and-Filters architecture style can also be used for a distributed system. Then filters are the distributed subsystems and pipes are realized with messaging systems.

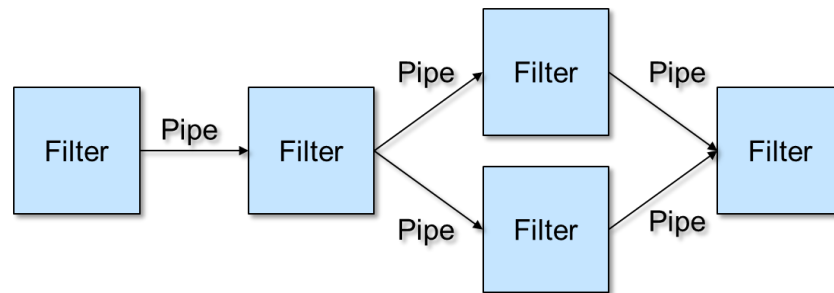


Figure 52: Pipes and filters

17.5 Microkernel Architectures

Microkernel architectures modularize by functionality. However, each module is not statically linked at build time but dynamically loaded at runtime and on demand. These kinds of modules are also called **plugins**. This dynamic loading must be supported by the programming language used to make this work.

EXAMPLE 61: MICROKERNEL ARCHITECTURE VIA CLASS LOADING IN JAVA

In Java, classes can be dynamically loaded and registered like this

```
String className = new Scanner(System.in).nextLine();
String pluginUrl = new Scanner(System.in).nextLine();
Class classLoader = new URLClassLoader(plugin).loadClass(className);
Plugin plugin = (Plugin)classLoader.newInstance();
plugin.doSomethingPluginish();
```

Although this style still produces a single program, it is typically not deployed in a single deployment artifact, i.e., a jar file in the case of Java, but one for the microkernel and one for each of the plugins.

18 Scaling Systems

There are systems that are distributed to several nodes but are still expose the very same interface and behavior to external nodes as if they were not distributed. This is what we put under the term **scaled system**.

18.1 Horizontal Scaling

As described in Section 7.2, there is the concept of horizontal scaling: We can start the very same program multiple times and get several equal nodes⁴⁰. If these nodes expose some kind of network interface, they can act as servers. We can put a load balancer between clients and theses server nodes which exposes a single network address to the clients and distributes requests to the respective server nodes according to some strategy (cf. Chapter 19).

To achieve horizontal scaling, we allocate different machines to the server nodes such that they do not have compete for computing resources (CPU, memory etc.). More on that in Section 18.3.

18.2 Tiers

When we talk about scaling a system, we must mainly refer to the aforementioned horizontal scaling. Such scaling happens within tiers, which are conceptual groups of nodes of the same kind, like application servers, database management systems, messaging systems and the like. An exhaustive discussion can be found in Chapter 23.

However, why do we have more than one tier in a typical system cluster instead of just one? Do we need nodes of different kinds at all? This is rather due to technical necessities that a deliberate architectural choice as we describe in the following.

18.2.1 User agent separation

Sometimes, the user agent is separated from the actual application. For example, each web application is distributed by nature just by the fact that clients (browsers) and servers are separate processes and normally remote to each other.

18.2.2 Standalone middleware

On the server side, certain **middleware** like web servers, application servers and database servers typically are separate nodes by design, each yielding a tier. In theory, one could think of linking all middleware as library or framework components together with self-crafted application logic into one big single program. However, linking code of different languages into a single program bears its own complexities:

- **memory management** differences, particular between languages with and without automatic garbage collection,

⁴⁰ Bear in mind that it is a little simplistic to assume that starting a single program results in exactly one process: Once started, a process can spawn child processes. For example, web servers or database servers spawn separate worker processes in order to process concurrent requests and background housekeeping processes. For example, if you start a PostgreSQL database on a machine, you get a root process called “postmaster” that spawns a so called “backend” child process for every connected client and “auxiliary” child processes for starting auto vacuuming, several for reading and writing to the write ahead log, one that writes statistics etc. (Why this is better than just using multiple threads within a single process is not in scope of this document.) Other examples are startup scripts, which are interpreted by some kind of command line interpreter (like bash) and may create also a bunch of child processes, which load their own executables. As mentioned in Section 2.3.8), we consider such process trees to be a single node.

- **platform independence** might be sacrificed, e.g., if C code that is called from Java makes some calls that are specific to a certain operating system.
- **type mapping** between languages with subtle differences in their primitive types (string, floating point types etc.),
- **exception handling** may work differently between languages,
- **multithreading** along with safety and synchronization guarantees may differ, and
- **debugging** such programs is usually less supported by common debugging tools.

Conversely, (remote) inter-process communication of two nodes is independent of the programming language used for the programs running in the nodes. In a way, (remote) inter-process communication serves as a common denominator w.r.t. communication for all programming languages. This solves at least the problems with memory management and platform independence mentioned above.

EXAMPLE 62: IN-PROCESS TIERS

Databases systems come both ways. For example, it is common to use an in-process database like H2 or SQLite during development, which may even not persist data onto disk. For production, these are being replaced by a Postgres database tier which can be scaled to cope with bigger workloads.

Even further, different middleware is often not just separated in distinct programs, it also is usually packaged and deployed using separate container images (cf. Section 25.3.3c). In theory, different programs for different tiers could easily be put together in a single container image. However, that would lead to a combinatorial explosion of container images since there is a great variety of products to choose from in each tier. So, in practice each container image typically targets exactly one tier.

EXAMPLE 63: TIERS AND HORIZONTAL SCALING

Figure 53 depicts a typical combination of tiering and horizontal scaling: We have three tiers, each of which is horizontally scaled plus one for the user agent.

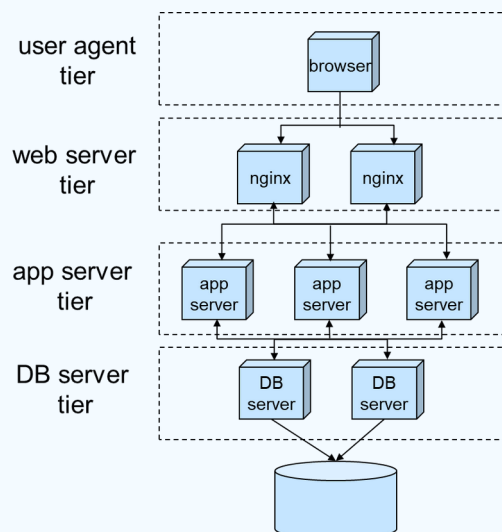


Figure 53: Tiers and horizontal scaling

18.3 Tiers versus Layers

Tiers and layers are often confused in literature, because both resemble each other in typical architectural depictions. Both have in common that they assume a certain technical responsibility, like converting in-memory data to HTML in HTTP responses (presentation layer).

However, and as described in Section 17.2, layers are a means to structure a single self-written program by technical aspects. Thus, layers contain (self-written) program units, e.g. classes in the case of an object-oriented programming language. In contrast, tiers are a way to arrange self-written programs with other middleware to form a whole system.

EXAMPLE 64: TIERS AND LAYERS OF A SERVER-SIDE-RENDERED TODO-LIST APPLICATION

Figure 54 expands the app server tier. It implements a completely server-side rendered ToDo list web application. Its code can be organized into three layers as described in Section 17.2.

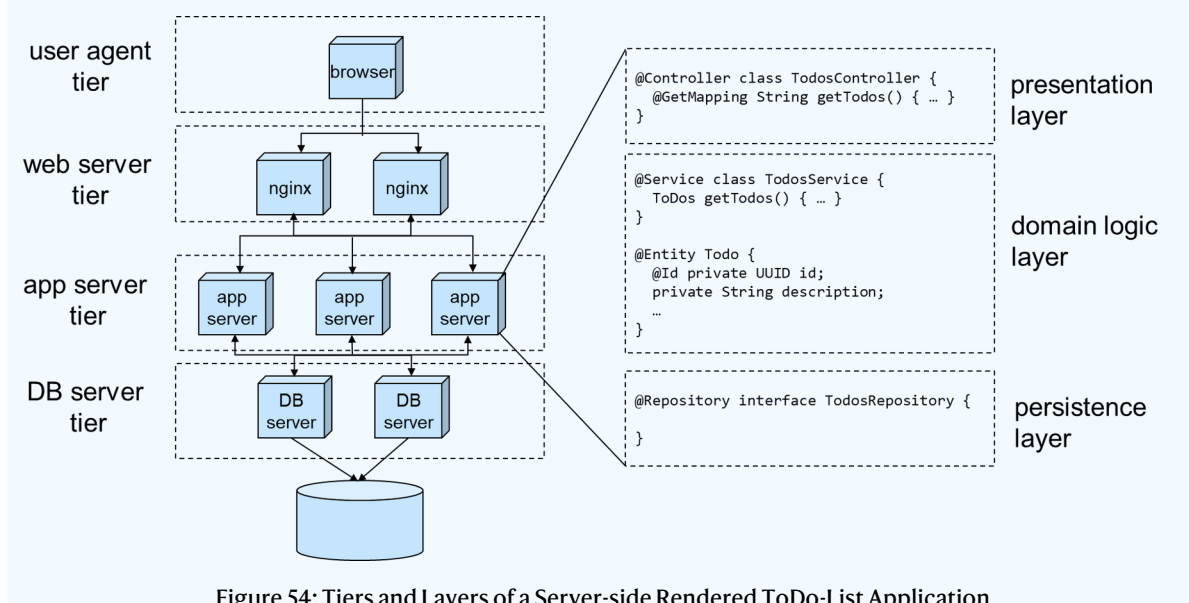


Figure 54: Tiers and Layers of a Server-side Rendered ToDo-List Application

18.4 Node-to-Machine Allocation

Figure 53 still simplifies things since the nodes run in “thin air”. The figure abstracts from allocations of actual machines to the nodes. However, such allocation and how it is done does matter: On the one hand, nodes on the very same machine compete for the machine’s resources. On the other hand, nodes on the same machine can communicate with high throughput and low latency as there is no network bottleneck between them. Furthermore, nodes of different tiers may have different demands for different resources, e.g., be rather CPU bound and thus require much CPU time, or memory-bound or disk I/O bound etc. and are thus better be allocated to machines with many CPU cores, memory, fast storage etc. respectively.

In the following, we describe patterns about how to allocate nodes to (virtual) machines.

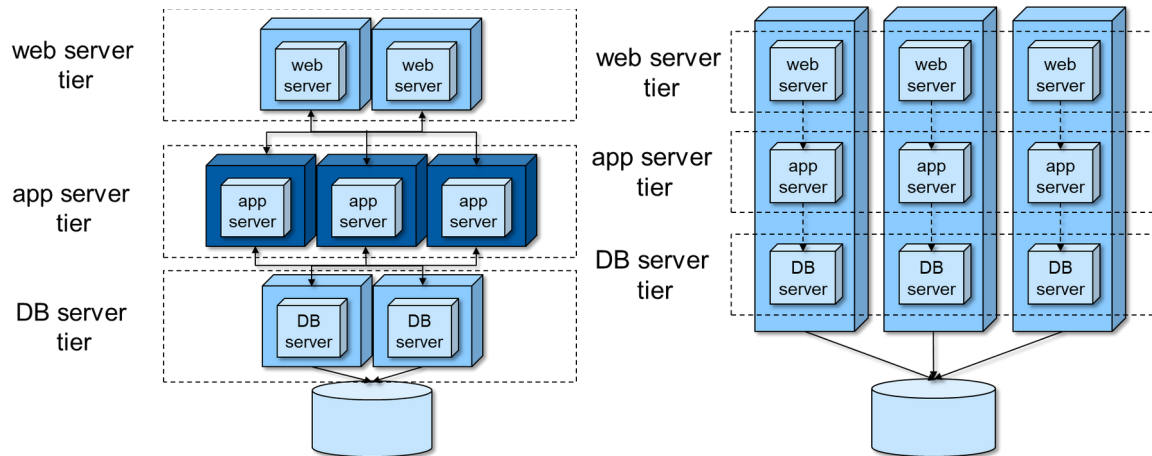


Figure 55: Scaled with specialized machines (left) vs. cookie-cutter scaling (right)

18.4.1 Specialized Machines

One can assign nodes to those (virtual) machines which best meet their requirements. For example, nginx web servers and database servers are rather I/O bound whereas application servers can be rather CPU and memory bound.

The downsides of this pattern are the following:

- There is much real network communication between machines, which is naturally limited in throughput and latency, as opposed to intra-machine communication.
- A redeployment of the whole system is more complicated, if e.g., the new version of a web server depends on the new version of the application server.

18.4.2 Cookie-Cutter Scaling

Another scaling allocation pattern is called **cookie-cutter** [64]. Here, one node of each tier runs on every machine.

The downsides of this pattern are:

- The machines must be one-size-fits-all as opposed to specialized machines. In cloud computing, this has a negative effect on costs, as specialized machines leave more room for cost optimizations.
- Each machine must have exactly one node per tier and there cannot be load balancing between the tiers.

The upsides of the cookie-cutter pattern are that the downsides of the specialized machines pattern can be avoided. The opposite also holds true.

18.4.3 Automated Allocation

As described in Section 25.4.1, container orchestration middleware like Kubernetes can automate the allocation of machines to nodes. Kubernetes uses scheduler algorithms to maximize the usage of the finite set of allocatable machines but also the availability of the overall system. This automation can just be influenced by some extent by means of "node affinity" configurations.

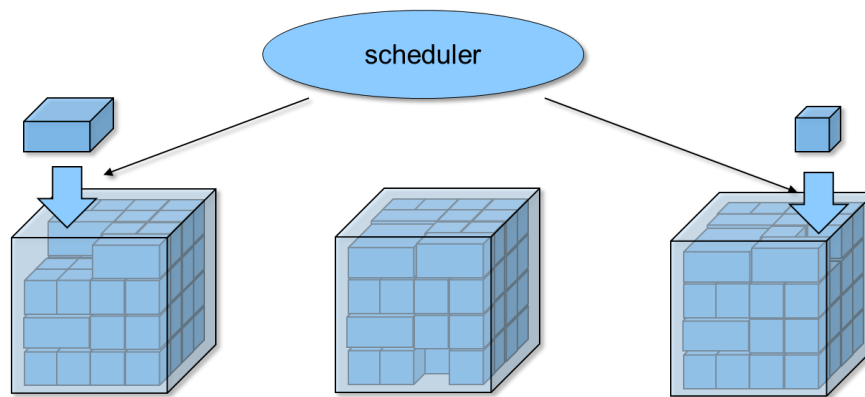


Figure 56: Automated node-machine-assignment

18.5 Cell-based Architectures

In a cell-based architecture, no single nodes but a cohesive set of microservices, their respective databases, and supporting infrastructure components—such as isolated subnets, message brokers, caches, and Kubernetes clusters—are bundled and scaled as a whole. These sets are called **cells**.

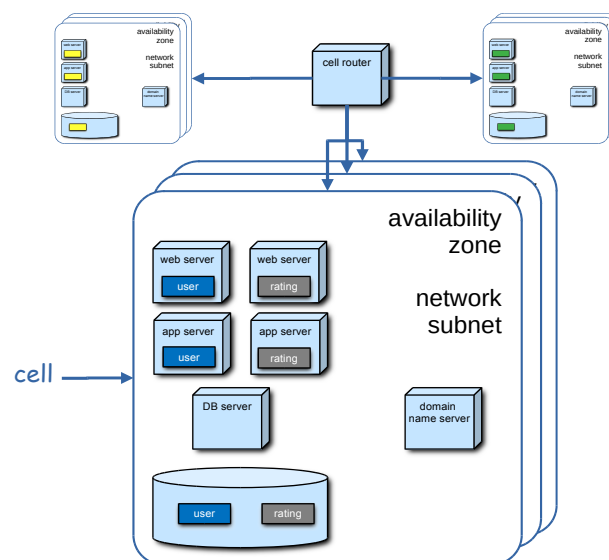


Figure 57: Cell-based Architecture

Because cells follow a strict **shared-nothing approach**, failures within a single cell are completely contained and have no side effects on other cells: The **blast radius** is limited to a particular cell.

EXAMPLE 65: FAILURE SCENARIOS MITIGATED BY CELL-BASED ARCHITECTURES

The failure scenarios include:

- **buggy database migration** scripts, which are isolated to a single cell's database during a phased rollout, preventing global data corruption or table locks,
- **overutilization** and **connections-depletion** of a database as a result from, e.g., **cache stampede**, due to the eviction of a frequently accessed data record, where the resulting thundering herd is physically bounded by the maximum capacity of a single cell's cache and database tier or a rogue request, i.e., an inadvertently complex database query or API flood.

- **cascading timeouts** of dependent microservices and depletion of thread pools, because synchronous service-to-service calls never cross cell boundaries, forcing any architectural meltdown or thread exhaustion to hit a hard brick wall at the edge of the cell.
- **DNS misconfigurations**, where a broken private routing zone or local name server update only goes down inside one cell without disrupting network resolution in the rest of the fleet.
- **network partitions**, because cells are engineered to be completely zone- or region-contained, so fiber cuts or network degradation between these availability zones have no affect on the cells
- cloud provider API **rate-limiting**, since distributing workloads across cells—often housed in completely independent cloud accounts—splits underlying infrastructure quotas and prevents a single **runaway autoscaling loop** from paralyzing the global platform.
- **network congestion** and **port exhaustion** via noisy neighbors, as each cell is provisioned with its own dedicated VPC subnets, internet gateways, and NAT Gateways, trapping localized traffic saturation within its own perimeter.

All requests from a specific client are routed the same cell by means of a **cell router**, meaning a failure within a cell strictly limits the blast radius to only the subset of clients affine to it (cf. 19.3.1b).

18.6 Intra-Tier Communication

Besides allocation to machines, whether and how nodes within a tier communicate and collaborate with each other is another important aspect.

The examples of Figure 53f have in common that there is no communication at all between nodes within the same tier. Even if these nodes run the very same program, their state, i.e., their memory contents may deviate over time due to the different requests they process.

Yet in some cases, communication between nodes within the same tier is unavoidable, e.g., if the nodes must synchronize sessions which reflect the state of the communication with a certain client. A discussion about that can be found in Section 21.2.3.

Replication of caches between nodes it is advantageous to avoid disk I/O in the database tier or—even better—answer requests without querying the database tier at all. In **space-based architectures**, as described in [58], frequently accessed application data is being cached in the memory of the app server nodes.

Specialized middleware like Apache Ignite, Hazelcast, or Oracle Coherence care for the timely replication of these data, which is organized in hash maps. The program in the app server manipulates these in-memory data. The synchronization with the other app servers takes place in the background. Persistence in a real database is also opaque to the program in the app server. In case of manipulation of the application data, the app server writes that change to a message queue which is consumed by a data writer that translates it to a database insert. Conversely, a data reader equips at least the first app server in such a tier with the application data.

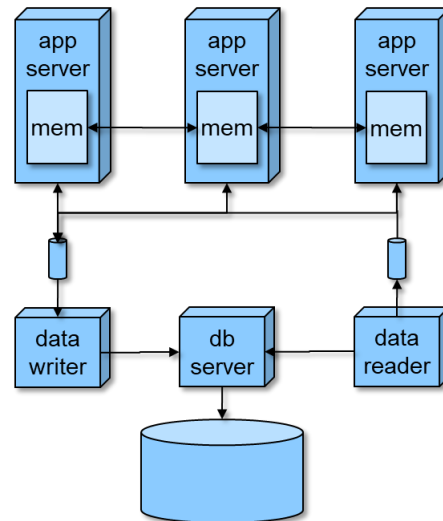


Figure 58: Data grids

18.7 Redeployment of Tiers

Increased availability is among the upsides of a horizontally scaled system. This is particularly true for planned downtimes of single nodes due to redeployment, i.e., the update of software artifacts by newer versions and the subsequent restart of a certain node. There are different redeployment strategies that differ in how incoming requests are routed to the different nodes and in the chronology of updates of different nodes during a redeployment. These different strategies balance desirable properties like zero downtime as observed by clients, testability of the new version with requests from real production clients during redeployment and how these clients can be selected, how long it takes to roll back to the former version and the additional resource consumption.

Table 28 sums up and compares different redeployment strategies. To simplify things, we consider just one tier. Usually, redeployment occurs most frequently in the application server tier within a system cluster. Moreover, we abstract from how the nodes are allocated to one or many (virtual) machines as discussed in Section 18.3.

Redeployment strategy	Zero downtime	Real production workload testing	Releasing to clients based on conditions	Rollback duration	Additional resource consumption
Recreate: v1 is terminated, and v2 is rolled out. Before	x	x	x	Fast but disruptive because of downtime	No extra resources required
Rolling update: v2 is gradually rolled out Before	✓	x	x	Slow (rollback = rolling update reversed)	Can require extra resources for surge upgrades i.e.,
Blue/Green deployment: v2 is released alongside v2; the traffic is switched to v2 after it is tested. Before	✓	x	x	Fast (change in LB)	At least temporary doubled resource consumption
Canary deployment: v2 is released to a fraction of client, followed by a full rollout. Similar to Rolling Update but with tests during rollout. Used to detect bugs in production before all clients are affected. Before	✓	✓	x	Fast (change in LB; only some v1 nodes need to be recreated)	No extra resources required
A/B deployment: v2 is released to a subset of clients that meet certain criteria (LB checks certain http headers, geolocation etc.). Used to determine if new version v2 actually works better than v1, e.g., in a web shop system, leads to higher conversion rates. Before	✓	✓	✓	Fast (change in LB; only some v1 nodes need to be recreated	No extra resources required
Shadow: requests are duplicated and processed by both versions	✓	✓	x	Not applicable	doubled resource consumption

Table 28: Redeployment Strategies in Scaled Systems (after [65])

19 Load Balancing

In Chapter 18, we have omitted an important question: Each request message that nodes exchange represent workload and causes processing and therefore utilization of the machine of in the receiving node. So, how should messages be forwarded to the out-scaled nodes of the same tier, in other words, how should a **load balancing** strategy look like.

In this chapter, we focus on load balancing strategies for application layer messages, HTTP requests most notably. Please note that load balancing is directed: certain nodes are the origins of messages, and some are the destination. For the sake of simplicity, we think of the origins nodes to be clients and the destination nodes to be servers even if other communication patterns as described in Section 21.1 are possible.

Load balancing has some goals in common with **traffic control** but focuses stronger on the right utilization of network resources. **Process scheduling** is another related area but rather refers to algorithms that are executed locally in a machine by the operating system and have the capability to suspend running processes. Also note that load balancing is different from application layer **routing** as described in Section 23.3.1: Load balancing assumes that every known destination node can process every incoming request in principle whereas routing determines the right server based on properties in the incoming request.

19.1 Load Balancing Goals

Load balancing aims at the following goals:

- **Opaqueness:** Clients should be unaware of load balancing at the server side such that a client needs not to be modified (reprogrammed or reconfigured) if a single server is replaced by a scaled tier of servers with load balancing.
- **Equal response times:** Response times should not vary and depend on the load balancing target.
- **Equal utilization:** Every out-scaled node should be utilized to a reasonable degree (cf. Section 5.4). Because out-scaled nodes are usually identical even in their vertical scaling, equal utilization leads to equals response times on average.
- **Reliability:** Load balancing should be able to detect nodes which are effectively down, i.e., do not respond at all or not in an acceptable time range and remove these nodes (temporarily) as balancing targets.

19.2 Load Balancing Strategies

19.2.1 Service-Discovery based Load Balancing

Service discovery means that clients are not programmed or configured with concrete (but potentially volatile) server (network) addresses. Instead, clients are programmed with static names, which they send to dedicated name servers which respond with concrete and currently valid (network) addresses. Of course, the servers behind the addresses are not arbitrary but have exposed web interfaces that clients expect.

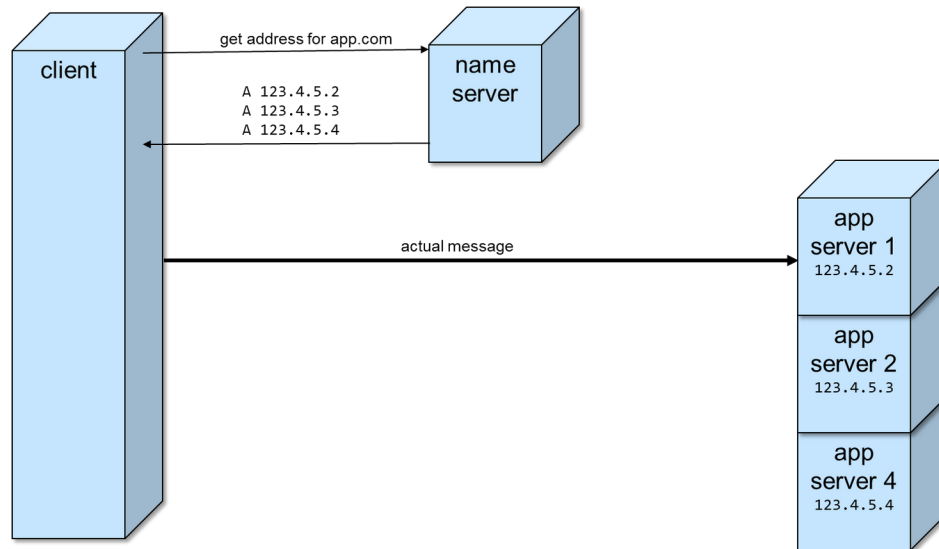


Figure 59: Service-discovery based load balancing

If there are several such servers for a certain name, e.g., because they part of a horizontally scaled tier, the name server can either choose one of the addresses according to a load balancing algorithm or return all addresses and leave the selection to the client.

The Domain Name System (cf. Section 23.8) can be viewed as a system of name servers. In DNS based load balancing, a domain name server responds with an ordered list of A(AAA) records of destination servers to a client query instead of a single A(AAA) record. The client is meant to connect to the first destination server in the list but may fall back to the other entries in case of server failures. As domain name servers lack any information about the status of the servers, they use a simple round-robin algorithm when they compile a list of records for a certain client query.

Obviously, DNS based load balancing misses the goal of opaqueness for the client. The biggest drawback in DNS based load balancing, however, is that DNS makes extensive use of caching which counteracts even distribution of loads and therefore equal utilization and response times. Availability is ambiguous: On the one hand, there is no additional serial node that would decrease the combined availability. Conversely, domain name server put all known matching server addresses in their response and leave the detection of faulty servers to the client.

19.2.2 Dedicated Load Balancer

Usually, a server tier has a dedicated load balancer as part of a specialized inbound gateway (cf. Section 23.3). The inbound gateway exposes a public IP address to which a public domain named is mapped. Clients therefore connect to this inbound gateway. The load balancer forwards messages from clients to the destination servers where they can be processed.

Such load balancers avoid the drawbacks of service-discovery based load balancing. Additionally, the destination servers need not to be exposed to the internet with a public IP address. Thus, an inbound gateway acting as a load balancer reduces security risks. However, after all it is a serial node between the client and the destination server which reduces the combined availability of the system as discussed in Section 8.3.3.

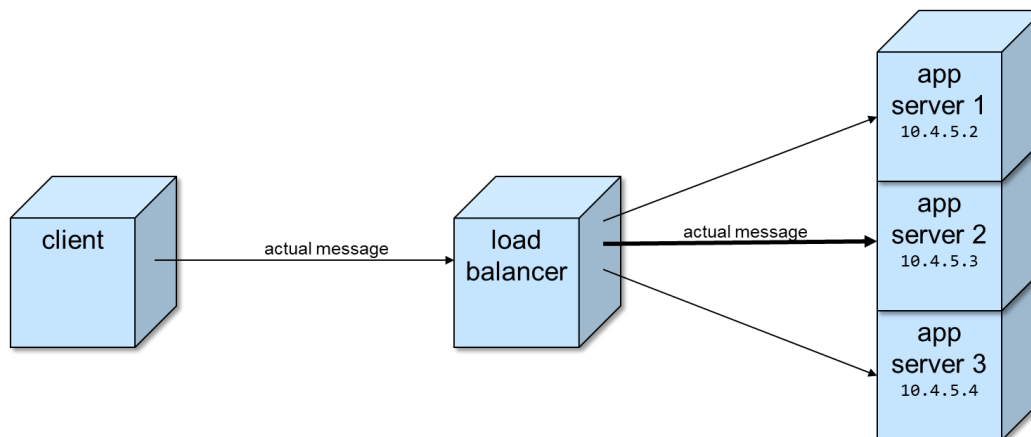


Figure 60: Dedicated load balancer

19.3 Load Balancing Algorithms

While service-discovery based load balancing is restricted to the very simple round-robin algorithm, dedicated load balancers can be implemented with a variety of algorithms to achieve load balancing goals.

19.3.1 Static Load Balancing

Static load balancing is agnostic about the current utilization of destination servers.

a) (Weighted) Round-Robin

Round-Robin is the simplest algorithm and forwards messages to servers in a clockwise manner or any other permutation. In Weighted Round-Robin, one can assign weights to destination servers in the load balancer configuration. The weight of a destination server is proportional to the average arrival rate (cf. Section 5.3) for that server. Weighted Round-Robin is particularly useful to evenly utilize destination servers which are different in their vertical scaling.

b) Client Identifier

In some communications, messages from a certain client must be routed to the very same server. In such cases, we speak of **server affinity**. The most common reason for server affinity is that the server holds (parts of) the session as discussed in Section 21.2.2).

A load balancer therefore must map a unique client identifier to a destination server address. Such identifiers can be found on different layers of the network stack (cf. Section 19.4), e.g., in the TCP packets of messages, namely the source IP address together with the source port. They can also be part of the application layer payload: In case of a HTTP request, such identifiers are typically put into the HTTP header as cookies, named SESSIONID or JSESSIONID.

EXAMPLE 66: SERVER AFFINITY VIA SESSIONID

In Figure 61, there are three client-side sessions each on a different machine. The server-side counterparts are identified via a browser-cookie named JSESSIONID which is transferred in every request. The load balancer stores a mapping of JSESSIONID values to destinations, web server 1 to web server N in the example. This way, requests from a particular client-side session are routed to the very same server that hosts the corresponding server-side session.

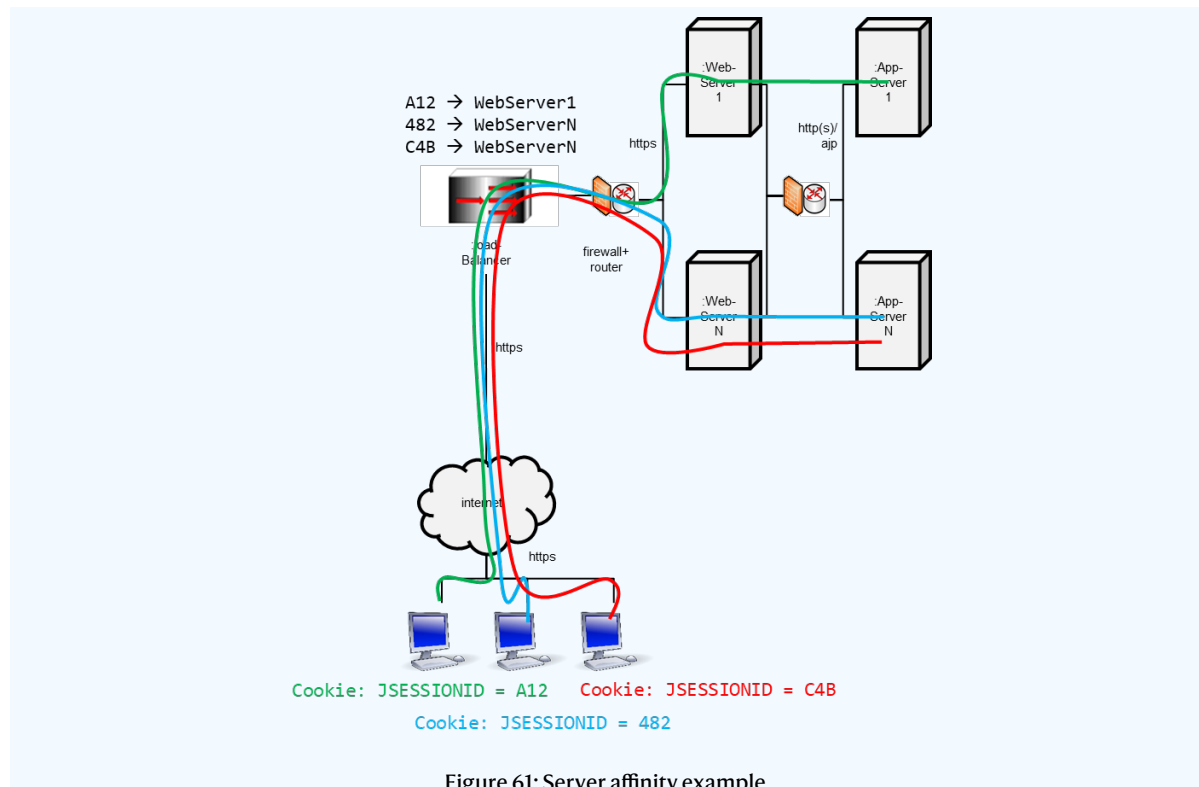


Figure 61: Server affinity example

Using session identifiers in application layer payloads has the advantage that server affinity still holds even if the client IP address changes, which happens, e.g., if the IP address lease expires or has to be renegotiated for other reasons. Using a session ID instead of the client IP address plus source port has one major drawback: The load balancer must be able to peek into the application layer payload of incoming messages. If these are encrypted, the load balancer must decrypt these messages. If the messages are encrypted using the Transport Layer Security Protocol (TLS) as described in Section 30.3, TLS passthrough as defined in Section 23.3.3 is not possible.

EXAMPLE 67: KEY-VALUE CACHE AFFINITY

A horizontally scaled server tier that serves inference for a Large Language Model poses an interesting edge case: clients send the complete context on every request – the latest user message, the conversation history, relevant chunks from a RAG retrieval, and so on – so in principle any destination server can process any request.

In practice, though, the server that processed the previous turn in that conversation has likely already computed and cached the Key-Value (KV) tensors for a prefix of that context. This lets it skip redundant prefill computation and process only the newly appended tokens, making it substantially more efficient than a server encountering that context for the first time. This technique, commonly called prefix caching, only pays off if the load balancer performs cache-aware sticky routing where (a hash value of) the context prefix serves as the client identifier.

Modulus-based Mapping

Yet, the question remains how the mapping of client identifiers to server addresses works. In order to apply some math, we should encode identifiers integer numbers.

EXAMPLE 68: MODULUS-BASED MAPPING (1)

Say, we have a client with IP address 132.2.10.6 and source port 23456. Such a combination can be unambiguously mapped to the number

$$132 \cdot 2^{40} + 2 \cdot 2^{32} + 10 \cdot 2^{24} + 6 \cdot 2^{16} + 23456 \cdot 2^0 = 145144292989853.$$

Of course, the identifier space, i.e., the set of all valid identifiers is usually much larger than the set of valid destination server addresses. Therefore, we have to apply a modulus operation.

EXAMPLE 69: MODULUS-BASED MAPPING (2)

If we have 7 destination servers, numbered from 0 to 6, messages from this client would be forwarded to server 2, since $145144292989852 \bmod 7 = 2$.

Since IP addresses and source ports are not evenly distributed, it is advisable to put the (mapped) IP address and source port into a hash function as explained in Section 30.2.1 and compute the modulus of that hash value.

EXAMPLE 70: MODULUS-BASED MAPPING (3)

$$\text{sha } 256(145144292989852) \bmod 7 = 5$$

At the end, the computed value must be mapped to a valid hostname or IP address of the server, which can be done via a simple lookup table in the load balancer.

Consistent Hashing

Issues with modulus-based mapping occur when there is a change in the total number of available destination servers. This happens, e.g., during vertical scaling or if a server failure is detected.

EXAMPLE 71: CONSISTENT HASHING (1)

$$\text{If we scale out our server tier to 8 servers, we get } \text{sha } 256(145144292989852) \bmod 8 = 4$$

Of course, we must accept that clients that were assigned to a faulty server which is taken out of the set of valid load balancing must be reassigned to another server. Likewise, a newly added server initially has zero assigned clients. However, we should implement client identifier-based load balancing in a way that assignments to unaffected servers remain unchanged. This can be achieved using consistent hashing.

Consistent hashing maps client and server identifiers to the same clock-like structure integers, i.e., where the highest integer is again a predecessor of the lowest integer, normally 0. Because client as well as server identifiers are both mapped to the structure, they can be compared. We assign a specific client c to a specific server s , if that s is the closest predecessor of c in that clock-like structure.

Usually, the size of the click, i.e., the number of integers on that clock is huge compared to the number of servers. Since the mapping of servers to the clock is essentially random and therefore generally not equidistant. This results in an uneven distribution of clients to servers. We can mitigate this by another indirection: Each server gets the same number of virtual servers, each with a distinct identifier and thus a distinct mapping to the clock. The segments on the clock which are delimited by the mapped virtual servers are thus shorter and vary less in size.

EXAMPLE 72: CONSISTENT HASHING (2)

In Figure 62 we see an example for consistent hashing. In sum, we have 15 clients (green) each identified by a speaking name `client_0` to `client_14` for simplicity. (Only those clients that are relevant in the following are labeled in the figure.) We have three servers `server_0` to `server_2`, each of which is represented by virtual servers 0 and 1, so we arrive at names like `server_1_0`. We use a hash function that maps `client_3` to `server_0_0` for example, meaning that requests from `client_3` are forwarded to `server_0` by the load balancer. If we take `server_0` out of the load balancer, `client_3`, `client_14`, and `client_8` have `server_2_0` as their new nearest server predecessor and thus assigned to `server_2`, whereas `client_4`, `client_12`, `client_11`, and `client_0` have `server_1_0` as new nearest predecessor and are thus reassigned to `server_1`. The other clients (without a label), remain at their respective servers `server_1` and `server_2`.

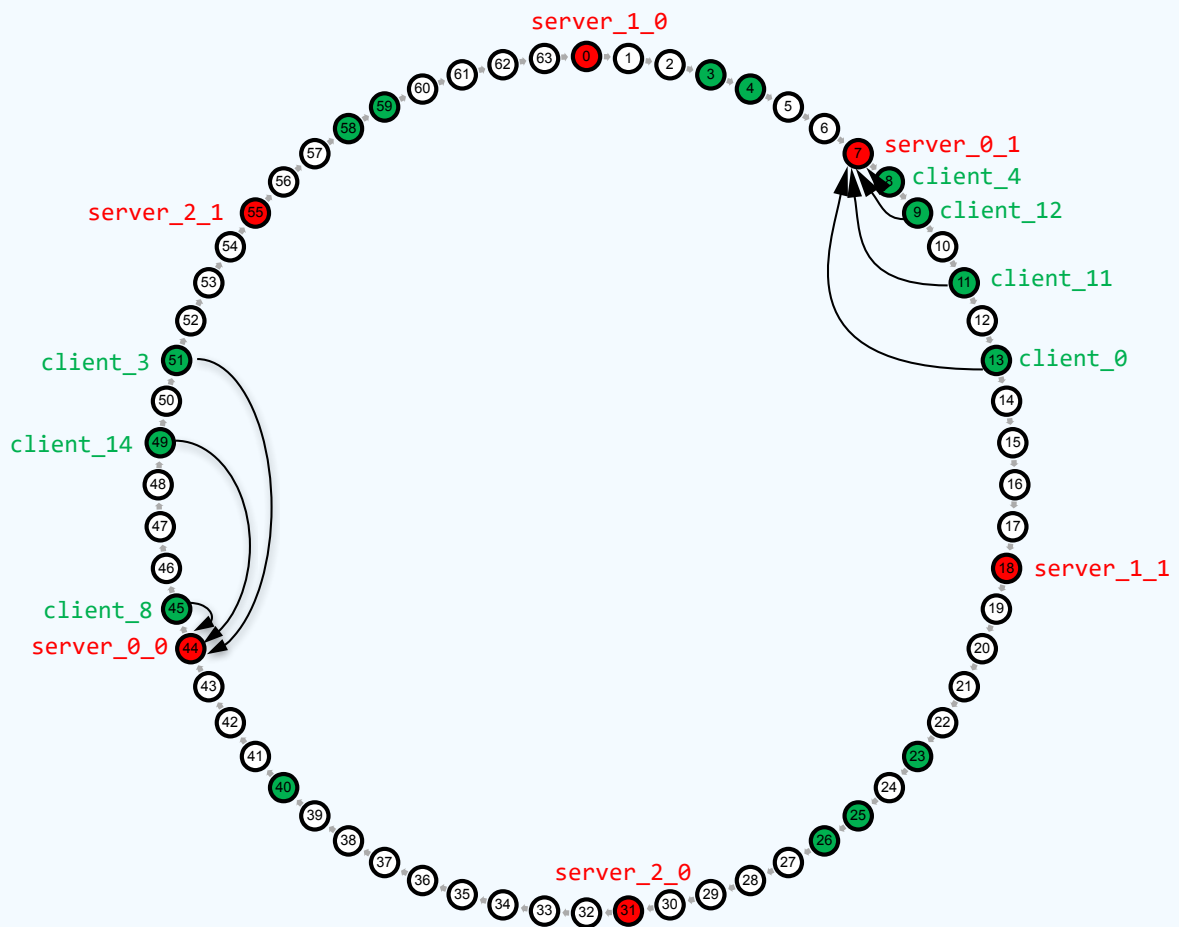


Figure 62: Consistent hashing example

19.3.2 Dynamic Load Balancing

Dynamic load balancing continuously adjusts its load balancing strategy to the (assumed) utilization of destination servers.

a) Least Connection

The number of open TCP connections between a load balancer and a destination server varies. If no packets traverse a TCP connection for a certain time, the connection is usually closed. If this timeout is low enough, the number of open connections correlates with the utilization of the server.

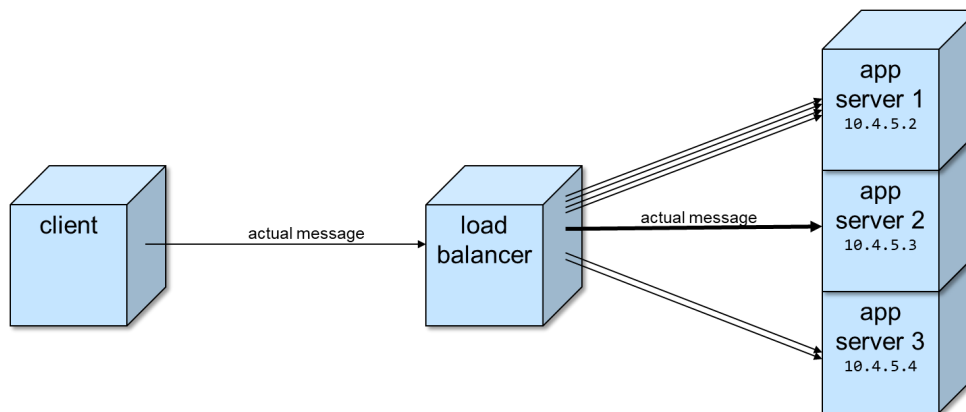


Figure 63: Least Connection load balancing

A load balancer can therefore route a message to the destination server with the least number of connections.

b) Least Response Time

If the destination servers are upscaled unevenly, the number of open connections may not reflect the actual utilization properly. A load balancer can therefore also base its routing decision on the response times of the destination servers. Of course, least response time algorithms must compute a reasonable average of these response times, e.g., take only recent response times into account.

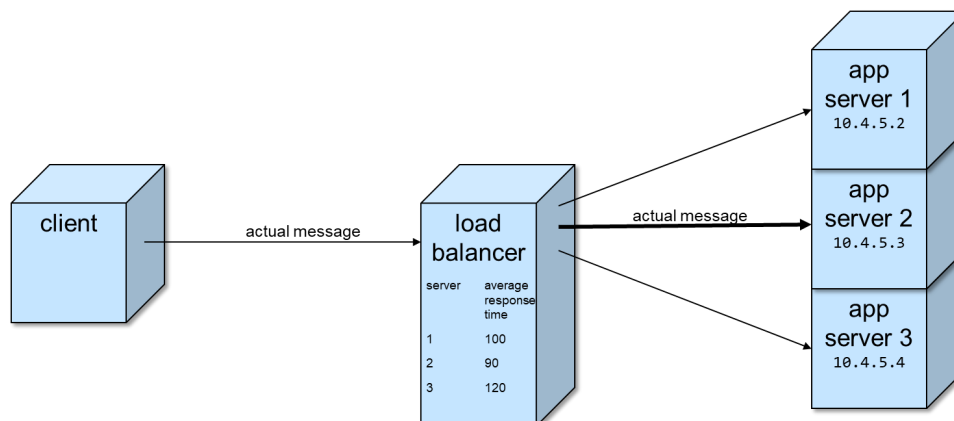


Figure 64: Least Response Time load balancing

c) Resource-based

The most accurate way to route messages to the least utilized server is of course, if the load balancer actually knows about the utilization of each destination server. This can be achieved by installing programs often called agents on each destination server that repeatedly measure utilization and feed the measurements back to the load balancer.

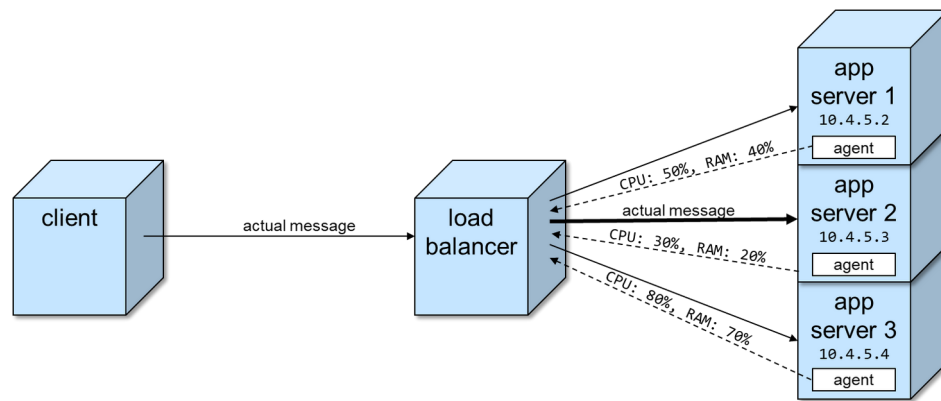


Figure 65: Resource-based Load Balancing

19.4 Load Balancing Network Layer

So far, we have focused load balancing on the application layer (L7) of the ISO/OSI network stack. Load balancing on the transport layer (L4) is also possible. The following table characterizes both approaches. In summary, L4 is agnostic of upper layers and thus more flexible and easier but on the other hand less capable with respect to load balancing of specific application layer protocol.

Criteria	Layer 4 Load Balancing (L4)	Layer 7 Load Balancing (L7)
OSI Model Layer	Transport Layer	Application Layer (HTTP assumed in the following)
Support for Upper Layer Protocols	Supports arbitrary security and application layer protocols, e.g., HTTP, MQTT	Specific for certain application layer protocols like HTTP
Load Balancing Subject (Load Balancing for each...)	TCP connection, UDP packet	HTTP request
Traffic Inspection	Limited to IP, TCP/UDP headers	Inspects HTTP headers and bodies
TCP connection handling	TCP connection between client and load balancing destination server OR between client and load balancer and load balancer and destination server	TCP connection between client and load balancer + TCP connections between load balancer and destination servers.
TLS connection handling	passthrough, thus no TLS offloading	termination, thus TLS offloading
Client Identifiers (cf. Section 19.3.1b)	IP address, port, protocol	URL (e.g. query strings), cookies, HTTP headers, content
Configuration Complexity	Simpler to configure and manage	More complex, with advanced routing rules
Computational Complexity / Utilization	Lower, as it deals with less data	Higher, due to deep packet inspection
Applicable Load Balancing Algorithms (cf. Section 19.3)	- (Weighted) Round-Robin - Client Identifier (on Source IP/port)	- (Weighted) Round-Robin - Client Identifier (on Cookies) - Least Connection

Criteria	Layer 4 Load Balancing (L4)	Layer 7 Load Balancing (L7)
	- Least Connection - Resource Based	- Least Response Time - Resource Based
Failure Detection	Restricted to L4: TCP/UDP port checks, ICMP pings	HTTP requests to health check URLs
Failover	- TCP connection draining for planned destination server downtimes - Excluding unhealthy server for new connections - Clients with connections to unhealthy servers must reconnect	- Excluding unhealthy server as destination for incoming HTTP messages - In principle, HTTP clients can be kept agnostic of a failover
Flexibility in Load Balancing Algorithms	Limited (round robin, least connections)	More flexible (content-based, weighted)
Security Features	TCP/UDP based checks (source IP, port etc.)	Content-based checks / application-level firewall
Other Possible Inbound Gateway Functions (cf. Section 23.3)	Packet-based rate limiting	Message routing, message transformation, TLS termination/offloading, authentication/authorization, intrusion detection, rate limiting, usage monitoring

Table 29: Load Balancing Network Layer Comparison

20 Distributing Systems

In this chapter we discuss the up- and downsides of distributing a single system beyond just scaling it.

20.1 Avoiding Distribution with Monoliths

A monolithic system (short: [monolith](#)) consists of just a single program that implements a coherent application on its own. If that system is scaled in the sense of Chapter 18, there is just one application server tier in which a single program is being deployed. Typically, such a program is being deployed by a [single deployment artifact](#), ([deployment monolith](#)) e.g., a big "Java archive" file (jar) or an executable file.

A monolith's code is typically being developed by a [big team](#) using a [single technology stack](#) (Java, Ruby, Golang, .NET ...). A [monolith](#) is typically and versioned in a [single big repository](#).

20.2 Downsides of Monoliths

In monolithic architectures we avoid distribution except for scaling. This has some up- and downsides. Since monolithic architectures are in a sense one side of a spectrum and microservices on the other side, we just discuss the downsides of monolithic architectures here. You can consider the downsides of distributed systems and microservices in particular, which we will discuss in Section 20.5, to be relative upsides of monoliths.

20.2.1 Organizational Downsides

- Big teams tend to work inefficiently as [communications paths](#) grow quadratically with each new team member.
- Team members might not be familiar with the monolith's [single technology stack](#).
- Large teams working on a large monolith have a higher risk of committing erroneous source code changes that break the build or tests of the monolith and hamper other team members' [productivity](#).
- Oftentimes, development within a company is outsourced to [different suppliers](#). It is unlikely that different suppliers are willing to develop a joint monolith since [responsibilities and liabilities](#) are then hard to delineate.
- Different functional areas may have [diverging constraints or quality requirements](#). For example, requirements for payment processing rarely change. It must predominantly be reliable. In contrast, a recommendation engine works on everchanging data and assumptions. Ideally, changes here are tested using an A/B deployment (cf. Section 18.7) to figure out in production what works better.

20.2.2 Architectural and Technical Downsides

- Common programming languages offer constructs for code structuring, information hiding and modularization. However, these constructs remain optional: After all (except for distribution due to technical reasons), a monolith is developed as a single program running in a single process in which all data and program units are potentially accessible from every other program unit. Only process and even more network boundaries actually enforce loose coupling and real information hiding. Yet, monoliths [lack these boundaries](#). Therefore, maintaining loose coupling of program unit highly depends on the discipline of its developers. Yet, discipline tends to fall victim to [technical shortcuts](#) when deadlines close in.

EXAMPLE 73: TECHNICAL SHORTCUTS

A typical shortcut is a function call from function A to a function B, which was not intended by the creator of B, thereby introducing an unsolicited coupling between both functions.

Another typical shortcut is that program unit A creates an alias (pointer or reference) to a data object, which is "owned" by another program unit B. Programmers might have a hard time here to figure out how and why the value of this data object changes.

Accumulated shortcuts turn a once **structured monolith** into a so-called **big ball of mud** with sprawling **technical debt**. Huge surface areas for **unsolicited side-effects** unsettle developers who then shy away from necessary changes including those refactorings that would bring relief. Even if a system remains a structured monolith, unaware developers must assume that unexpected behavior may be caused by a hidden technical shortcut since those shortcuts cannot be ruled out in principle, e.g., by the aforementioned presence of network boundaries.

- Technology stacks (languages, libraries, frameworks) become ultimately out of fashion, which makes it harder to acquire developers which are familiar with that stack. It also bears the risk that parts of the stack get out of maintenance (even for security updates). Normally, a program's migration to another technology stack can only be done a single step. This is harder for a monolith due to its larger program size. Conversely, a system of microservices (i.e., multiple rather small programs) can be migrated incrementally microservice by microservice.
- **Build-times** (including compilation and test automation) and **start-up times** might grow large for monoliths (up to hours) therefore hampering rapid development of even small changes.
- **Testing just units** (which would map to a single microservice) of a monolith is therefore comparably tedious.
- The larger a program, the higher the probability of bugs that risk the running process' **faultlessness**. At runtime, such bugs might cause a failure of the whole system and render the respective application inaccessible for users. Such bugs include
 - too much or frequent allocation of heap **memory** that could cause
 - **garbage collector stops** for the whole process or, d
 - even worse, get the respective process killed by, e.g., the Linux' Out of Memory Manager (OOM) [66, p. 685 ff.]
 - overconsumption of **CPU** power due to, e.g., infinite loops or accidental complex computations like evaluations of certain regular expressions [67]
 - depletion of **thread pools**.⁴¹ since, e.g., all threads got stuck at the same call (to, e.g., some external service) that is relevant just for a certain functionality.

EXAMPLE 74: PERILS OF A ReWo-MONOLITH

Larger monoliths tend to contain functionality of different criticalities. For example, if ReWo-Rate remains a part of a monolithic ReWo-Shop system, frequent updates in the ReWo-Rate related program parts could introduce new bugs that also endanger the availability of functionality like checkout and payment which is of higher business criticality than rating.

Please not that an operating system kernel can only schedule and allocate resources like CPU and memory among different processes but not among different units of a program that runs in a single process. Conversely, in microservice architectures there is a change for keeping such bugs

⁴¹ Using Apache Tomcat embedded in in Spring Boot is part of many server-side technology stacks. By default, the Java thread pool is limited to 200 threads (cf. property `server.tomcat.max-threads` in [68]). So, a single JVM process can process at most 200 requests in parallel.

isolated in the respective microservice and gracefully degrading the application by just switching off the affected functionality.

EXAMPLE 75: GRACEFUL DEGRADATION IN ReWo-RATE

In ReWo-Rate the non-availability of a microservice for the user management, which implements use cases like "user registration", can be "gracefully degraded" by redirecting the respective registration link to a static and informative HTTP 500 web page.

- Even if just small parts of a monolith need to be updated, **redeployment** means that the monolithic process must be restarted. Since a monolith normally exposes a lot of functions (invokable via different routes, API REST resources, web pages etc.), a lot of functions are potentially unavailable during the redeployment. This problem can be mitigated if the monolith is scaled horizontally, and the redeployment is conducted as a **rolling update** (cf. Section 18.7).
- Moreover, the larger a monolith the higher the risk of an **erroneous redeployment** due to wrong configurations that would prevent the monolith from starting and again affecting a whole big application's faultlessness. Even worse, the risk is higher for **bad database (schema) modifications** thus risking the existing data of the whole application instead of just a subset.
- Processes running monoliths utilize a lot of resources like memory that makes them **unsuitable for binpacking** (cf. Section 24.4) or **fast migration** or even **fast rescheduling** on and between different machines particularly in cloud environments.
- Different program units within a monolith...
 - can only be **out-scaled together**. For example, a program unit that allocates much memory (e.g. a huge hash map), but which is rarely called might not be out-scaled as much in order to save memory.
 - might deal with **different workload types** (cf. Table 8). However, these program units cannot be deployed to different targets with specialized hardware. For example, if a web application monolith includes analytics and generative AI, i.e., model inference, it can only perform well on machines with plenty and fast resources of all types (CPU, GPU; memory, storage, network throughput). In cloud environments, all-rounder machines are expensive compared to specialized machines.
 - might also be rather **stateless** (cf. Section 28.2 and functional in nature while others inherently **stateful**. The former would be suitable for running in ephemeral processes and containers (cf. Section 25.4) or as Function-as-a-Service (cf. Section 24.2.4), the latter would not.

20.3 Distributed Systems

In a distributed system, different programs run in different nodes and communicate with each other to realize a single (larger) application. Each program forms a subsystem,

- which implements just a portion of what would subjectively⁴² considered an application,
- that again might be distributed but just for technical reasons,
- has its **own runtime environment**,
 - i.e., runs in its own processes which run in so-called in containers (cf. Chapter 25.3),
 - therefore, it may have its **own technology stack**,
 - therefore, it can technically be **(re-)deployed independently** from other programs,

⁴² This subjectivity makes it sometimes hard to tell if a system is a (sub-)system / microservice or realizes an application on its own and therefore is a monolith.

- communicates with other programs mostly **via network protocols** or inter-process communication since in-process calls are not feasible.

Please note that this understanding of a distributed system abstracts from the distribution on different machines. In principle, with nodes that are local to each other, e.g., run on the same (virtual) machine, one can avoid most of the challenges as described in Section 20.5.2. However, remote nodes that run on different machines are the normal case. Two communicating nodes that are local to each other normally is a mere coincidence which has to be rather attributed to a scheduler (cf. Section 25.4.2) than being a conscious decision.

There are certain **enabling factors** that partially gained traction in the last years and facilitated the breakthrough of distributed systems and the microservice architecture style in particular:

- Virtualization, containerization, and cloud computing make rapid **provisioning** of resources (machines, storage, network, etc.) easy. Before that, provisioning efforts would have stifled microservice architectures.
- Speed and **time-to-market** gained importance, while the diversity of technology stacks became larger. It is faster and more secure to let a team implement a working minimal viable product or function in a new microservice with their favorite technology stack than to extend an old and big monolith.
- **Conway's law**, which states that the architecture of a system or of a system landscape tends to resemble organizational structures anyway. According to [69], such alignments actually help to improve the quality of a software system and speeding up development. Architectural Styles like microservice architectures (cf. Section 20.4.1) are better suitable to reconcile with organizational structures than one or few monoliths could do.

20.4 Architecture Styles for Distributed Systems

In Chapter 17, we discussed how we can express architecture with the source code of a single—hence not distributed—program. Of course, such software architecture can also be applied to every single program in a distributed system. However, one layer above that, architecture styles can be seen in how the separate nodes of a distributed system are being arranged.

20.4.1 Accidental Distribution

Sometimes, the distribution is rather due to historical reasons than due to deliberate architectural choice. Systems of enterprise system landscapes are frequently affected by **organizational changes** like restructuring of departments as well as mergers and acquisitions. So, systems that were once developed separately need to be integrated in a post hoc manner. In most cases, it is out of the question to merge their code bases and to make one program out of two. Instead, they are left distributed.

20.4.2 Microservices

The most popular style as of 2025 for distributed systems seems to be the microservices style. A microservice is a small and technically independent (scaled) monolith, which contributes a part to the overall functionality of an application (typically: web application) but does not constitute an application on its own.

A system in a microservice style has the properties of a distributed system as described in Section 20.4 and additionally the following:

- A system of microservice is **modularized by functionality** (cf. Chapter 17.1.2), where each module runs in a system cluster on its own as opposed to modules in a monolith which are just (bigger) program units,
- **Redeployment** of a single microservice with a simple recreate strategy (cf. Section 18.7) should at worst just render some functionality of the overall system temporarily unavailable.
- Each microservice system cluster usually has its own application server tier but also its own database tier. Even if several microservices must share a database tier and store their data in a single shared database in order to reduce (license) costs, their data is stored in a separate schema, bucket, key space, or whatever concept the respective data model provides (cf. Section 32.2). Ideally, the shared database is replicated, where each microservice has an exclusive replica. This makes a single microservice independent of the availability of a shared database. When a microservice has its surrounding middleware tiers exclusive by itself we talk of a **shared-nothing architecture**.
- Microservices do not communicate over inter-process communication (IPC) but only over network protocols and mostly over HTTP and its derivatives (cf. Chapter 27), therefore, expose HTTP-based application programming interfaces (**API**) or web user interfaces (**UI**).⁴³
- Each microservice typically has its own code base in its **own source code repository**.⁴⁴
- Each microservice is **incrementally and iteratively developed as a product** as well as operated by a stable “**two-pizza team**”, i.e., a team of about 6 people.

If a monolith in a layered architecture is redesigned in a microservice architecture style, each microservice itself again constitutes a (scaled) monolith which of course can again follow a layered architecture or any other monolithic architecture style. So, microservice are nothing but small monoliths and there is no objective measure when a microservice becomes big enough to be considered a monolith.

EXAMPLE 76: SPLITTING A REWO-MONOLITH

In Figure 66, we have split the scaled monolith of Figure 55 into two microservices each of which again is a scaled monolith in itself. We assume that the application is something like ReWo-Rate.

Each tier is affected in different ways by the split: The web servers might serve static content that either belongs to user- or rating-related web pages, the application servers contain just those program parts that either belong to user or rating related functionality and the database must also contain just the schema parts, i.e., those tables, sequences, triggers and the like which belong the respective functionality.

⁴³ If a microservice serves its own web UI, this UI is also called **micro frontend** [70]. A microservices that serves its own micro frontend and has a database on its own is also called **self-contained system** [71]

⁴⁴ This property of microservice seems to fade in recent years. There are microservice architectures, whose microservice share the same code base. For example, Google's code base is stored in a so-called monolithic repository (**mono repo**) but deployed and run in a microservice fashion [72]

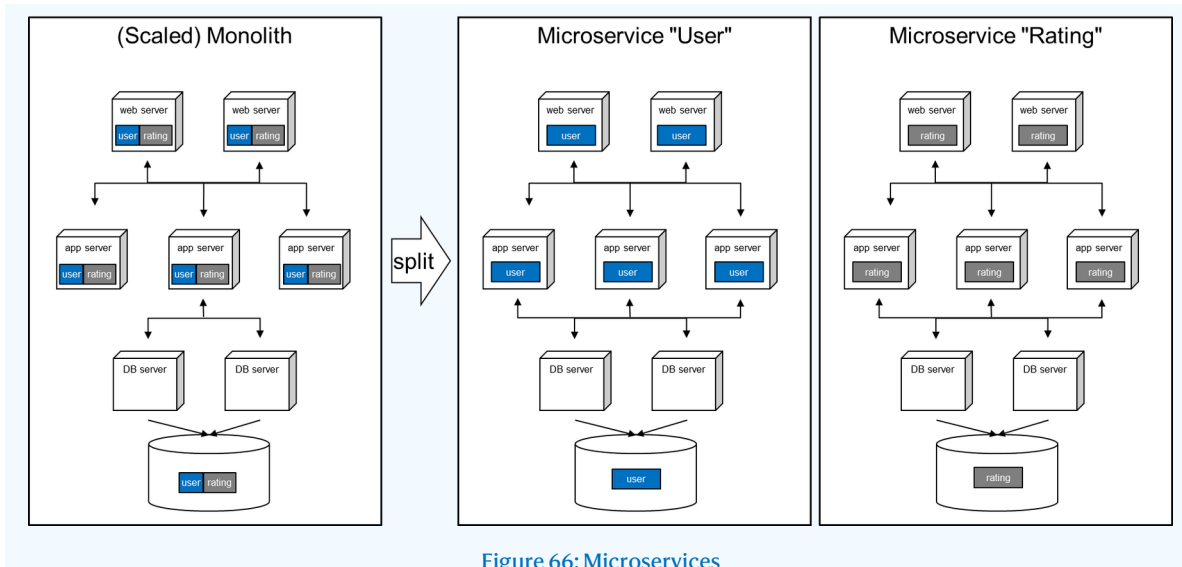


Figure 66: Microservices

20.4.3 Event-driven Systems

In an event-driven system, the distributed programs consistently communicate via the messaging pattern (cf. Section 21.5) instead of a request-response oriented pattern (cf. Section 21.1.1). The name of this pattern stems from the fact that exchanged messages contain data about domain events like "order created".

Senders (publishers) of such events do not need to be aware of their receivers (subscribers). This leverages the decoupling to some extent.

A more nuanced discussion of event-driven systems can be found in Section 26.1.3.

20.4.4 Domain-driven Design

Domain-driven Design (DDD) as introduced in [14] is a design methodology that – similar to Object-oriented Design – focuses on modeling the business domain of a system as a set of interrelated typed objects. It strives to establish a glossary called **ubiquitous language** containing such types. A type either represents a (mutable) **domain entity** like Customer, a typed **value object** like "200\$" (type would be Money) or an **event** like PaymentFailed.

Domain-driven Design domain models basically have four layers of decomposition:

- The most granular context layer 1,
- a layer 2 with so-called **bounded contexts**, e.g. OrderProcessing, that are further refined in
- a layer 3 with **aggregates** which are detailed
- in layer 4 by individual types like the **aggregate root** Order with and aggregated type like OrderLineItem.

EXAMPLE 77: BOUNDED CONTEXTS

Figure 67 shows three bounded contexts (white hexagons) decomposed to layer 4 each with one or two aggregates.

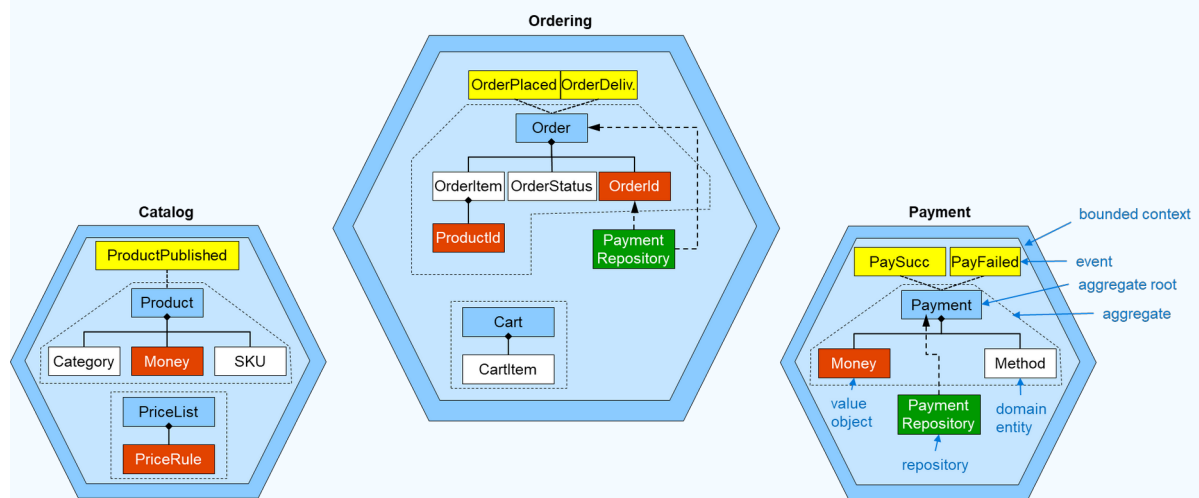


Figure 67: Bounded Contexts

The contents of the grey areas are left out for brevity. They contain technical classes that implement ports and adapters as described in Section 17.3.

Domain-driven Design advocates to implement as much pure business logic in the entity classes as possible. If some logic does not naturally fit into one of these entity classes, it is put into **domain service classes**. Classes of the bounded contexts have no dependencies to the ports and adapters. Ideally, these classes at most depend on each other and on the standard library of the respective programming language but no other libraries nor frameworks.

Domain-driven Design is particularly useful in transactional systems with heavy, unclear and change-prone business logic (pricing, logistics, healthcare etc.). It provides a systematic guidance how to decompose a larger system by function into microservices or at least loosely coupled modules. In other systems, like analytics systems, the (transformation) logic is better put elsewhere than in entity classes and domain services.

20.4.5 Client-Server Systems and Peer-to-Peer Systems

If we focus on the mutual roles of communicating systems, we recognize a client-server pattern, i.e., a server (system) that serves many clients. The server is central in this pattern, and all clients depend on the server. Web applications naturally follow this pattern if you consider the browser to be a client. Chapter 20.6 further characterizes this pattern regarding communication patterns between clients and servers.

If all systems both act as clients and servers, we speak of a peer-to-peer pattern. Peer-to-Peer systems have no central node on which other nodes depend on. Filesharing networks and crypto currencies work in this fashion. Oftentimes, peer-to-peer systems still need a central node by which the other nodes, i.e., the peers, can discover each other in the network. Also, peer-to-peer systems have no natural leader node if data is replicated as opposed to client-server systems where clients typically have no or at best some kind of subordinate follower replica.

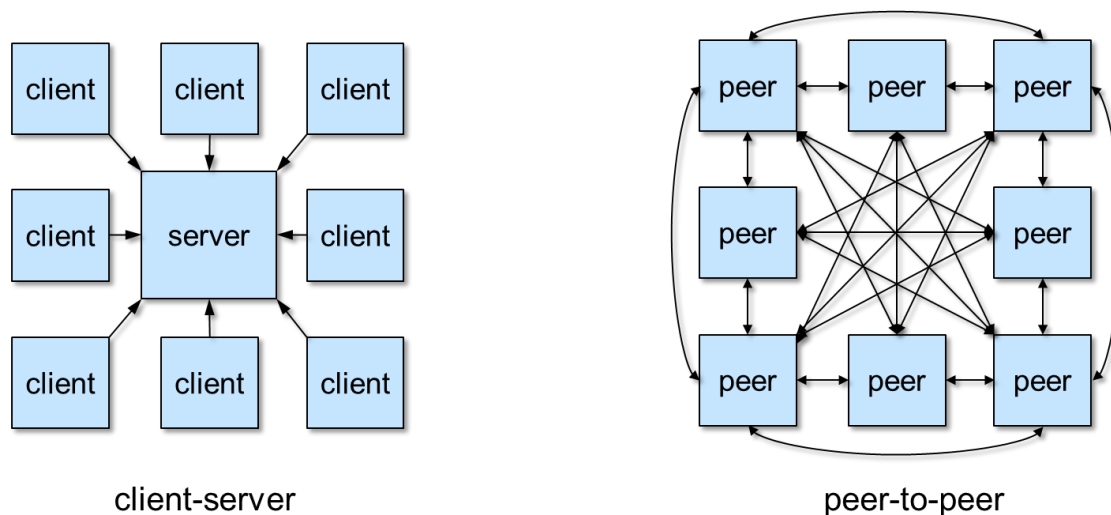


Figure 68: Client-server vs. peer-to-peer architecture

20.5 Downsides of Distributed Systems (Microservices)

The downsides of distributed systems and microservices should not be neglected. In a way, they resemble those of distributed systems in a typical enterprise system landscape, except that microservice are supposed to be distributed with thoughtful boundaries that once might have been parts of a monolith. Conversely, systems in a system landscape have typically been developed mostly independent from each other and often have no clear delineation.

Again, the up- and downsides are relative to the other side of the spectrum, which are monoliths as discussed in Chapter 20.1. So, think of the upsides of distributed systems that they avoid the downsides of monoliths.

20.5.1 Increase of Complexity

Although each microservice has a lower complexity, the entirety of all microservices of a system has a **higher complexity** than a well-structured monolith of equal functionality.

- The **diversity** of technology stacks makes it harder to keep track of deprecations, necessary security patches and to reassign developers.
- **Integrating microservices** requires additional efforts, e.g., glue code for network communication. This is particularly true if different microservices serve parts of the web UI, i.e., have micro frontends, or parts of a web API of a single web application, which is supposed to have a **coherent user experience** with a **single sign-on**.
- Starting a system of microservices requires more effort. Testing such a system as a whole, which requires it to be in a testable state, is therefore harder to achieve.
- Operations faces **more metrics** as well as **distributed and disparate log files** to follow and correlate.
- Distribution does not automatically loosen coupling (cf. **decoupling fallacy** in [73, pp. 240–241]). Even worse, what is a benign dependency in a monolith, like a normal in-process function call or dereferencing of a heap memory object, might become more malignant in a functionally equivalent microservice architecture if these dependencies overarch microservices boundaries. In this case, **broken dependencies**, e.g., a deviation of a function call from the function's definition, are harder to detect. This makes it harder to avoid new bugs during independent evolution of these microservices and even to avoid regression during a

refactoring of the overall code base. So, a decent **design up-front**, which particularly avoids overarching dependencies is even more vital for microservice architectures but nonetheless hard to achieve. These downsides can be mitigated if dependent microservices are versioned in a single so-called **mono-repo** and built in concert such that overarching broken dependencies are also detected at build time. Such builds still can produce distinct deployment artifacts for each microservice such that they can be deployed independently and run distributed.

- Independent deployability of different microservices is also a kind of an illusion.⁴⁵ as it comes with the cost that an overarching dependency might just harmfully break at the event of the deployment (and not before). Typically, instead of compiler errors at build time one finds HTTP status codes like “403 – Bad Request” in the logs at run time. Redeployment patterns (cf. Chapter 20.6) provide some remedy here.

20.5.2 Distribution of Systems

Some **problems are inherent to distributed systems** and grow bigger in a more distributed system. These problems include the following:

- Communication over a network is inherently **unreliable**, e.g., due to congested network buffers. Conversely, in-process calls between different components within a monolith never fail. Therefore, it is particularly hard to guarantee ACID properties for **distributed transactions**, particularly if performance is important. Although there is a standard protocol for distributed transactions named X/Open XA [74], many NoSQL database and message brokers do not support since it relies on a two-phase commit protocol [24, p. 354] which trades off **performance against consistency** (cf. Section 36.2).
- **Sharing in-memory data** between separate processes is infeasible if they are really distributed among different machines. Consequently, passing just memory addresses (references or pointers) is out of the question. Therefore, processing data in a distributed fashion requires **frequent data transfers** and thus **replication** of that data along with **additional processing times** for (de-)serialization. From the perspective of isolation, this might be considered an upside of microservices, from a performance perspective it is a downside.
- Communication over network induces **latencies** (cf. Section 5.2.2) in the order of milliseconds, whereas latencies of in-process calls are in the order of nanoseconds and thus neglectable⁴⁶. In particular, whereas in-process calls might just pass a small reference to a common but very large data object, **transmission time** (cf. Figure 14) gets recognizable when this large data object (e.g., a video) needs to be passed between microservices (which are, e.g., forming a kind of video processing pipeline). Moreover, **propagation delays** get significant if they add up since communications between two microservices (1) occur frequent and (2) are synchronous or (3) the microservices are quite far apart (e.g., not in the same data center). At least (1) and (2) can be mitigated to some degree: If one microservice B processes an array of data elements for another microservice A, then A should better put the whole array in a single request to B instead of looping over the elements and waiting for a response in every iteration or at least do the request-responses asynchronously. These problems have been observed even before the advent of microservices, as Martin Fowler points out in [75].
- Due to the faultiness and latency of networks, it is hard to reach a **system-wide consensus** even about basic facts such as the exact current local time [24, p. 287]. This turns out to be annoying

⁴⁵ also known as the **independent development and deployment fallacy** [73, p. 241]

⁴⁶ An in-process call in source code can also become no call at all in machine code due to a compiler optimization called **inline expansion**. This reduces latencies even more.

when one tries to trace the exact [chronology of events](#) based on log files from different machines [24, p. 292].

- Data in transit, i.e., communication over networks must be [secured](#) to avoid sniffing or tampering of messages. In-process calls do not need to be secured.

20.5.3 Increase in Overall Footprint

Early application servers were designed to run multiple (monolithic) applications. This was also due to reducing the overall memory footprint. To achieve sufficient isolation, running just a single application (packaged in a WAR or EAR file) per application server became best practice anyway.

With microservices, this approach has become even more extreme: Although, full-blown application servers got out of fashion, each microservice still comes along with a significant [memory footprint](#). Splitting a monolith vertically (cf. Figure 49) yields microservices each with a memory footprint right after start-up.⁴⁷ comparable to that of the monolith instead of just a proportional fraction of it. The same holds true for the mere start-up time. This is because the dependencies (libraries) of the monolith are mostly inherited by all microservices. However, build-time is significantly reduced mostly since just a proportional fraction of automated tests must be executed. It is just the application specific code with accompanying tests that is split into small amounts of code per microservice. Together with its dependencies, a microservice is not really that "micro".

Such memory footprint is [expensive particularly in cloud environments](#), where each gigabyte costs about 25\$ per year (in 2019) and vertical scaling large-scale systems quickly demand memory in the magnitude of terabytes.

With microservices, we also have an increase in [east-west traffic](#), i.e., network traffic within a data center (or even in the same cloud, depending on the definition) because of those additional messages that would be in-process calls in an equivalent monolith. Such traffic incurs additional costs also particularly in cloud environments.

Another discussion of up- and downsides of microservices can be found in [76] and about the distinction of splitting code bases vs. deployment artifacts in [77].

20.6 Redeployment in Distributed Systems

Redeployment of a monolith is uncritical since it cannot be redeployed in parts but just as a whole (at least except for a microkernel architecture). If the monolith is scaled, there are diverse redeployment strategies that can, e.g., avoid client-observable downtimes as we have seen in Section 18.7.

If there is a broken dependency within the monolith, this will normally surface before redeployment, i.e., during build-time. Redeployment of single programs in a distributed system however bears the risk of breaking a dependency with another program. There are basically two patterns that tackle this problem.

20.6.1 Release Trains

A release train is nothing else than redeploying all programs in a distributed system that are directly or indirectly affected by a change due to dependencies.

⁴⁷ Right after start, the size text section containing the executable code (cf. Section 2.3.5) dominates over the still unpopulated heap and stack section for common processes. For JVM processes the same holds true for the meta space vs. the heap.

EXAMPLE 78: RELEASE TRAIN DEPLOYMENT EXAMPLE

Figure 69 exemplifies a release train. Program 3 is redeployed, which breaks the dependencies from 2 and 4, so these programs must also be redeployed.

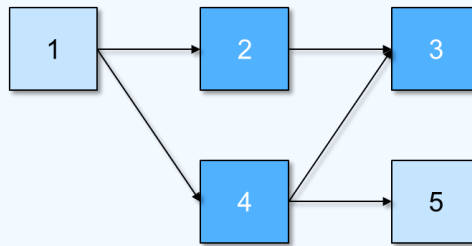


Figure 69: Release train example

The main drawback here is that the redeployment process is becoming more complex.

The release train pattern however is particularly not favorable in a microservice architecture, where the microservices are supposed to be deployed independently from each other. It becomes impracticable if the clients are external, from customers of your service.

20.6.2 Keeping Backward Compatibility

Another way is to avoid breaking dependencies by keeping them backward compatible. That is, if a server is redeployed, its API may change only in a way that existing clients still can access the server like before.

a) Message Compatibility

In a client-server system, the new version of a server should never require more information from a request message than in the previous version. Conversely, response messages of a new version should include all properties from previous versions.

EXAMPLE 79: MESSAGE COMPATIBLE DEPLOYMENT

Figure 70 depicts an example where 2 and 3 are redeployed. 2 may profit from new data in 3's evolved API but non-redeployed 4 can still use 3's API as it does not care about new properties in 3's responses.

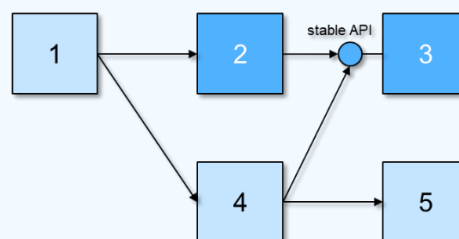


Figure 70: Message compatibility example

The main drawback here is that the **messages become more complex** insofar they tend to grow complicated as they contain more and more unused data.

b) Multi-Version APIs

Servers might expose an API in different versions. Clients must address a version in their request messages. In case of HTTP this might either be a certain segment in the request URI like `/v3/` or a certain attribute in the request header or body. Then, newer versions of an API do not need to keep message compatibility.

EXAMPLE 80: MULTI-VERSION API DEPLOYMENT

Figure 71 the same situation as in Figure 70 but 3 now exposes two APIs. 4 would not be able to use the new API version as it is not necessarily backwards compatible but can stick to the old API version.

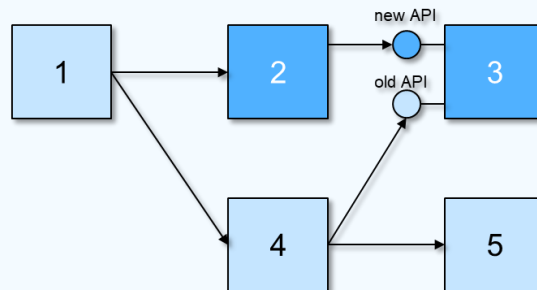


Figure 71: Multi-version APIs example

The main drawback of this pattern is that the **server becomes more complex** as with each redeployment more and more APIs must be implemented. This can be remedied to some extent if canary builds (cf. Section 18.7) are used for different deployments and their respective APIs. This, however, works in practice with just two versions. Before a particular program with two versions can therefore be redeployed again, dependent programs must be adapted and redeployed to the current API version of that program.

21 Communication

Communication patterns in non-distributed systems are simple. In a distributed world with communicating yet independent processes (nodes), there are more degrees of freedom but also complication.

21.1 Communication Directions

21.1.1 General Patterns

Figure 72 summarizes the most important patterns regarding directions of messages sent between two nodes A and B.

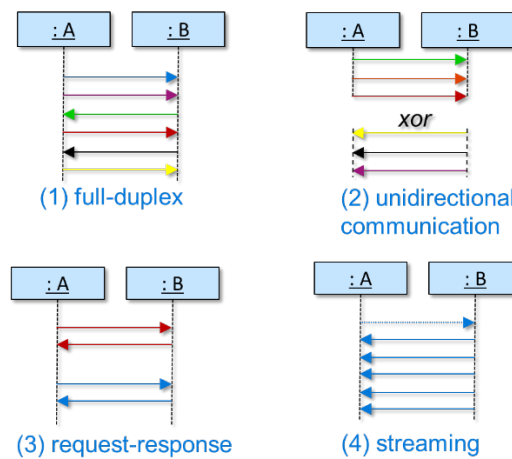


Figure 72: Communication directions

1. In the most general case, messages are sent **full duplex** between both nodes. Any node can send and receive messages at any time. There is no correlation between these messages.
2. In a **unidirectional** communication, messages just have one direction but are not necessarily related.
3. In a **request-response** pattern, we have pairs of correlated messages with opposite directions.
4. With **streaming**, there is a (indefinite) stream of correlated messages in one direction. This stream might be preceded by a single message in the opposite direction. In this case this single message often poses a request and the stream of messages the respective response.

EXAMPLE 81: COMMUNICATION DIRECTIONS OF TCP, WEBSOCKETS AND HTTP

The Transmission Control Protocol (TCP) and the WebSocket Protocol [78] are protocols for general network communication although on different layers of the ISO/OSI stack [11, p. 28].

The Hypertext Transfer Protocol (HTTP) as discussed in Section 27.1 is a request-response protocol.

A key abstraction of common protocol stacks is to correlate messages. That means that messages can be chronologically intertwined and multiplexed with other messages on the same communication channel in order to form a single act of communication.

Remote Procedure Calls (RPC) is a related abstraction and is usually provided by standard libraries of common programming languages. RPCs relieve programmers from (un)marshalling in-memory data, i.e., variable values, to and from messages and sending these messages to the right receiver.

Instead, RPCs create the illusion for the programmer of a local function call, which is actually a call between to remote processes.

21.1.2 Pull and Push

If node A sends a message to node B and that message contains a query and node B replies with a message containing an answer, we speak about **pull** communication. If node B proactively sends data in a message to node A, this is called **push** communication. The answer and the data might also be a stream of correlated messages instead of a single message like Figure 73 suggests.

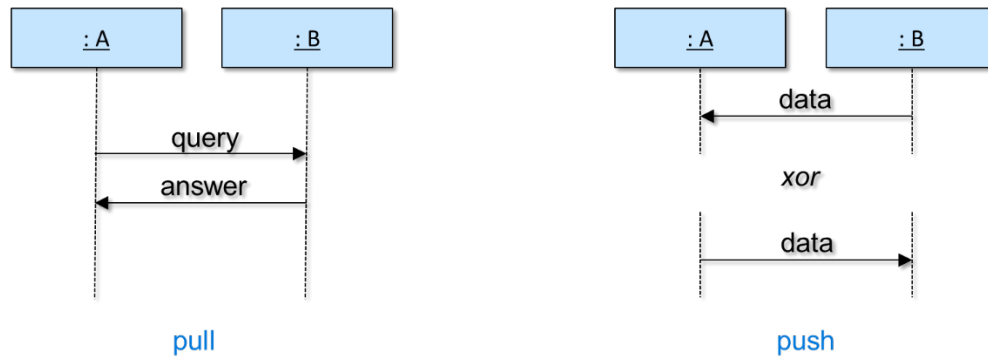


Figure 73: Pull and push

If both nodes form a client-server system, there are four combinations of pull and push from a client or server. Table 30 summarizes these.

	(Request) message originates from	(Request) Message contains	Description or example
Client Pull	client	query	example: HTTP GET request that queries some web resource from the server
Client Push	client	data	example: HTTP POST / PUT request that is used to upload a web resource to the server
Server Pull	server	query	- rarely used - example: server sends keep-alive queries to clients to determine which clients are still out there.
Server Push	server	data	- example: callback from server to client - example: notification about server-side data changes that a client is supposed to cache (cf. Section 8.5.3) - example: message in a long-running stream

Table 30: Pull and push in client-server systems

21.1.3 Communication Origin

By definition, a server is independent from its client(s). Consequently, a communication, which is a series of correlated messages, which starts on the server side, is hard to implement. A server typically does not know its clients, nor their network addresses by itself. One could make a client known to the server by adding its network address to a configuration record on the server side. This works for stable client-server systems, which might be two nodes in the intranet of a company. Besides, one could argue that this would violate the definition of a server of being independent of its clients, it leads to more concrete problems in the web:

a) Clients are hard to reach proactively

In the web, clients are often user agents like browsers that come and go, have dynamic addresses and are unreachable due lack of routes and/or due to firewalls. Therefore, network communication must be established by the client. In the simplest case, a client establishes a transport layer connection (TCP or UDP) with a server, which particularly reveals the client's (dynamic) IP address and source port, thereby registering a network **callback** address on the transport layer.

b) TCP connections are ephemeral

Transport layer protocol (mostly TCP) connections are ephemeral. They have to be reinitiated by a client if one of the communicating nodes dies or changes its address (IP address and source port) or the server runs out of capacity for concurrently opened TCP connections.

c) Request-response oriented use of HTTP

Moreover, even if the transport layer connection is assumed to be stable, many clients and servers just use HTTP in a strict request-response oriented fashion in which particularly the response message is finite and of predetermined length. In such cases, the only way to have something like server pushes here is **polling**, which is the client sends frequent requests for updates to the server. If such a request arrives but there is nothing to push on the server side, the server might defer the response. If a response is finally sent to the client or a deferral exceeds a certain timeout—normally 300 seconds in browsers—the client directly renews the request. We also speak of **long polling** in such cases, which is more like a workaround and comes along with diverse technical issues [79, pp. 4–5]. However, the adoption of HTTP-based technologies grows which go beyond this restricted communication pattern and allow for real server pushes. This is particularly true for browsers. These technologies are subject of Sections 27.2ff.

21.2 Communication States (Sessions)

Nodes which participate in a communication must keep track of the state of a communication to some degree. We call a data record that comprises the **state** of a specific communication between a client and a server-side a **session**. Such sessions are retained over the lifecycle of the whole communication (and not just over the processing time of a single message of a client).

The parts of the session that is stored on the client is called **client-side session** and that on the server **server-side sessions**.

21.2.1 Hierarchy of States

a) Protocol states

Sessions may comprise **protocol states** like connection states that contain technicalities like **client addresses** and the **connection state** (with states like CONNECTING, CONNECTED, CLOSED etc.). If a connection remains open, i.e., in a non-final state for many exchanged messages and a potentially indefinite time, we speak of a **persistent connection**; if it remains just for a single message or request-response exchange, it is a **transient connection**.

If a communication protocol does not rely on protocol states at all and lower-level protocols neither do, or just establish transient connections, then it is **stateless**, otherwise **stateful**.

EXAMPLE 82: STATELESSNESS OF COMMON PROTOCOLS

HTTP (cf. Section 27.1) is a prominent example of a stateless protocol. In principle, every single request-response communication between a certain client and a certain server may establish a new, transient TCP connection.

The File Transfer Protocol (FTP) is stateful since, firstly, each client-server pair communicate over two persistent TCP connections (control connection and data connection) and, secondly, there is an FTP connection state with states as described in [63, p. 70].

The WebSockets protocol (cf. Section 27.5) is stateful since each client-server pair exchanges WebSockets messages over a persistent TCP connection.

b) Application States

Using a stateless protocol does not render the entire communication stateless and sessions obsolete. Depending on the application in which the communication takes place, a session also contains [application states](#) like

- [navigational states](#), about, e.g., on which web page in a page flow the user is on or which flyouts have been opened or collapsed etc.
- [authentication states](#) that authenticate requests to originate from a certain user,
- as well as sometimes [preliminary transactional data](#), e.g., contents of a shopping cart.

21.2.2 Server-side Sessions

In horizontally scaled server tiers, a client cannot communicate with an arbitrary server if the communication is stateful. It can only communicate with a server that has access to the server-side session that contains the state.

EXAMPLE 83: ACCESSING SERVER-SIDE SESSIONS IN JAVA

Many HTTP programming frameworks provide means to associate a specific data record with HTTP requests that come from a specific client. In Java, one typically uses the Java Servlet API [80] to access the server-side session during the processing of a request like this

```
HttpSession httpSession = httpRequest.getSession();  
int requestCounter = (Integer)httpSession.getAttribute("counter");  
httpSession.setAttribute("counter", requestCounter + 1);
```

This quite simple example just stores the number of past number requests in the communication in the session as application state.

The `getSession()` method relies on a [cookie](#) [81], which is a key-value that is put in every HTTP request header that targets a certain server. By the cookie value, the server can identify the client and thus the server-side session which contains the current counter state.

A server without that access and without knowledge of prior messages—by which it could reconstruct the server-side state—may be incapable to understand and process a message from the client.

21.2.3 Sessions vs. Scalability

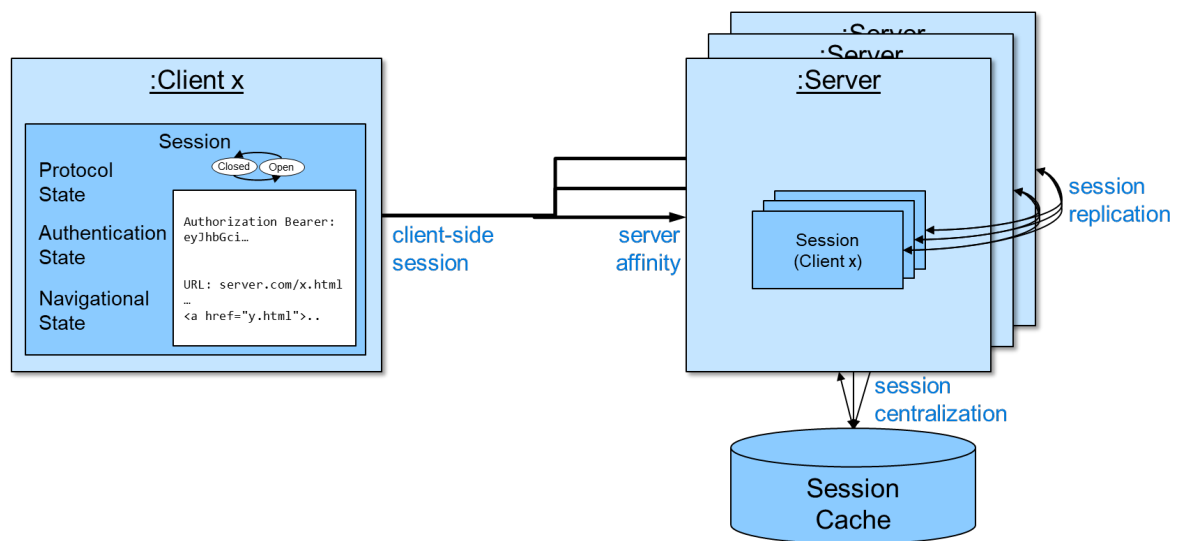


Figure 74: Sessions in outscaled tiers

There are different strategies to ensure that a client communicates with a server with access to the server-side session in an out-scaled tier. Each of these strategies has its impact on the tier's scalability:

- **Avoiding sessions** in the first place would solve the scalability problem but is often not feasible. However, even with an intrinsically stateless protocol like HTTP, application states often cannot be avoided: Even in the simple example above the current counter value is an application state that must be stored somehow.
- **Server affinity a.k.a. sticky sessions** let the load balancer route all messages of a certain client to the very same server (cf. Section 19.3.1b). The load balancer identifies incoming messages of the same communication by certain properties like the client's IP address and source port or a set cookie that contains a key for the server-side session. The drawback is obvious: If that particular server becomes unavailable, no other server can jump in to uphold the communication. Particularly in so-called serverless cloud service models (cf. Section 24.2.5), it is common that server processes are stopped and started after processing a single request or after a timeout without any incoming requests. Server affinity is not applicable here.

The following strategies just apply to application states. Protocol states cannot be replicated, centralized, nor entirely offloaded to the client.

- **Session replication** tries to replicate and thus synchronize all server-side sessions between all servers in a tier as quick as possible. However, session replication itself opposes scalability since the synchronization effort grows quadratically with each new server in the tier. Moreover, data replication has its own challenges (cf. Chapter 33).
- **Session centralization** means that all servers read and write server-side sessions from and to a central database called session store, which is often a simple key-value database like Redis. Because such reads and writes happen for every incoming message, such a database may become a bottleneck and thus counteract scalability.
- **Client-side sessions** means that sessions are stored in the client (browser) and included (in relevant parts) in every message to a server with the cost of higher transmission times.

21.2.4 Streaming and Full-Duplex Communications vs Scalability

In streaming or full-duplex communications the server proactively sends messages to a client. In an out-scaled tier, it is always a particular server that sends messages to a particular client. If such a server becomes unavailable another server must take over sending the messages to that client. In such cases, **message centralization** helps servers to obtain the messages that would have been sent by the failing server.

EXAMPLE 84: OUTSCALING A CHAT WEB APPLICATION

Consider a chat web application with an out-scaled server tier. Each client has a persistent full-duplex or streaming connection with a particular server to receive messages sent by other clients. The servers share all incoming messages via a message broker (cf. Section 26.1).

In case of a failure of some server A, the client (browser) must reconnect to a new server B. This server B then can send all messages to the client that have meanwhile been retained in the message broker.

The reconnection with the new server B must either be done by the origin client or by some intermediary load balancer, which then keeps the origin client agnostic of the failover.

21.3 Communication Contents

21.3.1 Meaning of Messages

A (request) message might convey a rather small **query** with selection criteria for data that node B can look up, e.g., in a database or a file system. A request could also contain **data** that is supposed to be stored or forwarded by node B. Alternatively, it is **computation input** for some kind of algorithm on node B, e.g., video re-encoding or prime factorization, whose output is placed in the response from B.

21.3.2 Encoding of Messages

Independent of their meaning, messages are either encoded in raw **binary** form or in a **textual** form by an encoding standard like UTF-8. The latter is one step towards larger yet more self-describing messages (cf. Section 28.3).

21.4 Synchronization

Exchanging messages in distributed systems is normally tightly connected to the execution state of involved processes.

21.4.1 Synchronous Communication

In a **synchronous** communication, a node A, i.e., a process or thread that acts as a client, pauses execution right after sending a request. Its state gets changed by the operating system into a waiting state and its program counter does not proceed any further, hence the term **blocking wait**.

21.4.2 Asynchronous Communication

In an **asynchronous** communication as depicted in the lower part of the UML sequence diagram in Figure 75, a node A sends a message but does not block, i.e., node A proceeds executing. If there is a response, it is processed by node B by a function that is independent of the function that sent the request in the first place.

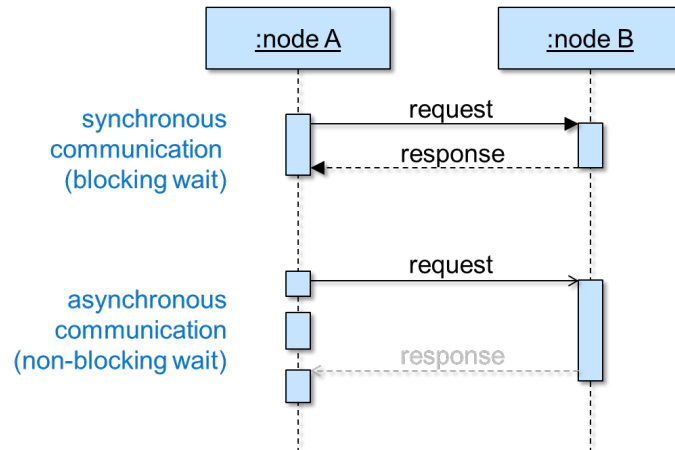


Figure 75: Synchronous and asynchronous communication

21.4.3 Implications for Programming

The best form of communication is often guided by requirements: If node A is a browser, the communication has to be synchronous since the user behind A is by himself in a blocking wait for the page to load. On the other hand, if, e.g., human intervention (picking something from the inventory, releasing a payment etc.) is required in order to fulfill a request, blocking waits are inappropriate thus asynchronous communication is better suited for this case.

However, technical trade-offs also play a role. Synchronous communication leads to blocking waits. This calls for

- multi-threaded processes in which some threads may be in a blocking-wait state, and others can still serve requests or
- asynchronous programming, i.e., where function calls never cause a blocking wait but require the registration of a call-back handler function that deals with the return value (response) of that function.

a) Multi-Threading

Creating processes and threads incur costs:

- Creating processes and threads (even by cloning) is slow (cf. Table 12)
- Each new process and thread consumes private memory (cf. Example 16:)
- Switching between existing processes and threads is also slow (cf. Table 12)

If a server process would create a new thread or even a new process for every incoming request would quickly become slow and run out of memory. Therefore, in server processes, it is particularly reasonable to use **thread pools** that are limited to some hundred threads. Once completely filled, new requests are handled by already created threads that wait idle in that pool.

However, if all threads from a thread pool are busy, i.e., in a running or blocking-wait state, new requests cannot be served. This can give rise to **fault cascades**.

EXAMPLE 85: FAULT CASCADE

Consider a cascade of dependent nodes $A \rightarrow B \rightarrow C$ where A sends requests to B which sends requests to C : If C is not responsive (for whatever reason), B might quickly run out of available threads which could handle request from A and thus also become unresponsive from the viewpoint of A .

There are several mitigation strategies that can contain fault cascades:

- A blocking wait should be cancelled by a client after a certain **timeout**. For example, browsers typically have timeouts of one to five minutes if they make http requests.
- Clients can direct requests to **proxies** which handle faults for them (cf. Section 21.6). Such proxies need to be local to the client, i.e., in the same process (as part of the same program) or at least on the same machine such that the connection between the client *A* and the proxy is reliable.

b) Asynchronous Programming

In asynchronous programming, function calls are never blocking. Instead, a callback handler function is registered during the function call. So, the control flow may proceed instead of blocking. When the function call returns a value, the callback function can be invoked that processes the return value. Asynchronous programming requires only a single thread that continuously loops through a job queue, in which these callback functions are placed. Hence, the downsides of multithreading as described before can be avoided.

However, asynchronous programming is more burdensome for developers:

- **Control Flow Obfuscation:** In synchronous programming, a function call and the processing of its return value can easily be kept in a clear sequence. In asynchronous programming, the in principle sequential control flow is obfuscated by the use of callback functions.
- **Stack Trace Partitioning:** Synchronous programs are easier to debug and profile. One can set a breakpoint to a blocking function call and then step to the next step (where the return value is processed). The stack frame remains the same along with all the local variables that have been used in the function call in the first place. Conversely, in asynchronous programming, the callback function has a totally new call stack. In particular, the stack frame of the function that made the blocking function call is not accessible anymore.

EXAMPLE 86: MULTI-THREADING AND ASYNCHRONOUS PROGRAMMING IN JAVA

The Java program of Figure 76 makes four concurrent http requests to an external service and output the status code of the http response. Every request is handled by a dedicated Thread. The control flow of each thread Thread-0 to Thread-3 is stopped by a break point in line 28. The call stacks of all running threads are shown on the left side with the one of the first thread expanded. Obviously, the stack frame of `makeHttpRequest` is still accessible including the local variable `url`.

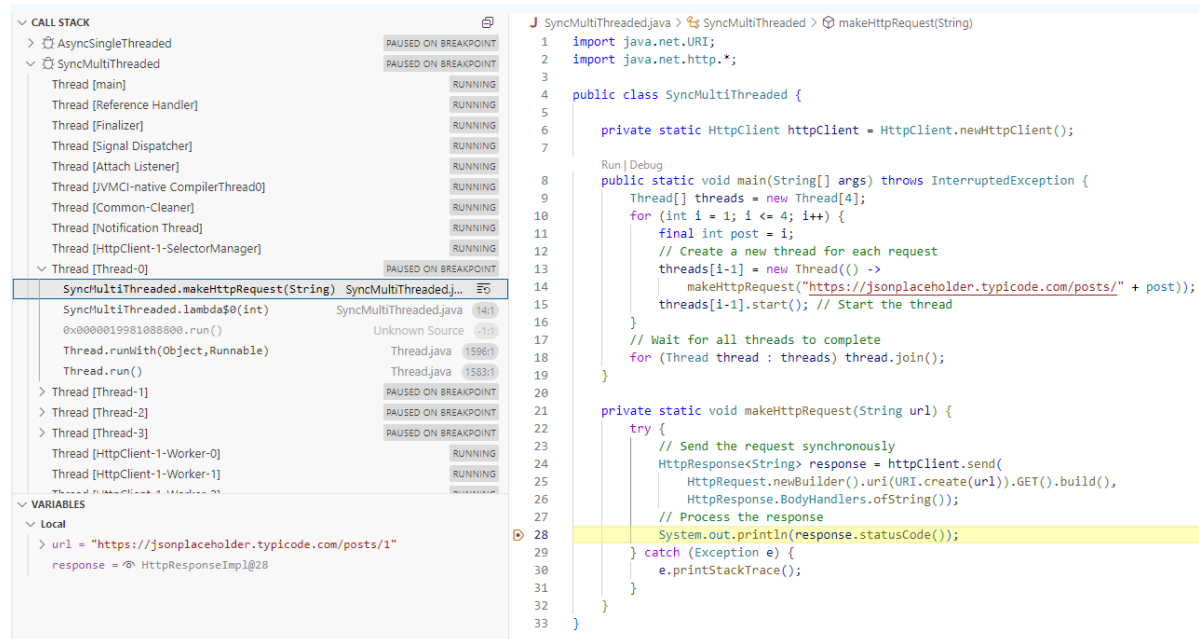


Figure 76: Synchronous, multi-threaded Java program

For comparison, the Java program of Figure 77 does the same but just with a single thread pool-1-thread-1 and asynchronous programming. Again, the break point is on the line that outputs the response status code. However, we are in a callback function registered as a lambda function in line 30. At this point, the execution of makeHttpRequest is already over and its stack frame is gone together with the value of url.

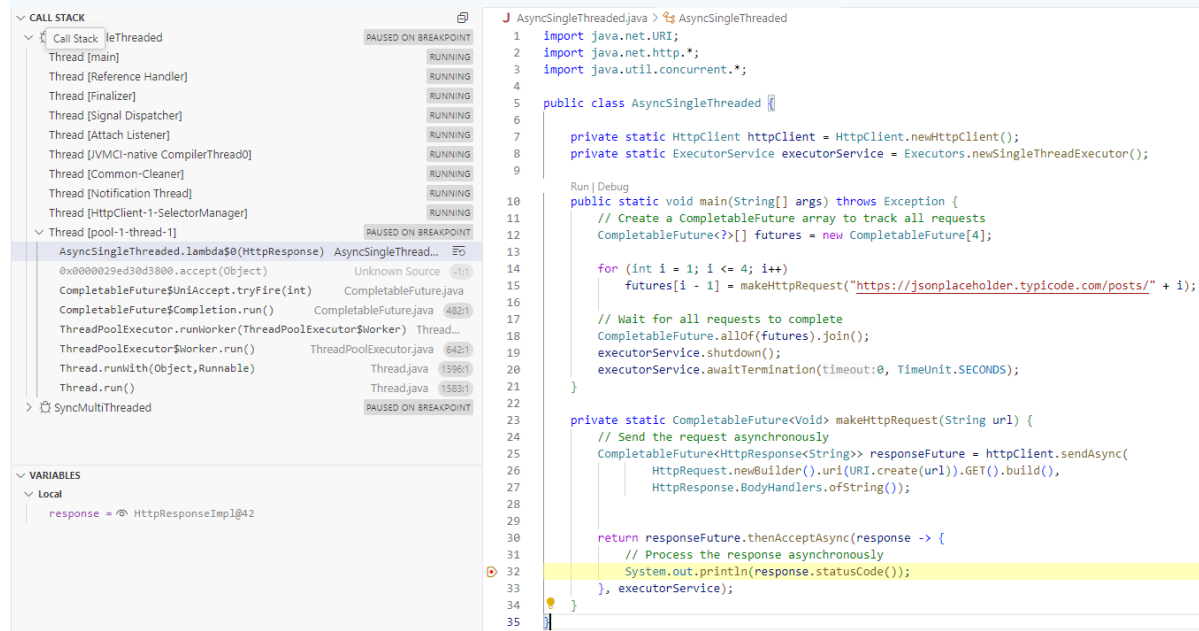


Figure 77: Asynchronous, single-threaded Java program

These examples are trivial. In typical programs callback function again do asynchronous calls and so do their registered callback functions and so on. Such programs can become difficult to debug due to stack trace partitioning.

c) Virtual Threads

So, multi-threading with synchronous programming is easier to program. To avoid the downsides of multi-threading with **platform threads**, i.e. wrappers for operating system threads, programming languages like Java provide the concept of **virtual threads**. From the viewpoint of a Java developer, virtual threads mostly work like platform threads with the advantage of synchronous programming. However, the Java Virtual Machine does not back a virtual threads an actual, dedicated operating system thread. Instead, it maps n virtual threads to a single operating system thread. Creating and switching between virtual threads is faster and memory consumption is lower as Table 31 shows. Developers can therefor treat virtual threads as an infinite resource; hence pooling virtual threads is neither necessary nor recommended.

	Platform Thread	Virtual Thread
Forking Time	200,000 ns	1,000 ns
Context Switching Time (direct costs in Java)⁴⁸	4,000 ns	3 ns
Allocated Private Memory	1,000 KB	1 KB

Table 31: Comparison between Platform Threads and Virtual Threads in Java

Other programming languages have comparable constructs to reconcile ease of programming and debugging with resource efficiency: Golang for example uses so-called **Goroutines** that are comparable to virtual threads in Java.

21.5 Messaging

Unidirectional messages between two nodes are the simplest form of messaging. **Message queues** add a layer of indirection between a sender and a receiver. Senders send messages not directly to receivers but to a message queue where they are (temporarily) stored and forwarded to receivers. This decouples senders and receivers with respect to (mutual) discovery, i.e., they can be kept agnostic of each other with respect to their network location.

Normally, messages are processed and forwarded in a "first in first out" (FIFO) manner, hence the name "queue". If the message becomes the oldest, the message queue dequeues and forwards it to a receiver. Normally, if the first attempt fails, the delivery is retried until success. This decouples senders and receivers in so far as temporary failures of senders or receivers do not lead to system failures but just to message queues that temporarily buffer more unforwarded messages.

⁴⁸ As measured by https://gitlab.com/rwoerzbe/public/-/raw/main/LCSS_Lecture_Notes_resources/JavaConcurrencyProgrammingModels/ContextSwitchBenchmarkStats.java

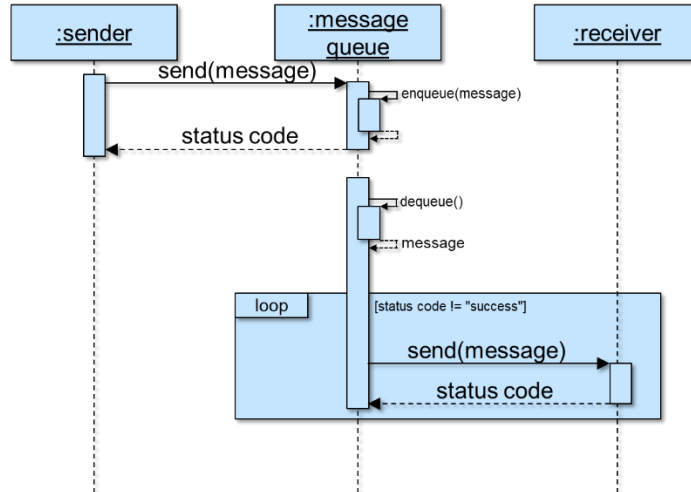


Figure 78: Sequence for forwarding a message with a message queue

21.6 Proxying

Like message queues, proxies also serve as intermediaries. However, they are intended for use in request-response or streaming-oriented communications. They can be used to implement cross-cutting concerns in the communication between two nodes:

- Proxies can act as **dedicated load balancer** (cf. Section 19.2.2)
- Proxies can be used to route messages based on application layer data (cf. top of Figure 79)
 - to expose different web applications under the **same domain** and
 - to implement **redployment strategies** (cf. Section 18.7).
- Proxies can be used as **caches** to reduce response times (cf. bottom of Figure 79 and Section 27.1.5)
- Proxies can act as **circuit breakers** to prevent or compensate oversaturation of a server and thus increase the resilience of a distributed system. For example, they count the so-far unanswered requests or measure the current ratio of unsuccessful to successful responses. If these numbers are kept below a certain threshold, requests are relayed to the server. Otherwise, they are intercepted and responded with either an error response or a (potentially stale) cached response as a fallback (cf. Section 8.5.3). In either way, the response time for the client and its blocking wait can be kept short.
- If every node is accompanied by a local proxy and the communication channel between the node and its proxy is therefore considered secure, proxies can take care of **secure communication**, i.e., encrypt and signing messages and mutual authentication on behalf of its node, e.g., with TLS (cf. Section 30.3).

Service Meshes as described in Section 25.5 use proxies in this way.

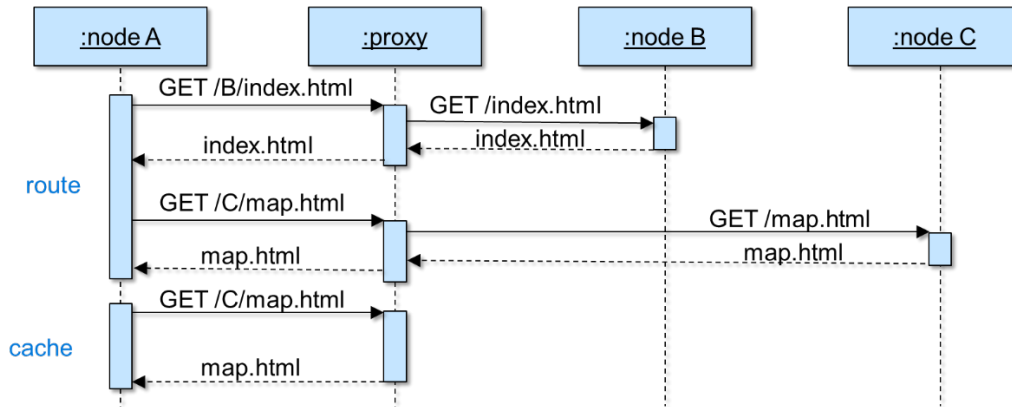


Figure 79: Proxying of HTTP messages

21.7 Flow Control

A distributed system consists of many communicating nodes. Each incoming message typically leads to a cascade of message exchanges between nodes.

Receivers can buffer short bursts of incoming messages in network buffers or message queues. However, if the average arrival rate as defined Section 5.3.1 exceeds the average service rate for too long, these certain nodes (or tiers) become overutilized. Overutilization leads to constant grow of wait queues and eventually to dropped messages since every wait queue is finite in size.

If senders are clients that communicate with the receivers (server) in a request-response style (cf. Section 21.1.1), growing wait queues lead to longer response times (cf. Section 5.4). Too long response times or entirely dropped messages lead to timeouts in client nodes and often to user-facing obscure error messages. So, preventing overutilization and (over-)full waiting queues are common goals in the design of large and distributed systems.

21.7.1 Limits for Scaling

Scaling as described in Chapter 18 is a common method to counter overutilization and thus long or even overfull waiting queues. However, scaling has its downsides:

- Scaling is **costly**, since the additional resources must be paid for in one way or another. This is true for vertical scaling (bigger servers are more expensive) as for horizontal scaling (more servers cost more).
- Scaling incurs **technical difficulties** like downtimes, lost persistent connections and the like.
- Scaling is intrinsically **inert**. Unexpected and sudden bursts of incoming messages therefore cannot be effectively compensated by scaling.

21.7.2 Flow Control Mechanisms

Flow control is another strategy that helps preventing (over-)full waiting queues. The following non-exhaustive table summarizes the most common ones.

Flow Control Mechanism	Category	Type	Context/ Usage	Key Features	Advantages	Disadvantages
TCP Congestion Control [82, Sec.	Back-pressure	Congestion Control	Network Congestion Control	Additive Increase + Multiplicative Decrease, Slow Start, Congestion		Can reduce throughput in stable networks

Flow Control Mechanism	Category	Type	Context/ Usage	Key Features	Advantages	Disadvantages
3.8.2]				Avoidance	Adapts to network conditions, avoids congestion	
TCP Sliding Window [82, Sec. 3.8.6]	Back-pressure	Window-based	TCP Connections	Dynamic window size, receiver-driven	Efficient data transfer, well-established	Can be inefficient under high latency
QUIC (HTTP/3) Congestion Control [83, p. 9.4]	Back-pressure	Congestion Control	Network Congestion Control	Advanced congestion control mechanisms	Better performance than TCP	Worse adoption than TCP
HTTP/2 [84, Sec. 5.2]& HTTP/3 (QUIC) [83, Ch. 4] Flow Control	Back-pressure	Window-based	HTTP messages	Stream and connection-level flow control	Prevents stream monopolization	Complexity in managing multiple streams
Adaptive Bitrate Streaming (ABR)	Back-pressure	Rate Adaptation	Video Streaming	Adjusts video quality based on network conditions	Smooth playback, reduces buffering	Can degrade video quality
Token Bucket Algorithm	Rate Limiting	Rate Limiting	Web APIs	Allows bursts, controls average rate	Flexible, handles burst traffic	Requires tuning of token rate/bucket size
Leaky Bucket Algorithm	Rate Limiting	Rate Limiting	Web APIs	Smooths out traffic, constant rate	Simple, predictable rate	Less flexible for burst traffic

Table 32: Flow control mechanisms

We can broadly distinguish between two categories of flow control mechanisms:

21.7.3 Rate Limiting

Rate limiting is a common function of inbound gateways. Blocking messages at this entry point spares other queues of the distributed system from growing or overflowing thereby affecting all clients.

Rate limiting sets senders (clients) a static, predefined maximum amount of incoming message for a certain time frame. Short bursts are therefore possible if the average message arrival rate is not exceeded. Feedback to senders is made only after a transgression, e.g., in form of HTTP responses with status code 429 – Too Many Requests.

For fairness reasons, receivers set limits and count messages not globally but per sender.

Rate limiting is particularly useful to block messages from senders that send erratically or maliciously as a denial-of-service attack at high rates. Erratic bursts typically occur if a sender buffers messages itself and sends them all of them at once to a web API.

EXAMPLE 87: ERRATIC BURST IN ReWo-RATE

A client of ReWo-Rate could for example buffer all ratings of a day and send them all at once at 5 p.m. to ReWo-Rate's web API.

Clients of web APIs are likely to be outscaled by themselves. Since all these clients are supposed to be treated as one w. r. t. the rate limit of the particular user, they are typically identified and authorized by a special **API key** that is usually included as an HTTP header.

a) Sliding Window

Sliding Window is an implementation for a rate limiting algorithm. It counts the requests in a window that "slides" along the time axis. If a set threshold is exceeded, incoming requests are blocked.

EXAMPLE 88: SLIDING WINDOW

Assume a sliding time window of 10 ticks and a maximum of incoming messages of 5. If one message arrives on time $t=0$, none on $t=1$, one at $t=2$ and so on as the following table describes, the limit is exceeded at $t=14$ because 6 messages are in the time window $t=5$ to $t=14$.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
x		x			x	x				x	x	x		x	

Note that this algorithm only counts the number of messages but does not take message sizes into account. If the message size strongly correlates with the service time to process the message, algorithms like Token Bucket are more suitable.

b) Token Bucket

Another rate limit algorithm is called **Token Bucket**. Every client is assigned a bucket that initially and at most has b tokens.

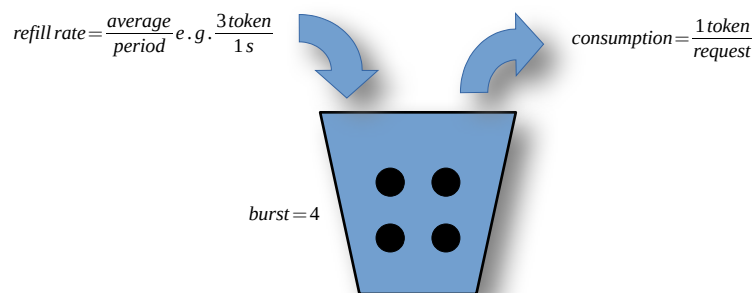


Figure 80: Token Bucket

Every requests consumes one token. (One can make the number of tokens dependent on the requests, e.g., its size, but we will assume that only one token is consumed per request in the following.) Token Bucket allows clients to consume all token at once in a burst. However, the bucket is filled again by a finite rate r , which is defined by time period p and average a as $r = p/a$.

EXAMPLE 89: TOKEN BUCKET

If we have a period of 10 seconds and an average of 20 tokens per period, then

$$r = \frac{20 \text{ tokens}}{10 \text{ s}} = 2 \frac{\text{tokens}}{\text{s}}$$

i.e. every second two tokens are added, i.e., every half second, one token is added to the bucket.

Usual implementations actually do not increase a bucket level counter every $1/r$. Instead, timestamps of requests are logged. Only when a request arrives, the current level counter of the bucket is computed and updated if the request is allowed to pass through.

21.7.4 Backpressure

Backpressure keeps senders (clients) apprised of how close they are to being (temporarily) blocked or for how long they are blocked. Therefore, senders can throttle their request rate soon enough to avoid a blockage in the first place or spare themselves futile retries.

EXAMPLE 90: BACKPRESSURE IN THE TRANSPORT CONTROL PROTOCOL (TCP)

In TCP, senders can put a window size in each sent packet. If a node A sets a window size in a packet sent to node B, this window size tells node B how much TCP packets it can send to node A before B has to wait for an acknowledgement (TCP ACK).

EXAMPLE 91: BACKPRESSURE IN THE HYPERTEXT TRANSPORT PROTOCOL (HTTP)

In a HTTP communication, the server may respond with a HTTP status code 429 – Too many requests but include a Retry-After: <Timestamp> in the HTTP header (cf. [85, Sec. 10.2.3]). The timestamp can be dynamically chosen. The request is then effectively buffered on the client side which may resend the request when the Timestamp is reached.

Server may also include RateLimit Header fields in HTTP responses before a limit transgression takes place. However, such HTTP header fields are still under ongoing standardization [86]. API Gateways like Kong can be configured to limit requests for multiple periods (seconds, minutes, days, months) per client [87]. Response headers typically look like this:

```
HTTP/2 200
content-type: application/json
content-length: 43
x-ratelimit-limit-minute: 1000
x-ratelimit-limit-hour: 10000
x-ratelimit-limit-day: 50002
x-ratelimit-remaining-minute: 999
x-ratelimit-remaining-hour: 9999
x-ratelimit-remaining-day: 50001
ratelimit-limit: 1000
ratelimit-remaining: 999
ratelimit-reset: 1
date: Mon, 20 Oct 2025 10:13:58 GMT
server: unicorn
via: kong/3.6.1
```

By using multiple time periods (minute, hour, and day in this case), a server can prevent bursty requests from its clients but still provide generous limits over longer periods.

Of course, backpressure requires that receivers can signal rate changes back to the senders like in the headers of exchanged packets or messages.

EXAMPLE 92: SERVER-SENT EVENTS VS. BACKPRESSURE

Server-sent Events (cf. Section 27.3) is a streaming communication protocol (cf. Section 21.1.1). Once a stream is established, a receiving client has no chance to signal backpressure to the sending server except with a complete separate HTTP request to a separate endpoint.

22 Caching

Caching is a common technique for reducing response times of read requests. Therefore, they are ubiquitous in distributed systems. Caches are subject for design decisions like caching policies that are context dependent.

22.1 Caches in General

A **cache** is a middleware resource or a hardware component that retains replicas of data records from an original source, such as other nodes or hardware components. These cached data records, to which we refer as **cache entries** or **cache lines**, can be served by the cache instead of the original data records from the original source. In comparison to the original source, a cache typically has a shorter response time because of

- a **lower service time**, since cache entries only must be looked up but not computed,
- a **lower (network) latency**, because caches are local or at least much closer to requesting clients,
- a **higher (network) throughput** for the same reason.

A cache typically stores and retrieves cache entries in its own storage, which is typically low-latency memory that might be backed by persistent storage. A cache typically does not contain a full replica of all data records the original source could respond with. A full replica is even infeasible if the original source is a process that produces arbitrary many different responses but also impractical if the original source is some kind of large storage with finitely many data records.

Writing access to data from a client is considered incomplete unless the original source has been updated. Thus, a cache is always a clear follower and the original source a leader in the sense of Section 34.2.1.

22.2 Read Caches and Write Caches (Buffers)

Caches are often pure **read caches**, i.e., clients do not write to them. A **write cache**, better called a **buffer**, stores either data that cannot be processed, written to disk or transferred via networks immediately. Deferrals may occur because the assigned processing unit is occupied, or because the storage or network interface is currently in use. Additionally, deferrals can result from underlying I/O or network protocols that transfer data in larger blocks, pages, frames, or packets.

Since often caches also serve as buffers, both terms are frequently used interchangeably.

22.3 Use of Caching

Caches can be found on many layers in computing. In the following, we discuss some examples.

22.3.1 Caches for Hardware Components

a) CPU Caches

CPU caches mitigate the so-called von Neumann bottleneck [88, p. 615], which is a fundamental limit imposed by the separation of processing units and memory in von Neumann computer architectures, where the processing units act as internal clients and the memory as internal server. Modern central processing units and graphics processing units are equipped with up to three layers of caches as part of the so-called **memory hierarchy**.

Table 33 depicts a typical memory hierarchy with processor registers at the top and tape store at the bottom of the hierarchy. Performance related properties are better (lower latency, higher

throughput) at the top of the hierarchy as we have already seen in Table 12, but capacity, price per byte, density (and thus size for a given capacity) and durability become better in the lower parts of the memory hierarchy.

Hierarchy component	Latency	Throughput ⁴⁹	Capacity	Price / Byte (assumed 8-core CPU with price of 500\$)	Density	Durability
processor registers	0.3 ns (1 cycle)	n/a	2 KiB (128 bytes for general purpose registers + ~2 KiB for special purpose registers per core=	32\$ / KiB	very low (SRAM)	low (power on)
L1 cache	1 ns (4 cycles)	1000 GiB/s	32 KiB per core	2\$ / KiB	low (SRAM)	low (power on)
L2 cache	4 ns (10 cycles)	1000 GiB/s	258 KiB per core	0.25\$ / KiB	low (SRAM)	low (power on)
L3 cache	14 ns (35 cycles)	400 GiB/s	8 MiB per CPU	0.04\$ / KiB	low (SRAM)	low (power on)
Dynamic Random Access Memory (DRAM)	100 ns	100 GiB/s	4 – 1.000 GiB	5\$ / GiB	Medium (DRAM)	low (power on)
Non-Volatile Dual In-line Memory with NAND (NVDIMM-N)	100 ns	20 GiB/s	32 GiB	50\$ / GiB	Medium	~10 years (like SSD)
Solid State Drives (Flash)	150 μ s (random)	1 GiB/s	2.000 GiB	0.005\$ / GiB	High	Usage depended (limited writes) ~10 years
Hard Disk Drives (magnetic)	8 ms (seek)	0.5 GiB/s (linear)	4.000 GiB	0.025\$ / GiB	High	Usage depended ~5 years
Tape	100 s	0.3 GiB/s	500 TiB	0.002\$ / GiB	Very high	30 years

Table 33: Memory Hierarchy (after [28], [29], [30], [31, p. 472], numbers may vary and represent just orders of magnitude)

⁴⁹ Throughput means the throughput from the respective layer (row) to some upper layer. In case of processor registers, there is no upper layer.

CPU caches range in the upper part. In comparison with random access memory, these caches reduce the latency of looking up data in memory by a divisor ranging from seven (L3 cache) to 100 (L1 cache).

EXAMPLE 93: CPU CACHE EFFECTS

The following example is inspired by [89] and shows surprising effects of CPU caches. Consider the following C program:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <limits.h>
#define ARRAY_SIZE 64*1024*1024

void loop1(int* arr) { for (int i = 0; i < ARRAY_SIZE; i += 1) arr[i] *= 3472; }
void loop2(int* arr) { for (int i = 0; i < ARRAY_SIZE; i += 16) arr[i] *= 3472; }

int main() {
    int *arr = (int*)malloc(ARRAY_SIZE * sizeof(int));

    // Initialize array such that arr[i] *= 3472 itself cannot be optimized
    srand(time(NULL));
    for (int i = 0; i < ARRAY_SIZE; i++) arr[i] = rand() % INT_MAX;

    // Loop 1
    clock_t start = clock(); loop1(arr); clock_t end = clock();
    printf("Time elapsed in loop 1 is: %f\n",
        ((double)(end - start)) / CLOCKS_PER_SEC);

    // Loop 2
    start = clock(); loop2(arr); end = clock();
    printf("Time elapsed in loop 2 is: %f\n",
        ((double)(end - start)) / CLOCKS_PER_SEC);

    free(arr);
    return 0;
}
```

On Linux, one can compile and run this program like so

```
curl -o loops.c
https://gitlab.com/rwoerzbe/public/-/raw/main/LCSS_Lecture_Notes_resources/
c_caching/loops.c
gcc -O1 loops.c -o loops_gcc
./loops_gcc
```

(Please use compiler optimization -O1 since -O0 produces unnecessary inefficient code for the loops and -O2 and -O3 generate machine code using vector operations for loop1 but not for loop2.)

At first glance, one would expect loop2 to run 16 times faster since it takes 16 times less iterations. In fact, loop2 is only twice as fast as the output shows:

```
Time elapsed in loop 1 is: 0.043323
```

```
Time elapsed in loop 2 is: 0.019697
```

This is because in loop1, a new cache line with a typical size of 64 bytes (= 16 integers each with size of 32 bits), must be transferred from either memory, L3 or L2 cache to the L1 cache only in every 16th iteration but in loop2 in every iteration. These memory transfers take many more clock cycles than the body of the loop.

b) Disk Caches / Buffers

A disk cache stores data that is repeatedly read by processes, thus is also called a read cache. Disk caches also act as buffers, storing data that is about to be written on disks (write cache). Disk caches either reside directly on the disk as hardware caches or are allocated by the operating system in the main memory as soft caches.

Caches for hardware components are mostly opaque to user space processes and developers, except for write caches which can be flushed to disk or network by means of certain system calls.

22.3.2 Network Caches

a) DNS Caches

The whole Domain Name System (DNS) makes extensive use of caching. DNS servers that are not authoritative for certain domains are still able to cache domain records. Even the operating system acts as a nameserver in this regard and locally caches domains records according to their time-to-live (TTL) entry. This drastically reduces the page load time of most webpages, which typically make tens or hundreds requests to the same domain.

b) HTTP Web Caching

Caching is central to the protocol of the world wide web HTTP as discussed in Section 27.1.5.

22.3.3 Materialized Views in Databases

Materialized views in relational databases are views that store data which is derived from other views or tables. When clients access a materialized view, this derivation, e.g., projection or joins, has already been done so processing times and thus responses times are shorter in comparison to executing a complex SQL query that derives the requested data on demand.

On the flip side, a materialized view may contain stale data if there is more recent data that resides in the tables on which a materialized view depends but that has not been propagated to the materialized view yet.

22.3.4 In-Process Caches

In programs, private caches may store and retrieve responses from databases or web APIs. Such caches are easy to establish using data structures like hash maps. However, implementing the access to such private caches and a proper eviction policy is a tedious task. Implementing them as shared caches is even more difficult.

Middleware like Memcached [90] can help deal with these tasks.

22.4 Caching Goals

Whenever caches are used, there are different opposing goals:

- **Privacy:** Cache entries may contain confidential data and thus must be kept private.
- **Efficiency:**
 - **Storage size:** Bigger caches cause bigger costs for bigger storage and/or memory. Thus, caches are small compared to their original source and cannot (by far) replicate all the data. Cache entries may therefore have to make room for other ones if the cache reaches its capacity limits.
 - **Computational simplicity:** Keeping a cache and particularly its eviction policy simple reduces demand for computational resources, which is crucial in certain contexts.
- **Recency / staleness** of cache entries: Caches are usually not notified about changes of data in the original source. If a data record is updated and thus more recent than a corresponding cache entry, the latter one is out of date and thus called to be stale.
- **Response Times**
 - **Cache hit/miss ratio:** If a cache serves a cache entry as response, we speak of a **cache hit**. If it has not valid cache entry and must therefore relay the request, we speak of a **cache miss**. This message relay of course incurs additional response time. A high hit/miss ratio is therefore desirable as every hit reduces the processing time to a mere lookup. In such cases, we also speak of a **warm cache**. Note that a new and therefore empty cache is cold in that sense.
 - **Client proximity:** Caches may differ in their response time from the viewpoint of a single client in case of a cache hit. Pure latency is generally lower if the cache is close to the respective client.
- **Opacity:** Clients should be kept agnostic of whether they communicate directly with the original source or with a cache that proxies the original source.

In the following sections, we describe how to deal with these challenges.

22.5 Private and Shared Caches

Caches may serve data exclusively for a single client and thus contribute to the goal of privacy.

Private caches are usually (but not necessary) near the client, e.g., integrated into the same hardware component or at least residing on the same machine, which reduces latency. Shared caches are remote and thus incur higher latencies but can justify bigger capacities and are likely to have a higher hit/miss ratio.

22.6 Population Policies

Population policies describe when and how caches are populated with entries in the first place. One can classify these policies by the direction of the control flow: Either a cache entry is populated during a write to the corresponding data record or a read from it.

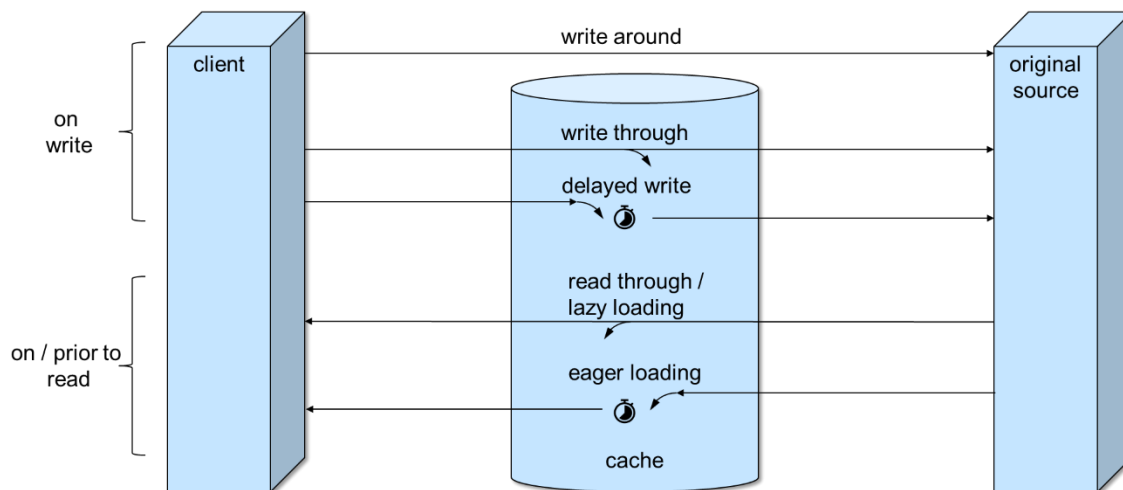


Figure 81: Cache population policies

22.6.1 Cache Population on Writes

- **Write around** means that a write does not populate the cache at all, so it goes “around” the cache. The population is delegated to a read policy.
- **Write through** means that a data record is simultaneously put into cache but also written to the original source.
- **Delayed write** means that the cache acts as a buffer and delays the updates in the original source.

22.6.2 Cache Population on or Prior to Reads

- **Read-through** or **lazy loading** means that every cache entry is added to the cache at the first request to it. So, the cache is initially empty, and every first request leads to a cache miss.
- **Eager loading** means that entries are added to the cache even before they were requested for the first time by a client. This is useful if the data records and thus cache entries are dependent and thus likely to be requested shortly one after another. HTTP server push provides an example for eager loading.

In comparison, lazy loading focuses on the goal of smaller cache sizes and eager loading on shorter response times. If a cache is shared, the distinction between both types gets less strict: If two clients A and B request the same data record after another, A may have a cache miss but from the viewpoint of client B, the cache entry has been eagerly loaded.

22.7 Eviction Policies

Eviction policies are strategies for choosing cache entries that should be evicted, i.e., deallocated from a cache or replaced by other ones. These policies are meant to balance the aforementioned goals. The following are simple but common strategies:

- **First-In-First-Out (FIFO)** is a simple policy where the cache behaves like a queue and the ordering of eviction of cache entries is the same as the ordering of population.
- **Least Recently Used (LRU)** is a policy where each cache entry has a timestamp of the last access to it. The entry with the oldest (lowest) timestamp is evicted first.
- **Least Frequently Used (LFU)** is a policy where each cache entry has an access counter. The entry with the least frequent accesses is evicted first.

- Segmented LRU is an extension of LRU the above mentioned where cache is divided into a probation and protected segment. In case of a cache miss, the cache adds the new cache entry to the most recently used end of the probation segment. If a client requests this entry again, we have a cache hit, and the element is promoted to the most recently used end of the protected segment. If an entry in the protected segment is not requested any more, it will eventually be demoted to the most recently used end of the probation segment. The protected segment is usually constant in size while memory from the probation element may be reclaimed in which case a whole tail of least recently used entries is evicted.
- Random Replacement is a policy where cache entries selected randomly for eviction.
- Time-To-Live (TTL) is a policy where cache entries are evicted a certain time after they have been put into the cache. Time-To-Live focuses on recency of cache entries and is used, e.g., in the Domain Naming System (cf. Section 23.8)

EXAMPLE 94: SEGMENTED LAST RECENTLY USED

Figure 82 shows an example for Segmented LRU. Here, at time 30, item G is hit and therefore promoted to the MRU top of the protected segment. Conversely, item D has to yield from the LRU bottom of protected segment and is being demoted to the MRU top of the probation segment. At some other time, the probation segment is shrunk, and the least recently used items J and K are evicted.

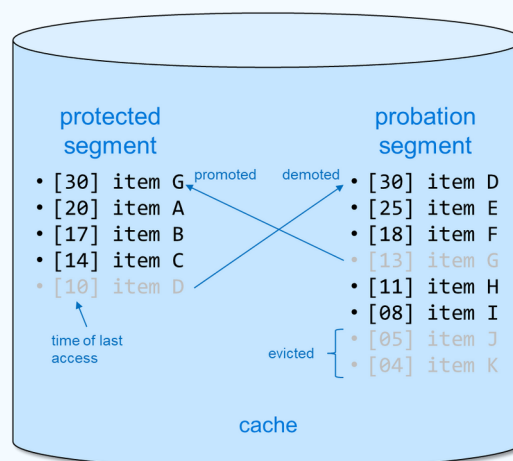


Figure 82: Example of Segmented LRU

FIFO, (S)LRU, LFU and Random Replacement try to balance the goals of efficiency and high hit/miss ratios. For example, Random Replacement's strength is that no additional data like access counts or timestamps need to be stored along the cached data.

Conversely, TTL balances recency with hit/miss ratios. Of course, a policy like TTL can be combined with the aforementioned, like in Time Aware Least Recently Used (TLRU) [91].

High hit/miss ratios are not guaranteed by any of these strategies but actually depend on the access patterns of the clients. If popular cache entries with many accesses remain popular, LFU is a good strategy. If cache entries are typically popular just for some time, LRU is a better strategy. In essence, when choosing an eviction policy, one has to make some assumptions and predictions about these client access patterns. This is why there are many more eviction policies than described in this section.

22.8 Bloom Filters

Bloom Filters are hash tables that map addresses to resources (e.g. URLs) to boolean values that represent either “definite miss” or a “likely hit”. Bloom Filters have a static size and are usually small enough to fit in memory thus have very short response times. Typically, they are associated with a server (cache, database etc.) that serves lots of resources with have to be generated or retrieved from with a comparable long response time.

If a client requests a resource form a certain address, the client or the server can first do a lookup in the associated Bloom Filter. If that resolves to a “definite miss” that miss can directly responded to the client. In case of a “likely hit” according to the Bloom Filter, the origin server is requested thereby accepting a comparable long response time. Since Bloom Filters have a static size, hash collisions cannot be out-ruled. This is why a positive response can only be considered a likely but not definitive hit.

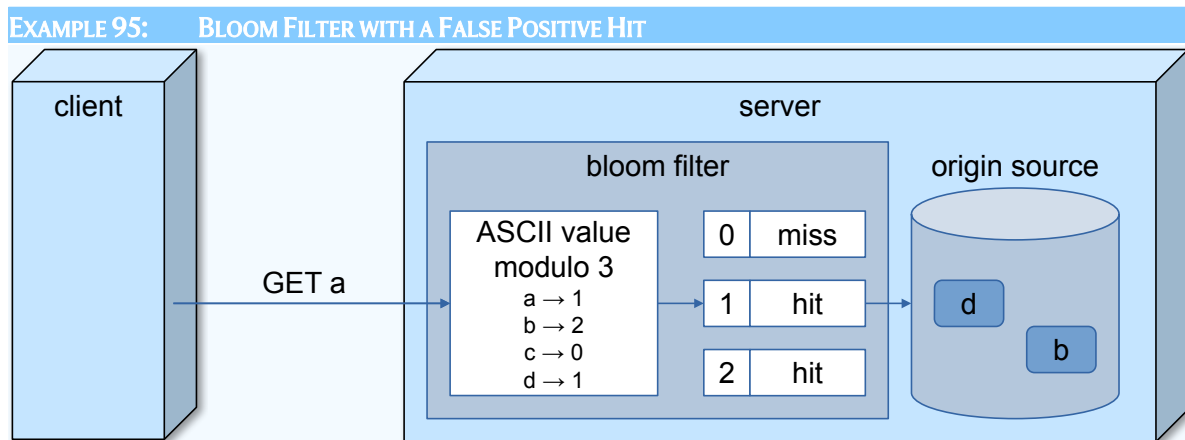


Figure 83: Bloom Filter with a False Positive Hit

In Figure 83 we see an origin source with resources that are addressed by single lowercase letters. The bloom filter uses a very simple hash function that takes the ASCII value of the letter modulo 3. For example, the letter “a” has an ASCII value of 97, thus is mapped to $97 \bmod 3 = 1$. Unfortunately, the “hit” is a false positive that stems from the prior storage of “d” which also maps to $100 \bmod 3 = 1$.

III. Infrastructure

23 Middleware

Today's system clusters consist of many different systems, which are often provisioned using a cloud offering like the Google Cloud (cf. Section 24.8).

Figure 84 depicts an architecture view (using a non-standard, non-formal notation) of a typical cloud-based system that realizes a web application. The figure shows typical tiers and layers with their respective technical responsibilities. For brevity, it abstracts from data and control flows between the tiers and layers as well as from horizontal scaling within tiers.

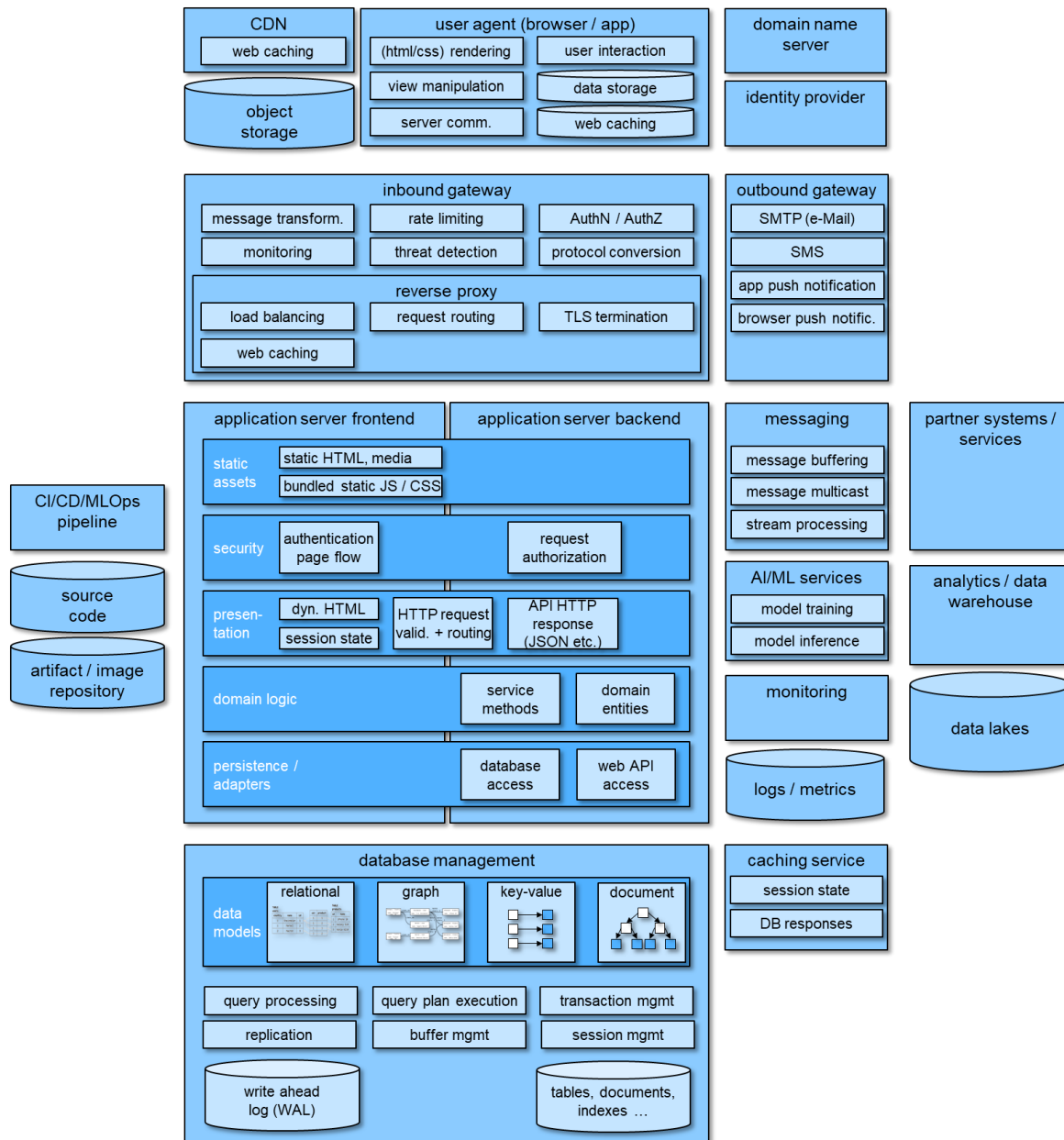


Figure 84: Typical web-based system architecture

23.1 Application Server

At the very heart of the system resides an **application server**. Typically, the application server receives HTTP requests and computes HTTP responses. These responses typically contain

dynamically generated [web resources](#), often represented as HTML or JSON documents. Additionally, the application server may be the original source of static web resources, like static HTML, picture, or video resources.

It is common to separate an application's [frontend](#) from its [backend](#) into different tiers. Table 34 clarifies the responsibilities of these tiers depending on different rendering strategies.

Rendering	Responsibility of Frontend	Responsibility of Backend	Effect of Hypermedia (clicks) on Client side
Static server-side rendering	Answer all requests as-is by lookups into filesystem or memory.	none	Request for HTML + full page reload
Dynamic server-side rendering	Parse HTTP request and forward to domain logic layer (in the backend). Generate HTML in HTTP responses. Keep track of the navigational state (cf. Section 28.2). Serve static resource as origin server (cf. Section 23.9) with appropriate cache headers (cf. Section 27.1.5).	Parse HTTP requests forward to domain logic layer. Implement domain logic. Access database.	Request for HTML + full page reload
Client-side rendering	Answer requests for static resources as-is (Nginx, Apache HTTPD) or after transpilation of TypeScript [92] or Sass, minifying and/or bundling them (NodeJS [93]). In single-page application these static resources are HTML frames (HTML with an almost empty body), CSS and big JavaScripts and JavaScript libraries like ReactJS [94] that do the actual rendering in the browser.	Like in dynamic server-side rendering but called from client, not from frontend.	Incremental update of client-side DOM optionally including requests to web API

Table 34: Typical forms of rendering

23.2 Client

A whole system cluster might act as a server for a client. One might distinguish between different types of clients.

23.2.1 Browsers

[Browsers](#) are generic user agents that can be clients for many servers. One can think of a browser as mere runtime environment that “runs” specific clients, which are made of HTML, CSS and JavaScripts that are served by the server. So, it is completely up to developers on the server side to know what a specific web application looks like in a browser regarding layout, available actions (buttons) etc.

23.2.2 Specific User Agents

A [specific user agent](#) is more tightly coupled with a specific server. A [native smartphone application](#) typically communicates only with a specific server in order to obtain or share (raw) data. The same is true for clients “in the middle”, which have no UI for end-users but just are themselves servers for other clients.

Within the server cluster, usually it is the application server that mainly acts as the main client's counterpart. Such clients send requests to the application server, e.g., by GETting a certain web page or by POSTing form data and renders respective responses.

23.3 Inbound Gateway

An **inbound gateway** is an umbrella term for the following system resources that deal with routing and distributing ingress (incoming) traffic. These are sometimes implemented by a single middleware program or sometimes by multiple programs that run in different processes (maybe on different machines) which are arranged in a network cascade.

Inbound gateways are oftentimes exposed to the internet as opposed to, e.g., application servers. In this case, inbound gateways care for terminating **transport layer security (TLS)** encryption and particularly for **HTTPS** requests.

23.3.1 Message Routing

Proxies (cf. Section 21.6) can be distinguished into two types: **Forward Proxies** are local or close to a sender (often client). **Reverse proxies** are typically close to a receiver (often server) and are particularly meant to route traffic to other nodes like load balancers or application servers, where it can be further forwarded or processed.

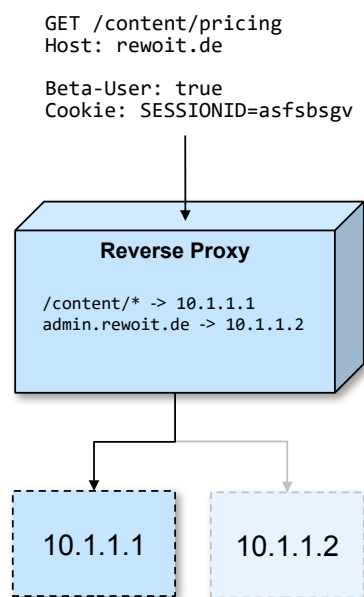


Figure 85: Reverse proxy

Oftentimes, reverse proxies analyze the application layer, e.g., HTTP. Typically, the following attributes in a HTTP request header are used to determine the destinations:

- **Request Line:** Every HTTP request begins with a request line like `GET /a/foo.html HTTP/3`. As in the example of Figure 121, the reverse proxy could use the path segment `/a/` to determine the receiver. This can be used to integrate different web applications each served by a dedicated system cluster within a single web site, e.g., behind a single domain name. By doing so, the user experiences a single cohesive web user interface (UI). For example, `www.dhl.de` works this way.
- **Host header field:** The `Host` HTTP request header field [85, Sec. 7.2] could include a subdomain that indicates the receiver. With respect to the example, if there is a wildcard DNS entry like `*.rewoit.de` then every fully qualified domain name ending with `rewoit.de` would map to a

single IP address, e.g., that of a reverse proxy. Still, the a particular fully qualified domain name like `a.rewoit.de` is set in the Host header field and can be used by the reverse proxy for routing.

- **Custom header attribute:** A custom header attribute like `X-Beta-User: true` could indicate that the request should be forwarded to nodes with a experimental version of a service during a Canary Deployment (cf. Section 18.7)
- **Cookie:** Cookie as described in [79] is a special HTTP header attribute. Its value is a semicolon-separated list of key-value pairs. Each key-value-pair constitute a cookie. In contrast to other headers, once set by a server (by a Set-Cookie header attribute in a HTTP response), a cookie is placed by the client in every request header that complies with the constraints specified in the Set-Cookie attribute, e.g., aims at a certain domain name and has a certain path prefix. Note that load balancers use cookies as well to route requests to the same server node for a particular client in case of sticky sessions (cf. Section 19.3.1).

23.3.2 Message Transformation

Inbound gateways can transform request and responses. A reason for such transformations could be to provide a consistent developer experience.

EXAMPLE 96: MESSAGE TRANSFORMATION IN INBOUND GATEWAYS

If the system identified by `a.rewoit.de` exposes a RESTful web API in the sense of Chapter 28 but `b.rewoit.de` only supports the older standard Simple Object Access Protocol (SOAP), an inbound gateway could translate RESTful HTTP messages to SOAP messages thus providing a consistent experience to client developers.

Inbound gateways often add additional HTTP header to incoming HTTP messages such as the X-Forwarded-For header that forwards the original client IP to the destination server (which otherwise just sees the inbound gateway's IP address).

23.3.3 TLS Offloading

An inbound gateway often is exposed to the internet with a public IP address that is mapped to a public domain name. Clients on the internet thus have TCP connection with the inbound gateway. If their TLS connection is terminated at the inbound gateway, we speak of TLS offloading; if the TLS connection is terminated at some node behind the inbound gateway, we speak of TLS passthrough.

TSL offloading is the only option, of the inbound gateway needs to analyze application layer data to perform other functions like the ones described in the sections above.

23.3.4 Authentication and Authorization

Like TLS offloading, authentication, or authorization information like OAuth2 access tokens (cf. Section 30.4.1) can be evaluated at an inbound gateway, such that servers with protected resources do not have to if they trust the inbound gateway.

23.3.5 Network Intrusion Detection and Prevention

Inbound gateways can analyze ingress data on every network layer and thereby detect attacks. That way, they act as sophisticated firewalls that go beyond the rather simple rules of common firewalls whose rules are based on source IP addresses and ports.

23.3.6 Rate Limiting

Limiting the arrival rate of messages from specific clients can be useful to defend overutilization of systems, which may be inadvertent or malicious and then better known as denial-of-service attacks. For a detailed discussion refer to Section 21.7.3.

23.3.7 Usage Monitoring

Usage monitoring is similar to rate limiting but used for billing usage of certain services that are billed based on number of requests.

23.3.8 Load Balancing

Load balancing has been discussed in Chapter 19.

23.4 Database Management

A database server is a set of user-space process that manage a database. Further information about database servers can be found in Section 31.2.

23.5 Caching Service

A [caching service](#) is used to implement caches as discussed in Chapter 22. These cache entries are typically responses of database queries or partial HTML documents. Such caching services are typically implemented using a key-value in-memory store, which is some kind of NoSQL database.

Another typical use case is to cache session data in a caching service in order to keep such session data out of the memory of single server-side processes (cf. Section 21.2.3).

Common key-value in-memory stores for implementing a caching service are Memcached [90] and Redis [95] (cf. Section 32.2.1).

23.6 Analytics / Data Warehouse

Data warehouses consolidate mostly transactional data from different databases in order to provide a holistic view on business data for analytical purposes. This consolidation takes place in a recurring (e.g., once per night) so-called [ETL process](#), which stands for extraction of data from different databases, transformation of that data into the uniform schema of the data warehouse and loading that data into the data warehouse database storage.

With the rise of big data, it is got fashionable to not just analyze transactional data but to track events at a very detailed level, e.g., to track the navigation (clicks or even mouse movements) of the users on a website. These event data are pushed to streaming platform servers (cf. Section 26.2) and stored in data warehouses.

23.7 Partner Systems

In system landscapes, systems are integrated with each other. Integration mostly works by APIs that are exposed by the application servers of the respective systems (via a load balancer). Technically, integration is implemented by synchronous communication or asynchronous messaging between the systems (cf. Section 20.6) using a classical message broker like Apache ActiveMQ [96].

23.8 Domain Name System (DNS)

The internet's [domain name system \(DNS\)](#) provides a dynamic mapping of mnemonic addresses to IP addresses. So, a domain name is easier to memorize for a human than the IP address it maps

to. Moreover, the mapping to IP addresses is dynamic: Domain names for web applications can be kept constant if the mapped IP address changes.

A **domain name** is a sequence of **labels** separated by periods. A **domain** is a set of domain names that share a certain suffix. The **top-level domain** `de` contains all domain names that end with `de`; within that domain, the **second-level domain** `rewoit.de` contains all domain names, which end with `rewoit.de` and the **third-level domain** `parcer.rewoit.de` contains all domain names ending with `parcer.rewoit.de` and so forth. `rewoit.de` is also called a **subdomain** of `de` and `parcer.rewoit.de` a subdomain of `rewoit.de`.

Domain names that map to IP addresses are also called **hostnames**. To be exact, the leftmost label in such domain names is the hostname; the complete domain name is called **fully qualified domain name (FQDN)**.

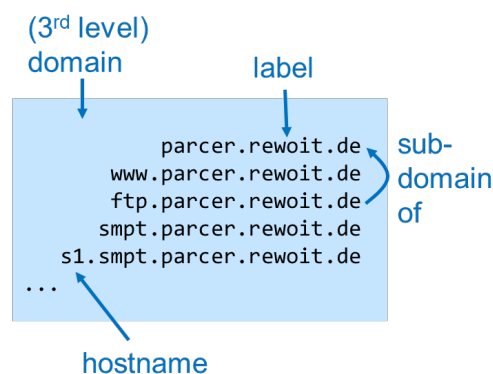


Figure 86: Domain names

The **domain name system (DNS)** is a hierarchical distributed database whose nodes are **name servers**. Each name server is equipped with a **zone file**, which contains a number of **resource records**. Each resource record is a mapping of a domain name⁵⁰ to

- another name server (NS record),
- an IP address (A record for IPV4 or AAAA record for IPV6),
- another (alias) domain name (CNAME record)

just to name the most important among all resource record types [98]. A **zone** is the set of all resource records with domain names of a certain domain.

⁵⁰ In a resource record, wildcards as defined in [97] are also allowed, e.g., `*.parcer.rewoit.de`. A lookup for, e.g., `xyz.parcer.rewoit.de` would match such a record.

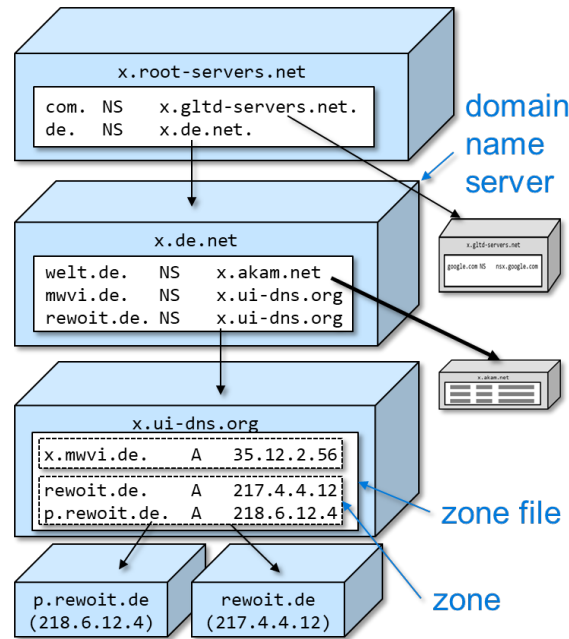


Figure 87: Hierarchy of domain servers

The hierarchy of domain servers is established by NS records.

EXAMPLE 97: HIERARCHY OF DOMAIN SERVERS

In the example of Figure 87, the domain server with the domain name `x.de.net` is authoritative for the zone `de`. However, its zone file itself mainly contains NS records for second level domains. For example, the server `x.ui-dns.org` is authoritative for the second-level domain `rewoit.de`. The zone file finally contains an A resource record for the host `p.rewoit.de`.

Though **authoritative name servers** are considered to be the single source of truth regarding the resource records of their zones, DNS makes extensive use of caching. Each client normally has an internal DNS cache with resource records of former queries from that client. In case of a cache miss, the client asks a nearby domain name server in either the cache of that domain server or its zone files. If the resource record still cannot be found, it queries root domain servers, which delegate the query to the appropriate authoritative domain server of the respective top-level domain. (There are just 13 hostnames for root domain name servers starting with `a.root-servers.net` and ending with `m.root-servers.net`, each of which with single IPV4 and IPV6 address. However, the query workload is actually distributed to many more servers by use of **IP anycast**.)

23.9 Web Servers / Content Delivery Network (CDN)

Modern web pages consist of numerous **web resources** like HTML, images, videos, cascading style sheets and JavaScripts. While HTML is often dynamically generated by request, the other resources are rather static, i.e., change just in the course of a deployment. **Web caching** [99] is a crucial concept in the architecture of the web. Of course, latency is smaller if a cache is close to the respective client. **Private local caches** (in browsers) with low latencies contain just those responses, i.e., web resources, which have already been requested by the particular client. On the other side, **origin servers**, which are often application servers (cf. Section 23.1), are far away from a client and can respond just with high latencies. **Shared caches** in **web servers** pose a compromise;

they are servers that receive requests before origin servers and serve matching responses in case of a cache hit.

A **content delivery network (CDN)** mainly consists of web servers acting as shared caches. These shared caches are distributed over the world. Requests from clients (browsers) for static web resources are routed to the closest shared cache in a content delivery network mainly in order to keep latency low. CDNs are an example of **edge computing**, which is a paradigm for building distributed systems where certain system resources are moved closer to the clients.

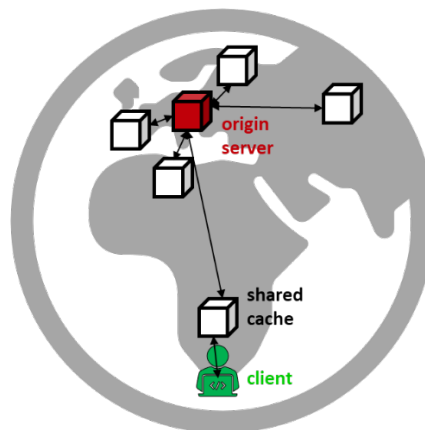


Figure 88: Content delivery network

23.10 Cloud Storage

Data can be persisted in many ways, i.e., stored in some kind of **storage**. A detailed discussion can be found in Section 31.1.

23.11 Outbound Gateway

Without an outbound gateway, a system can just react to incoming requests, i.e., communication from a user to the system. However, certain use cases require proactive communication from the system to the user. Here, an **outbound gateway** comes into play, which is just an umbrella term for diverse types of gateways.

The most common outbound gateways are **SMTP servers**, which can send emails generated by the system to the user, e.g., in order to verify an email address. Other types of outbound gateways include **SMS**, **WhatsApp** (for, e.g., **two-factor logins**) or servers for smartphone or browser **push notifications**.

23.12 Identity Provider

From a security perspective, a software system is a system of **entities** which might be end-users or clients. These entities might possess (digital) **identities**, which are basically sets of properties (key-value pairs) with elements like username, email address, etc.

In order to protect confidential data, each system backing a certain web application must perform

- **identification**, i.e., **recognizing an identity by some criterion**,
- **authentication**, i.e., implement a process by which an entity can prove to possess an identity as well as
- **authorization**, i.e., determining which entity is permitted for which actions, e.g., accessing which web resource.

In software systems, the common criterium for identification is non-confidential string, e.g., a username, which might be identical to an email address of that user.

Authentication can be implemented in various ways. It is still common that, in order to authenticate with a system, the user provides a secret string called password that is hashed and transferred to the system, where the hash value is compared with a stored one.

Oftentimes, authentication is just a means for authorization. As soon as the user is authenticated, he or she is authorized for actions that are, e.g., configured for the particular identity. Please note that authentication is not necessarily a precursor for authorization. For example, a bus ticket authorizes a passenger to enter a bus but does not identify let alone authenticate him or her. So, generally, one cannot use authorization as a precursor for authentication.

In the past, it was common that every user had a different identity for each web application and this identity (after authentication) was merely used to authorize the user for certain actions. Nowadays in order to avoid user password fatigue and enforcing compliance requirements (password strength, renewal periods, highly secure credential storage, etc.) authentication often is delegated to and centralized in an identity provider. An **identity provider** is a system that performs **authentication** of a user on behalf of another system. For web applications this is typically done by a temporary redirect of the browser from the requesting web application to the identity provider, which in turn issues some kind of **token**, to the requesting web application (cf. Section 30.4f). The token contains data that identifies as well as authenticates the user. A single identity provider usually performs authentication for multiple systems which then are called to be members of a **federated identity management**.

Besides authentication, identity providers can be used to **authorize** some web application A to access non-public web resources by some user stored in (the database of) some web application B. This is particularly useful for a user's master data, e.g., contact information (address, email address, phone number) which is otherwise cumbersome to keep current and consistent over multiple web applications.

In recent years, several standards have emerged in the field of identity management. **OAuth2.0** (cf. Section 30.4) is an authorization framework that often relies on exchanging (browser cookie) tokens following the **JSON Web Token (JWT)** (cf. Section 30.5) standard. **OpenID Connect (OIDC)** (cf. Section 30.6) standardizes the way authentication should be done using OAuth2.0 and JWT.

Identity providers can be part of a single sign-on implementation in a company's intranet but also be SaaS offering which is publicly available. Big players like Facebook, Microsoft, Google, and GitHub are the most common identity providers in the internet surfacing by the well-known "Login with X" button.

23.13 Monitoring Server

A monitoring server gathers, consolidates and visualizes metrics, traces and logs of one or more systems clusters.

23.14 CI/CD/MLOps Pipeline

A CI/CD pipeline is a program that builds deployable software artifacts or machine learning models, tests them using automated tests, archives them in artifact (image) repositories, and deploys them to the respective machines.

23.15 Source Code Repository

A source code repository contains different versions of file trees, where the files are typically the constituents of a certain program including auxiliary resource files (configuration files, media files etc.)

24 Cloud Computing

In Chapter 23, we have described infrastructure and middleware resources that are common for a cloud-based web application. In this chapter, we explain what cloud computing is about and how to actually build a system cluster with (some of) these resources in the Google Cloud.

24.1 Cloud Characteristics

[100, p. 2] gives the most accepted definition of cloud computing.

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models.

Essential Characteristics:

On-demand self-service. *A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider.*

Broad network access. *Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, tablets, laptops, and workstations).*

Resource pooling. *The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the resources provided but may be able to specify location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, and network bandwidth.*

Rapid elasticity. *Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.*

Measured service. *Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service."*

24.2 Service Models

Services offered by cloud providers can be categorized according to the degree of prefabrication and specialization for certain customer needs. There is a whole spectrum of service models bounded by **Software-as-a-service (SaaS)** at the upper end, which provides read-to-use, multi-tenant web applications with certain functionality. At the lower end of the spectrum, **Infrastructure-as-a-service (IaaS)** essentially provides storage, network, and virtual machines with installed operating systems and leaves much flexibility for the customer to operate arbitrary software on them. The other models are somewhat in-between as they provide preinstalled

runtime environment, middleware or container services which allow for a faster and more convenient deployment than IaaS.

A classic, commonly accepted definition can be found at [100, pp. 2–3]:

Software as a Service (SaaS). *The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings.*

Platform as a Service (PaaS). *The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment.*

Infrastructure as a Service (IaaS). *The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls)."*

Please note that there are cloud providers like Heroku [101] that are **second-level providers** as they do not possess own locations and hardware but provide PaaS or particularly SaaS offerings based on IaaS capabilities of **first-level providers**, which is AWS in the case of Heroku.

In the last years, even more nuanced service models than those above have emerged. Figure 89 attempts to delineate these from a technological point of view, where upper filled boxes represent resources that run and/or depend on the lower ones. The bars on the right side indicate what is managed by the cloud provider and what is left to manage by users.

It is tempting to prefer a SaaS over PaaS and PaaS over IaaS if the SaaS or PaaS match the specific requirements for a certain system. However, you should keep in mind that the more you delegate to a cloud provider to manage, the higher the **vendor lock-in** will be.

EXAMPLE 98: CLOUD VENDOR LOCK-IN

It is easy to migrate a program from an Amazon EC2 virtual machine to a Google Cloud virtual machine but less so to convert AWS Lambda function to Google Cloud functions with all their specific dependencies.

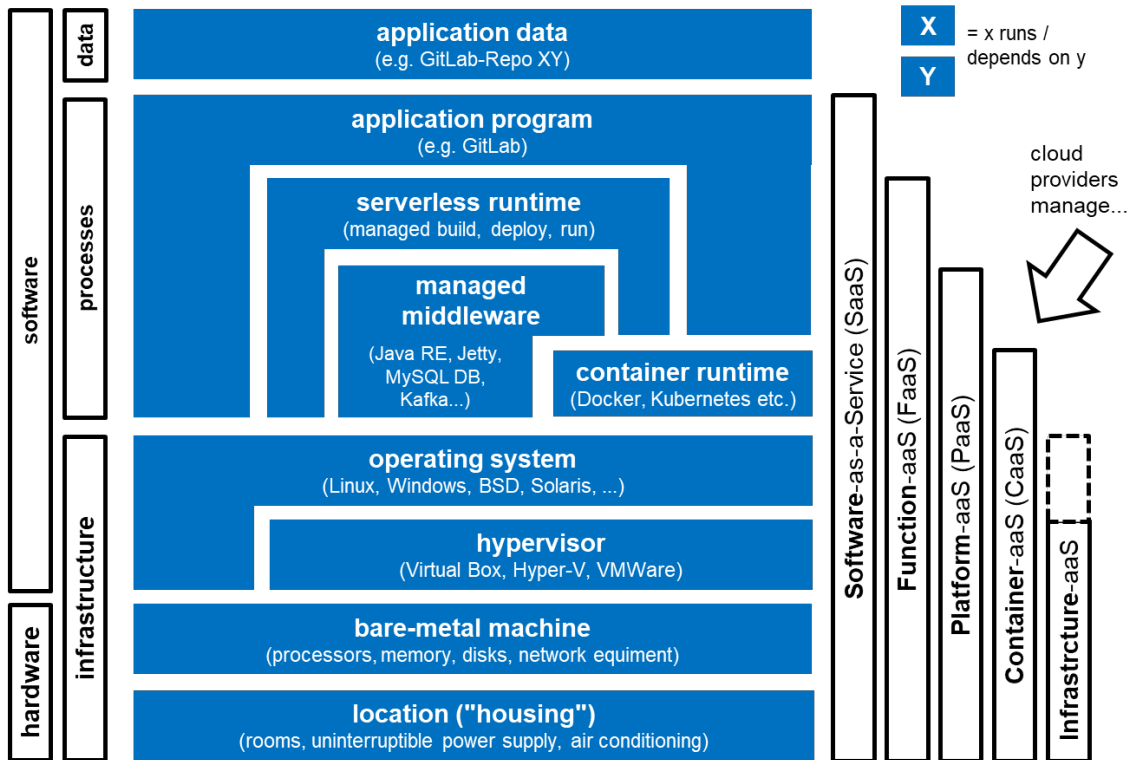


Figure 89: Cloud service models

Some cloud providers—particularly the big ones like Amazon, Microsoft, and Google—offer more than one service model. Since these offerings are of course integrated, it is sometimes hard to tell the service models apart.

EXAMPLE 99: SPECTRUM OF SERVICE MODELS IN THE GOOGLE CLOUD

The Google App Engine [102] falls in the category of a Function-as-a-Service while the Google Compute Engine [103] in its core is an Infrastructure-as-a-Service service model.

24.2.1 Infrastructure-as-a-Service (IaaS)

The bottom-most resource is the mere **location**, i.e., **housing** for hardware. There are providers which provide just this, i.e., computer-friendly space that can be rented in order to accommodate your own hardware there.

Bare metal machine denotes the actual physical hardware computers are made of. Such machines either run an operating system kernel or a **type-1-hypervisor** (cf. Section 25.1.3).

Usually, Infrastructure-as-a-Service (IaaS) also includes automated provisioning of **virtual machines** based on these hypervisors with preinstalled operating systems. Also, automated provisioning of **object storage** is part of the IaaS service model.

The amount and size in terms of CPU, memory and storage of these IaaS resources is the key factor for the pricing. Virtual machines usually have to be paid as long as they are switched on, irrespective of whether they are actually utilized by processes or not.

EXAMPLE 100: INFRASTRUCTURE-AS-A-SERVICE IN AWS AND GOOGLE CLOUD

Amazon and Google offer services for automated provisioning of virtual machines called Amazon EC2 and Google Compute Engine, respectively. Their service for object storage is called Amazon Simple Storage Service (S3) and Google Cloud Storage, respectively.

24.2.2 Container-as-a-Service (CaaS)

A **Container-as-a-Service** service model offers preinstalled and managed **container (orchestration) runtimes** like Docker (cf. Section 25.3.3) and/or Kubernetes (cf. Chapter 25.4). Users are therefore relieved from installing and managing these runtimes by themselves on virtual machines. Also, the build of container images from source code, which normally happens on locally on a developer's machine by means of build tools that come along with container runtimes, can be delegated to CaaS offerings and thus shifted to the cloud.

Pricing for CaaS services mainly depends on the actual usage of (virtual) hardware resources, i.e., number of vCPU * seconds, allocated memory in GiB * seconds and number of (HTTP) request messages. So, as opposed to IaaS, one particularly need not pay for idling virtual machines.

Persistent storage is generally not included in offering of CaaS servers and therefore must be booked separately.

EXAMPLE 101: CONTAINER-AS-A-SERVICE IN AWS AND GOOGLE CLOUD

Amazon and Google's services in this model are numerous and somewhat hard to delineate.

Amazon Elastic Kubernetes Engine (EKE) and Google Kubernetes Engine (GKE) are managed Kubernetes services. They relieve the cloud user from installing and maintaining Kubernetes clusters on virtual machines but still require the full desired state from the cloud user in all its complexity and flexibility. So, the cloud user should still have profound knowledge how Kubernetes works along with its tools like `kubectl`, schemas and configuration files (cf. Section 25.4.2).

Amazon Fargate and Google Cloud Run go one step further and abstract away the complexity of Kubernetes from the cloud user. With these cloud services, the cloud user just has to provide a premade container image or a Dockerfile. The creation of a Kubernetes deployment and service along with autoscaling, registering a subdomain name, and a TLS certificate is done automatically without any further ado from the cloud user.

24.2.3 Platform-as-a-Service (PaaS) / Backend-as-a-Service (BaaS)

Platform-as-a-Service even more relieves the customer from tedious, repetitive, and error-prone installations and configuration of common middleware like Java runtime environments, message queues, databases, and the like. Instead, such middleware can typically be provisioned by just a single (or just a few) clicks or commands.

Almost synonymous with PaaS is **Backend-as-a-Service (BaaS)**. BaaS particularly focuses on automated provisioning of middleware which is used on the server-side of web application and smartphone applications, like databases, cloud storages, outbound gateways for push notification, and identity providers.

24.2.4 Function-as-a-Service (FaaS)

Function-as-a-Service can be seen as a combination of CaaS and PaaS: FaaS also manages the middleware within containers, which are basically programming language runtimes, e.g., the Java Runtime Environment (JRE) or NodeJs, and proxy web servers, and the connection to PaaS services outside of containers like message queues.

Moreover, FaaS services also care for the build out of source code, i.e., compilation and packaging target code into container or micro virtual machine images. The cloud developers only have to provide the pure source code but not build configuration files like `package.json` for NodeJs or a

Dockerfile for building container images. That is why FaaS is usually restricted to just a few programming languages and ecosystems. Pricing in FaaS is similar to CaaS.

EXAMPLE 102: FUNCTION-AS-A-SERVICE IN AWS AND GOOGLE CLOUD

Amazon Lambda and Google Cloud Functions are concrete FaaS services.

24.2.5 Serverless Computing

The term **serverless computing** can be seen as an umbrella term for CaaS, FaaS and BaaS. The term is a clear misnomer, since not the servers vanish. Instead, just the burden of allocating resources cost-effectively and deploying them is taken off the shoulder of cloud users.

However, developers have to be aware a major restriction that comes along with serverless computing: Process recreation, i.e., stopping and restarting application programs on the same or different machine due to failover or horizontal scaling, is completely taken out of control from the developer. Therefore, developers have to cope with two important architectural constraints in FaaS and CaaS:

1. During restart, the process virtual memory is lost, i.e., the data, heap, and stack sections of it⁵¹. Thus, server-side sessions (cf. Section 28.2) are effectively unusable in FaaS and CaaS.
2. Processes may be stopped and not restarted until there is demand, e.g., a certain request arrives which needs to be processed by the respective process. Therefore, spin-up time (cf. Section 5.2.1) may increase the overall response time significantly known as the **cold-start problem**.

These constraints render FaaS and CaaS useful for workloads where (1) no state at all needs to be stored between requests or can be efficiently offloaded to the client or a backing database (cf. Section 28.2 and (2) spin-up time is negligible compared to processing time.

EXAMPLE 103: VIDEO PROCESSING FOR FAAS OR CAAS

A function that takes a video input stream as input and produces a, e.g., transcoded one as output would be a perfect match for FaaS or CaaS. In this case, no state needs to be stored since transcoding one video does not affect another one and processing time would be in the minutes or hours and therefore a, say, two seconds spin-up time would not matter much.

24.2.6 Software-as-a-Service

In SaaS, customers get a fully functional application. The only thing that customers still have to do is using this application and producing (transaction) application data.

EXAMPLE 104: GITLAB.COM AS SOFTWARE-AS-A-SERVICE

GitLab.com is a perfect example for a Software-as-a-Service.

24.3 Deployment Models

Cloud **deployment models** define where the cloud resides relative to a customer and a cloud provider. [100, p. 3] define the following delivery models:

"Private cloud. *The cloud infrastructure is provisioned for exclusive use by a single organization comprising multiple consumers (e.g., business units). It may be owned, managed, and operated by the organization, a third party, or some combination of them, and it may exist on or off premises.*

⁵¹ The text section can be restored by reloading the program.

Community cloud. *The cloud infrastructure is provisioned for exclusive use by a specific community of consumers from organizations that have shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be owned, managed, and operated by one or more of the organizations in the community, a third party, or some combination of them, and it may exist on or off premises.*

Public cloud. *The cloud infrastructure is provisioned for open use by the general public. It may be owned, managed, and operated by a business, academic, or government organization, or some combination of them. It exists on the premises of the cloud provider.*

Hybrid cloud. *The cloud infrastructure is a composition of two or more distinct cloud infrastructures (private, community, or public) that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load balancing between clouds)."*

So, an on-site private cloud is indeed like a classical data center owned and operated by the company that uses it but exhibits the same characteristics as a public cloud (cf. Section 24.1). An off-site private cloud, also known as **virtual private cloud**, resides in data centers of a public cloud provider. Such VPC are isolated from the public cloud and other virtual private clouds by means of, e.g., network topologies, i.e., each virtual private cloud has its own virtual private network.

Beyond these definitions, the **multi-clouds** (a.k.a. **inter-clouds**) have become popular in the last years. These clouds distribute systems of a customer to public clouds of different cloud providers, mainly in order to avoid dependencies from and lock-ins to a single cloud provider.

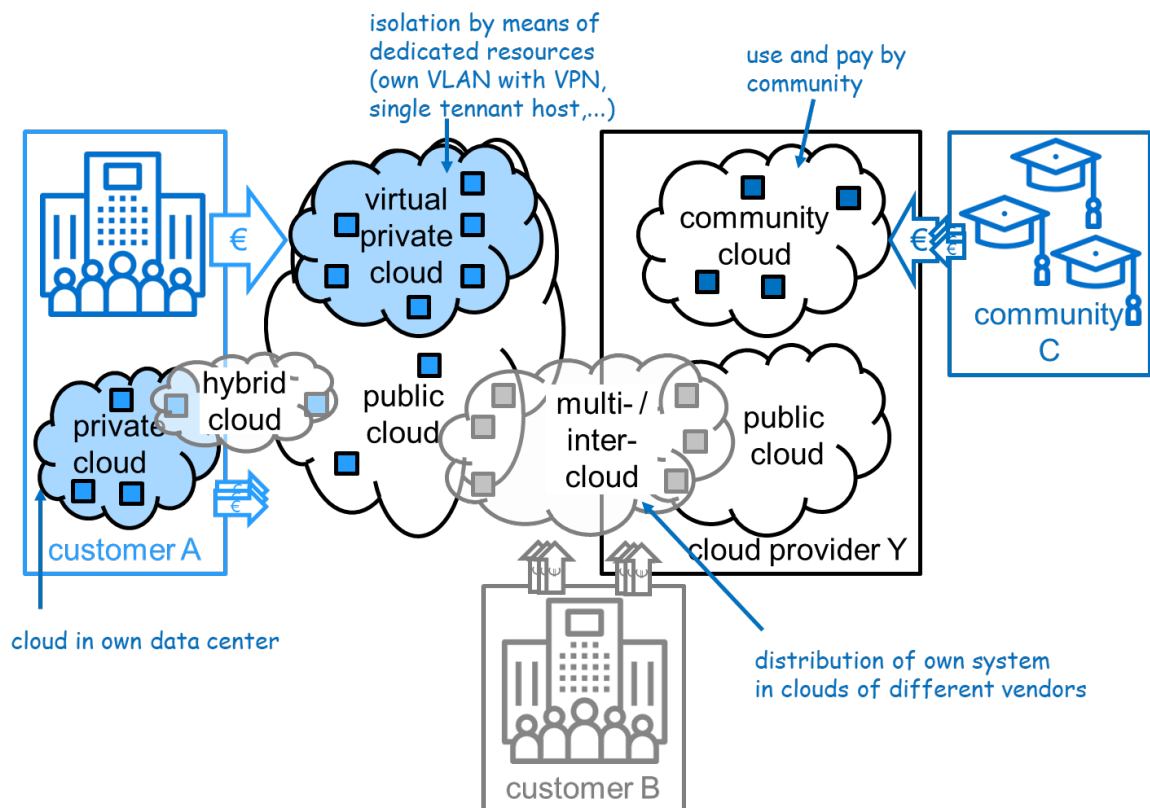


Figure 90: Cloud deployment models

24.4 Utilization and Costs

Resource pooling on large scale hardware resources (CPU, memory etc.) leads to a better utilization of these resources. The assignment of demands for (virtual) resources to actual resources often follows some bin packing algorithm [104]. Normally, assigning comparably large resource demands to comparably small resources leaves much unused "daylight" in these resources, i.e., undersaturates them and does not use their full capacity, respectively. In the visualization of Figure 91, the resource on the left side is just used by $5/8$. Therefore, a request for $1/2$ of that resource cannot be satisfied any more. The fraction of unused resources is lower if demands are comparably small as the right side of Figure 91 shows.

From a cost perspective, it is in the very interest of cloud providers to keep the utilization of hardware resources close to 100% in order avoid waste. This can only be achieved by serving the demands of multiple customers because of the binpacking effects and since variations in demands of customers may compensate each other.

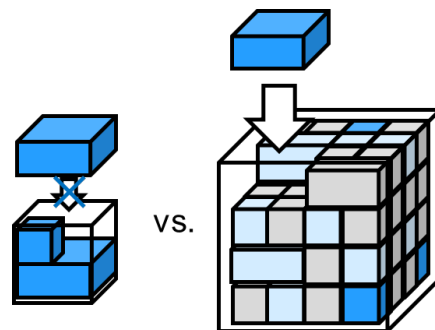


Figure 91: Binpacking in small and large resources

From a customer's point of view, rapid elasticity is attractive because it contributes to the ideal cost-effective scalable system of Figure 19. Self-hosting leads to over- and underutilized systems, which in turn lead to poor performance thus customer's customer dissatisfaction and waste of money, respectively. Figure 92 depicts both cloud-hosting and self-hosting from a cost perspective.

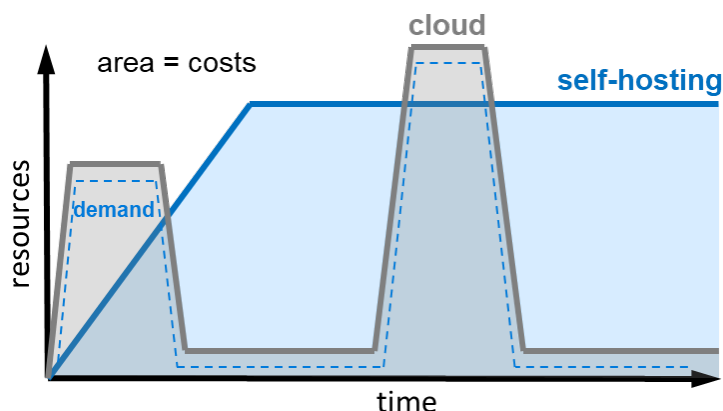


Figure 92: Cloud-hosting vs. self-hosting

Sometimes, cloud-hosting is also attractive for stakeholders of the financial department. Expenditures for self-hosting are **capital expenditures (CAPEX)**, those for cloud-hosting are **operational expenditures (OPEX)**.

24.5 Cloud Services

The scope of functionality which cloud providers offer is overwhelming. Therefore, it is not trivial to derive a common categorization for this functionality that suits the offerings of every cloud provider. At the bottom level, we group coherent functionality into **cloud services**, a.k.a. **managed services**.

EXAMPLE 105: "VIRTUAL SERVERS" AS TYPICAL CLOUD SERVICES

"Virtual Server" is a common cloud service that every cloud provider offers yet with its own branding. In the Google Cloud the respective cloud service is called "VM instances".

Since big cloud providers even offer more than hundred different cloud services, we can further group them into **cloud service categories**. One ambitious attempt to map the provider-specific cloud services to a common cloud service name and to further group them into cloud service categories can be found in [105]. In Table 35, we just provide a provider-agnostic and condensed overview of cloud service categories.

Cloud service category	Description
compute	Compute services provide provisioning of dedicated, single tenant, bare-metal machines, as well as virtual machines, or containers together with their orchestration.
storage	Cloud storage options and their trade-offs are discussed in Section 31.1.
database	Database services are preinstalled multi-tenant databases or automatically installed single-tenant databases with different data models, i.e., relational databases, NoSQL databases etc.
networking	Networking services include the provision of content delivery networks (cf. Section 23.9), of permanent public IP addresses, firewalls, load balancing, virtual private clouds and virtual private networks.
developing tools	Developing tools include repositories for versioning source code, build artifacts and containers, build automation (continuous delivery pipelines), software development kits, and plugins for popular IDEs.
administration & monitoring	Cloud services can usually be configured via command line tools, via a web UI or programmatically via a web API. Monitoring services include tools for metrics, logging, and tracing (cf. Section 10.1.1).
security	Security services include the automated provisioning of X.509 certificates (for secure communication via https), key management, centralized identity management (IAM)
analytics	Analytics services include services for storing, transferring, converting and analyzing huge amounts of (event) data. Data warehouses (cf. Section 23.6) are also included in this category.
artificial intelligence	Cloud computing provides potentially vast computing power, which leads to another spring in the field of artificial intelligence [106]. Cloud AI services typically support special AI use cases like language processing and translation, speech and image recognition, or machine learning.
internet of things	Although "things" on the internet of things are rather on the edge of a system and not in the cloud, certain services like message brokers in the cloud can also support IoT use cases.
misc. services	Some services are versatile, support very specific use cases or contribute to diverse categories from above. So, they are not included in one of the above categories.

Cloud service category	Description
	Message brokers (cf. Section 26.1) are among these as well as, e.g., email services, notification services, chatbots, blockchains, media transcoders or API management services.

Table 35: Cloud service categories

24.6 Legal Considerations

It is worth noting that US cloud providers as of Q2 2025 have a cumulative market share of about 70% as shown in Figure 93. These US cloud providers fall under the jurisdiction of US law and particularly U.S. Code 2713 that says they *“shall [...] preserve, backup, or disclose [...] communication and any record or other information [...] within or outside the United States”* [107]. These “outside” locations are particularly data centers (zones) of US cloud providers outside US soil.

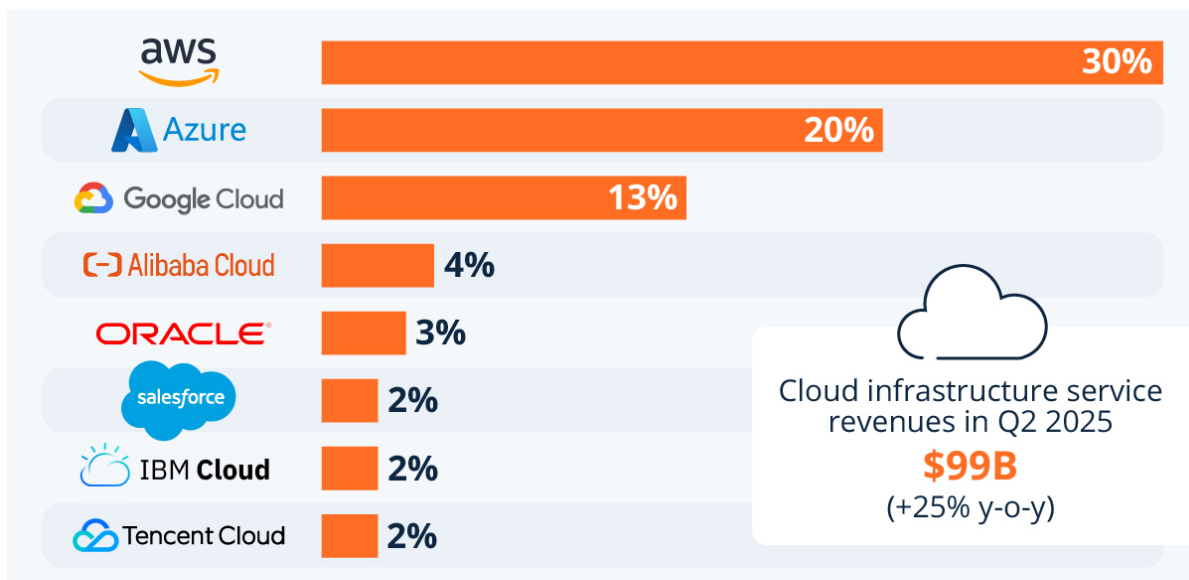


Figure 93: Cloud Provider Market Share Q2 2025 [108]

24.7 Cloud Native

Cloud computing leverages the use of certain technologies, architectures and practices. Although these are viable even without cloud computing, they are often apostrophized as **cloud native**.

The Cloud Native Computing Foundation (CNCF) defines **cloud native practices** as follows [109]:

“Cloud native practices empower organizations to develop, build, and deploy workloads in computing environments (public, private, hybrid cloud) to meet their organizational needs at scale in a programmatic and repeatable manner. It is characterized by loosely coupled systems that interoperate in a manner that is secure, resilient, manageable, sustainable, and observable. Cloud native technologies and architectures typically consist of some combination of containers, service meshes, multi-tenancy, microservices, immutable infrastructure, serverless, and declarative APIs – this list is non-exhaustive.”

This highly condensed definition requires some explanation. Cloud native propagates:

- **Portability between Deployment Models**, i.e., an application should be agnostic whether it is being deployed, e.g., in the Google Cloud, in a Kubernetes cluster in an on-prem private cloud or a mixture of them
- **Microservice Architectures** (cf. Section 20.4.2) with
 - loosely coupled, thus
 - independently deployable services that can
 - be independently out-scaled but also in-scaled to zero and are
 - stateless in the sense of Section 21.2
- **API-First Design**, meaning the design and development of a microservice starts with its web API that every service mandatorily exposes
- **Container Virtualization and Orchestration** (cf. Sections 25.3f) and **Serverless Computing**
- **Continuous Integration and Deployment (CICD)**
- **Infrastructure as Code (IaC)**, i.e., tools like Terraform that can automatically provision immutable cloud resources based on configuration files that are developed and versioned a together with source code files.
- A **DevOps** culture that aims at
 - increased deployment frequencies and success rates,
 - rarer and shorter system downtimes, and
 - broader job roles in the sense that the same people can build and operate a system.
- **Observability** by active use of platform services of the respective cloud provider that enables centralized logging, monitoring, and tracing, horizontal scaling, encryption etc.
- **Security** by means of principles like least-privilege, network-based security, zero trust architectures etc.

EXAMPLE 106: COUNTEREXAMPLE OF CLOUD-NATIVE DEVELOPMENT

A

- large (200K+ source code lines) monolithic application, that is
- built by means of obscure Bash scripts in dozens of manual steps by developers and
- deployed every six months by operators with a
- success rate of 50% in the first attempt, that has
- no web API,
- whose numerous deployment artifacts are directly copied to a single
- just vertically scaled virtual machine whose
- setup has been tweaked over years beyond any reproducibility, which
- tracks session states by sticky sessions, which
- neither encrypts data in rest nor transit, and which
- does not contribute to any centralized logging, monitoring nor tracing

is a realistic representative for enterprise applications before the advent of cloud computing but quite the opposite of being cloud-native.

24.8 Zero Trust Architectures

Before the advent of cloud computing, mobile devices or Internet of Things, it was a feasible to authorize users and/or clients by their mere network location, i.e., IP address including virtual IP addresses that are assigned to devices not by physical connection to some switch but by means of virtual private networks (VPN). A node was typically implicitly authorized to access another node

in the network, if both were located in the same local area network or if the network route was explicitly permitted by respective firewall rules. This security model is often referred to a **network perimeter based**: A node is either within a network perimeter and thus authorized or not.

This assumption does not longer hold true in **zero trust architectures** [110], which is applicable for enterprise computing in general (using cloud computing or not). In a zero-trust architecture,

- each system resources is protected on its own instead. Accesses to that resources require **explicit authentication and authorization**,
- by default, only those **minimal permission** should be granted that are necessary instead of granting all but harmful permission,
- grants are **short-lived** and therefore have to be evaluated frequently,
- network communication is **encrypted** regardless of the routes, and
- all system resources are strictly **monitored** w.r.t. their security state.

Since these are rather vague guidelines, cloud providers concretize them in their offerings. For example, Google propagates its zero-trust architecture BeyondCorp [111] for enterprise computing and maps it to cloud native architectures [112].

24.9 Google Cloud

This section provides an overview of the most important concepts and cloud services of the **Google Cloud**. We are particularly interested in the IaaS/PaaS offering **Google Cloud Platform (GCP)**. The SaaS offering **G Suite** is not in the scope of this course nor this lecture notes. Moreover, we just describe basic concepts and services of the Google Cloud. Please refer to the official documentation [113] for further information or to the GCP tutorials [114] for further training.

24.9.1 Resources

Resources of a cloud-based system are not directly accessible by customers. Instead, each resource is configured through a data object. In the following, we use the term **resource** homonymous for an actual resource and its backing data object. A resource is typically an instance of some cloud service.

EXAMPLE 107: GOOGLE CLOUD RESOURCE

In GCP, one can create a virtual machine using the cloud service "Compute Engine".

24.9.2 Projects

In GCP, each resource is assigned to exactly one **project**, i.e., a project is a logical aggregation of resources. Normally, a project corresponds to a single system cluster.

Projects are identified by

- a **project name** that need not be unique and thus can be arbitrarily chosen,
- a **project id**, whose suffix is generated in order to make the project id globally unique,
- a **project number** which is completely generated and globally unique.

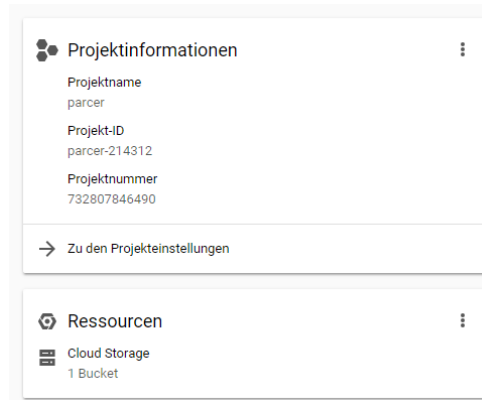


Figure 94: Project with its identifiers and a single resource

24.9.3 Administration

Cloud customers can access their projects and resources via RESTful web APIs (cf. Section 27.1.9), which are exposed by the Google Cloud Platform. On the client side, there are three ways to access these APIs:

- via the command line tool **gcloud**,
- via **client libraries** for diverse programming languages, or
- via a web UI called **Cloud Console**, accessible via <https://console.cloud.google.com> which particularly provides a browser-based terminal called **Cloud Shell** with a preinstalled gcloud.

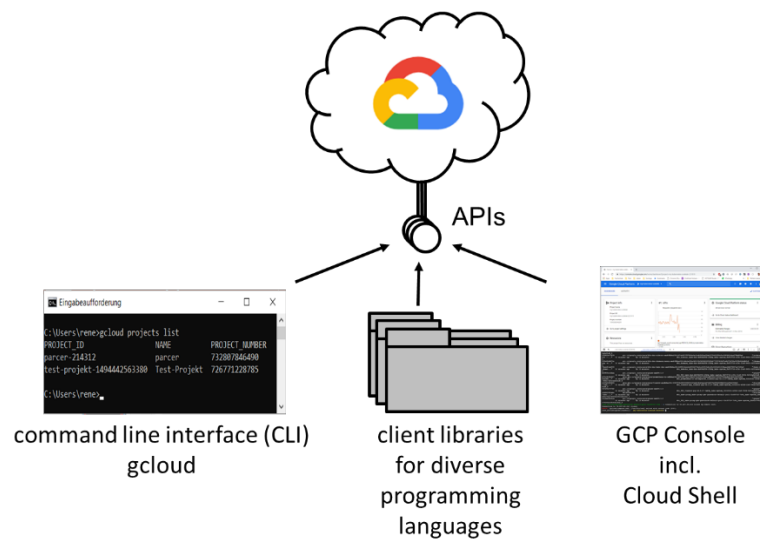


Figure 95: Ways to access Google Cloud Platform's web APIs

Since gcloud exposes all services and options of the Google Cloud Platform to the command line, it has many commands, which are documented in [115]. Therefore, commands are organized in a hierarchy. A command line typically looks like this

gcloud <command_group> <command> <subcommand>

Figure 96 depicts a small fraction of the command hierarchy.

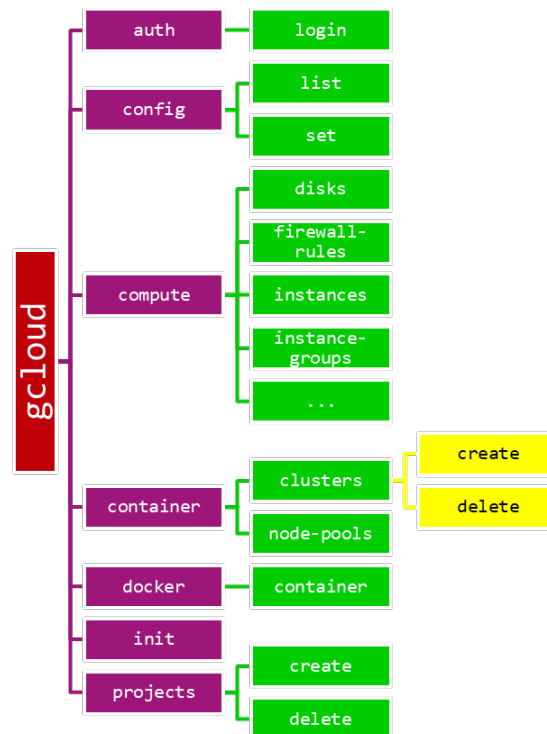


Figure 96: Gcloud command hierarchy

24.9.4 Regions and Zones

After all, each actual resource is located in some data center. In GCP data centers are called **zones**. Different zones might have slightly different equipment, i.e., just some zones provide single-tenant hosts, GPUs, local SSDs etc. <https://goo.gl/maps/Ew2YADyoKoq> shows a street view from the inside of a GCP data center.

Zones are geographically grouped into **regions**. Roughly speaking, a region corresponds to a city area like Frankfurt am Main and contains about three zones. At the time of writing, GCP has 18 regions worldwide.

Regions have a unique name like `europa-west3` for Frankfurt. A zone has the name of its region suffixed by a letter, like `europa-west3-a`.

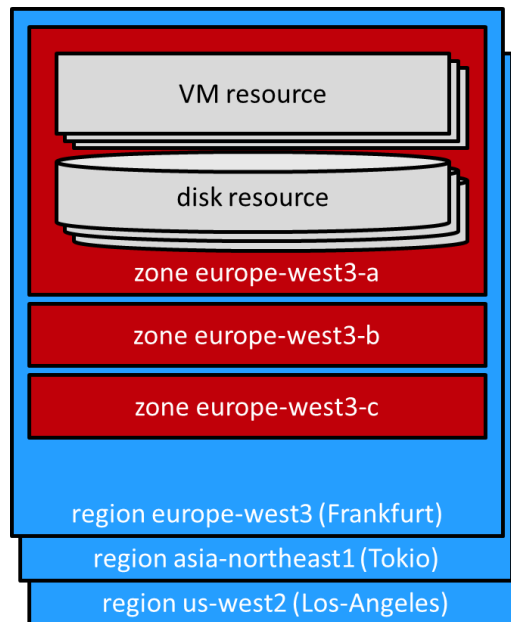


Figure 97: Regions and zones

24.9.5 Pricing

Two factors underly pricing of resources: (1) The quality of a resource, e.g., number of CPU cores and memory for a virtual machine and (2) the actual use of a resource, which is measured by cloud providers (cf. Section 24.1). The "use" is measured in usage duration and/or used capacity or throughput depending on the used resource.

Price calculations are very complicated. In order to give a rough impression: As by October 2018, a virtual machine with 4 vCPUs, 15 GiB memory, and a 375 GB SSD costs about 100 \$ per month.

Other pricing benchmarks are as follows:

- network
 - 0,10 \$ / GB for region ex-/inbound traffic
 - 0,01 \$ / GB for interzonal traffic within region
 - 0,06 \$ / h / VPN tunnel
- 1 vCPU
 - 20 \$ / month
 - 13 \$ / month (fixed for 3 years)
 - 6 \$ / month for pre-emptive VMs
- memory
 - 3 \$ / GB / month
- disks
 - 0,05 \$ / GB / month (default HDD)
 - 0,30 \$ / GB / month (SSD in particular region)

24.9.6 Billing

A project must be linked with a [billing account](#) before resources can be created within it. Resource costs are accumulated in a project and transferred from the linked billing account in certain intervals. Billing accounts can be saved by unlinking projects. However, this eventually leads to

the deletion of resources within the respective project after some time and is recommended just for test projects.

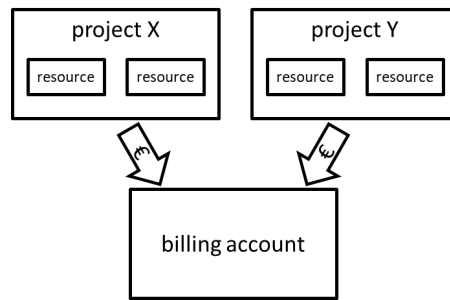


Figure 98: Resources, projects and billing accounts

24.9.7 Storage

As described in Section 23.10, storage is a hypernym for many kinds of storages. Figure 99 depicts a kind of decision tree for GCP storage options with common use cases in the leaves.

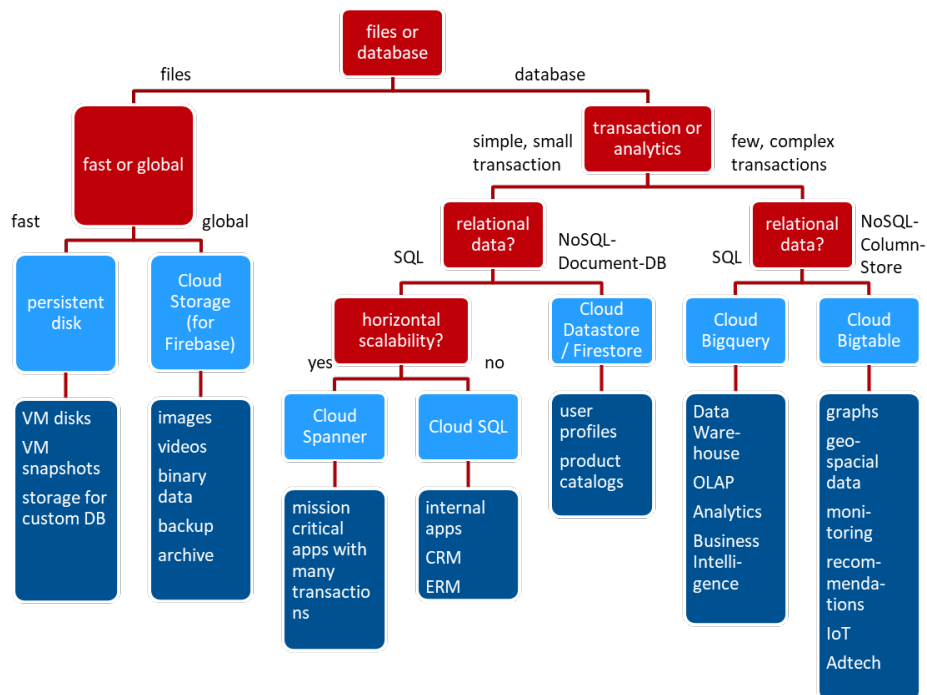


Figure 99: GCP storage options

The left-hand side of Figure 99 depicts file-based storages, where, for the sake of simplicity, files and objects in buckets are treated to be the same. Table 36 characterizes both file-base storage options, namely persistent disks and cloud storage.

	Persistent disk	Cloud storage
Access	- VMs must be in same zone/region - read-write just by single VM - via OS calls	- global access - read-write just by multiple VMs - via REST-API

	Persistent disk	Cloud storage
Performance	comparably fast	comparably slow
Maximum size	10 terabytes	multiple petabytes
Data model	- block-oriented (unformatted) - file-oriented (if formatted) - no versioning	- objects in "buckets" - versioning is possible
Use cases	- file server - storage for (No)SQL databases	- content distribution - backup, archive - data analytics (Map-Reduce)
Classes	- HDD / SSD / memory - regional / zonal / local	- multi-regional (e.g., web content) - regional (e.g., data analytics) - nearline (e.g., backup) - coldline (e.g., archive)

Table 36: Persistent disks vs. Cloud Storage

24.9.8 A GCP-based System Cluster

Figure 100 depicts a UML deployment diagram for a typical system cluster in GCP.

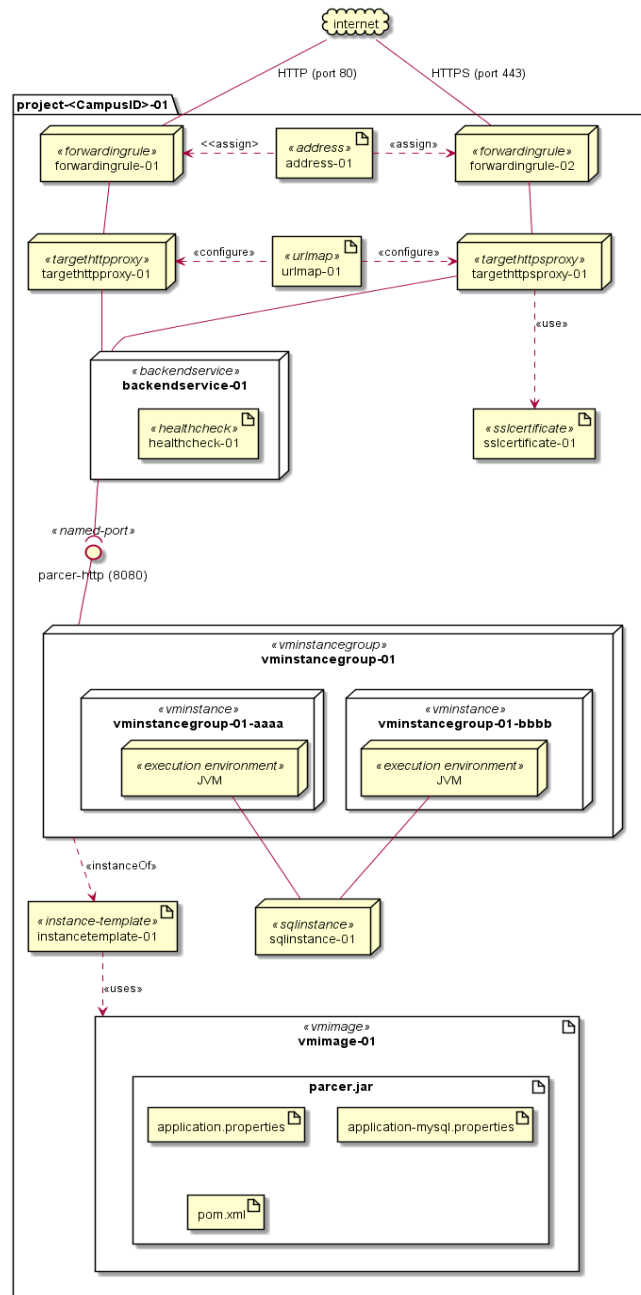


Figure 100: System cluster in GCP

a) Virtual Machine (VM) Instances

The cluster comprises the following resources (from bottom to top): A **VM image** is a disk image, i.e., a zipped boot disk for a VM instance. It contains necessary operating system files plus middleware or even a deployed application, `parcer.jar` in our case. **VM templates** reference such VM images but also contain specifications for VM instance resources (number of vCPUs, amount of memory). **VM instances** can be instantiated from a VM template, which are identical up to their IP addresses and hostnames. Such VM instances can process the same workload and are therefore grouped into **VM instance groups**. Figure 101 shows these relations.

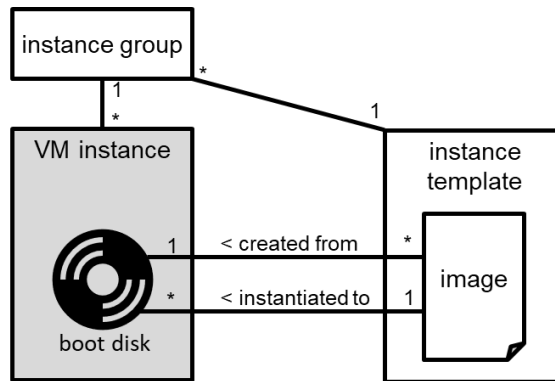


Figure 101: VM instances, groups, templates, and images

All VM instances expose the same TCP or UDP ports, which can be named at the VM instance group level as **named ports**.

b) Load Balancing

Incoming packets from the internet first hit a **forwarding rule** since it is this forwarding rule that is associated with a public IP **address**. If the packets constitute a http(s) request, the request is forwarded to a **http(s) target proxy**. If it is an https target proxy, it has an associated **SSL certificate** and terminates SSL (TLS) encryption. An associated URL map configures a mapping from requested URLs to **backend services**, which is capable of forwarding the request to healthy VM instances in VM instance groups. **Health checks**, which can be assigned to backend services, periodically check if VM instances respond reliably.

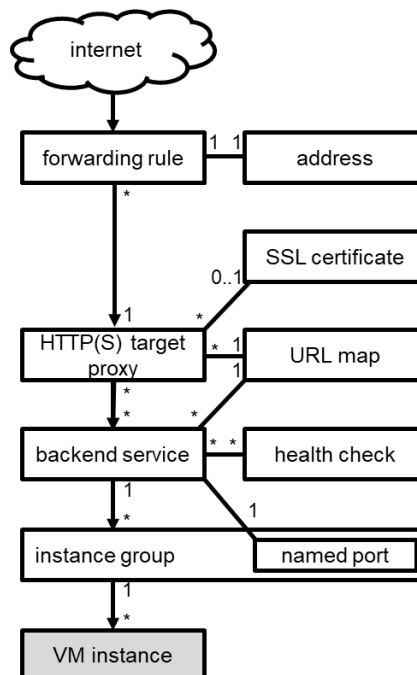


Figure 102: Load balancing

24.9.9 Scaling SQL Instances

Scaling out databases is a little more complicated than scaling out VM instances. In GCP a database management system is a set of so-called **SQL instances**. Besides the database itself, these SQL instances manage binary logs. A **binary log** buffers data modification requests (INSERT, DELETE,

CREATE etc.) before they are executed in the database. In order to create a **read replica**, which is a replica that receives only read requests (SELECT), the database gets a **backup**, which is restored to another SQL instance. Each subsequent modifying request is then propagated to both databases from there on.

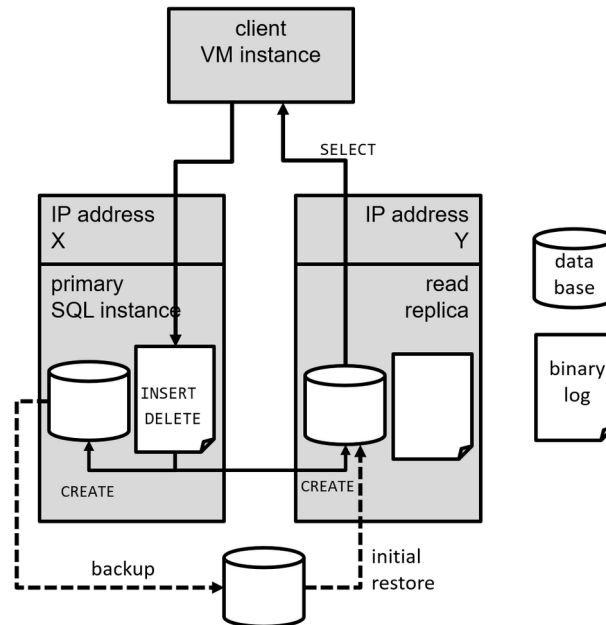


Figure 103: Read replicas

24.9.10 Database Failover

Failover in GCP databases can cross different zones. This keeps up the availability of such databases even in case of a disaster, which affects the whole data center (zone). In Figure 104, we have an active-passive-database (cf. Section 8.5.2), with an active **primary instance** in zone europe-west-3-a and the passive one in europe-west-3-b with a **failover replica**.



Figure 104: Database before (left) and after failover (right)

If the read-write-instances of europe-west3-a goes down, the SQL instance of europe-west3-b takes over. In order to keep the failure opaque to the database clients (e.g., VM instances), another read replica is created in europe-west3-b and the IP addresses are reassigned such that database connections in VM instances need not be reconfigured at all. Because a later failure could strike europe-west3-b, the former primary instances in europe-west3-a serves as the new failover replica once it is up again.

25 Virtualization

In Section 2.3.5 we have learned that a program always runs a process which manages resources for the program (memory in particular). A process runs on a machine with an operating system, which is a kind of runtime environment. In order to stress that process A runs on machine B, we say that A is **guest** of B, which in turn is called **host**.

When **machine**⁵² **virtualization** comes into play, we have even more runtime environments in-between: Processes run in "in-between runtime environments" that run on machines operating systems. So, processes are guests of these in-between runtime environments, which are in turn guests of the machine's operating system.

The purpose of such in-between runtime environments is to **isolate** their guest's process from those of other guests. Isolation means that the processes of different guests cannot interfere with each other via shared resources. Consequently, resources of bare-metal machines are **better utilized / less wasted** if a potentially conflicting process does not run exclusively on a bare-metal machine but along with others yet contained in a guest.

25.1 Kinds of Virtualization

There are different kinds of virtualization technologies that differ in how they work and trade off the qualities of Section 25.2.

25.1.1 Processes

Mundane operating system processes already provide some kind of virtualization: As explained in Section , **processes** are a small-footprint mechanism of operating systems to isolate running programs from each other by means of dedicated virtual memory address spaces (s. Figure 105). This makes it virtually impossible for a process to overwrite the memory of another process by accident.

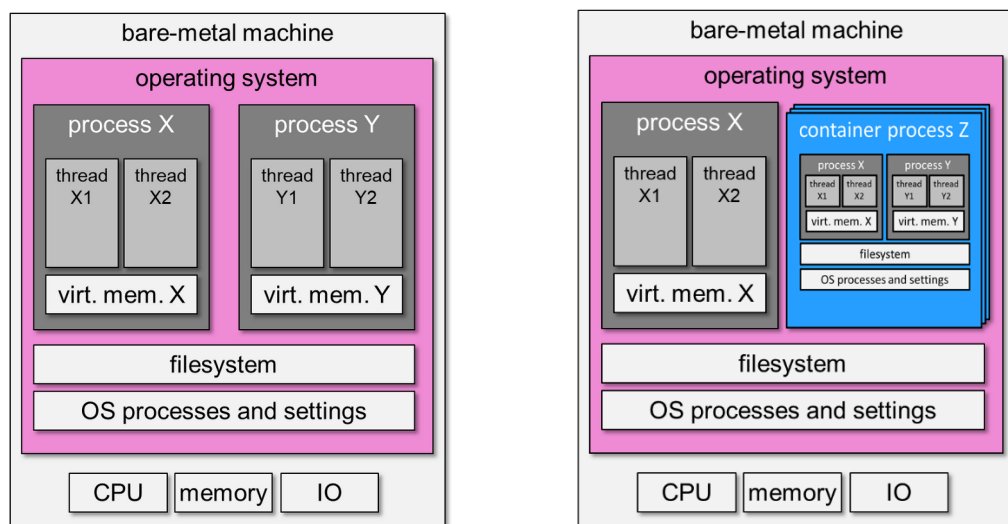


Figure 105: Virtualization by means of pure processes (left) and containers (right)

⁵² There are also other kinds of virtualization like **network virtualization** or **storage virtualization**, which are no subjects of this chapter.

25.1.2 Containers

Container virtualization makes use of Linux Kernel concepts for isolation (s. Chapter 25.3). Basically, a container is a Linux process hierarchy that is provided with an exclusive file system (mounts), hostname, namespace for users, groups and process ids, and network. Moreover, usage of resources like memory and CPU can be limited per container. These resources, however, are still used by means of system calls to the kernel of the container's host operating system.

25.1.3 Virtual Machines

Virtual machines abstract from real bare-metal machine hardware resources like CPUs, memory, and I/O devices. Operating systems can be installed on a virtual machine just like in a bare-metal machine. Ideally, an operating system cannot decide if it is installed on bare-metal hardware or on a virtual machine.

Virtual machines run on **hypervisors**, which provide virtual resources like vCPUs and map accesses to these resources to real hardware resources. There are two types of hypervisors:

A **native hypervisor**, a.k.a. **type-1 hypervisor**, directly runs on bare-metal machines. Figure 106 (left) depicts a typical stack on top of a native hypervisor. The main advantage of native hypervisors is their performance compared to hosted hypervisors. "VMware ESXi" is a widely used instance for native hypervisors.

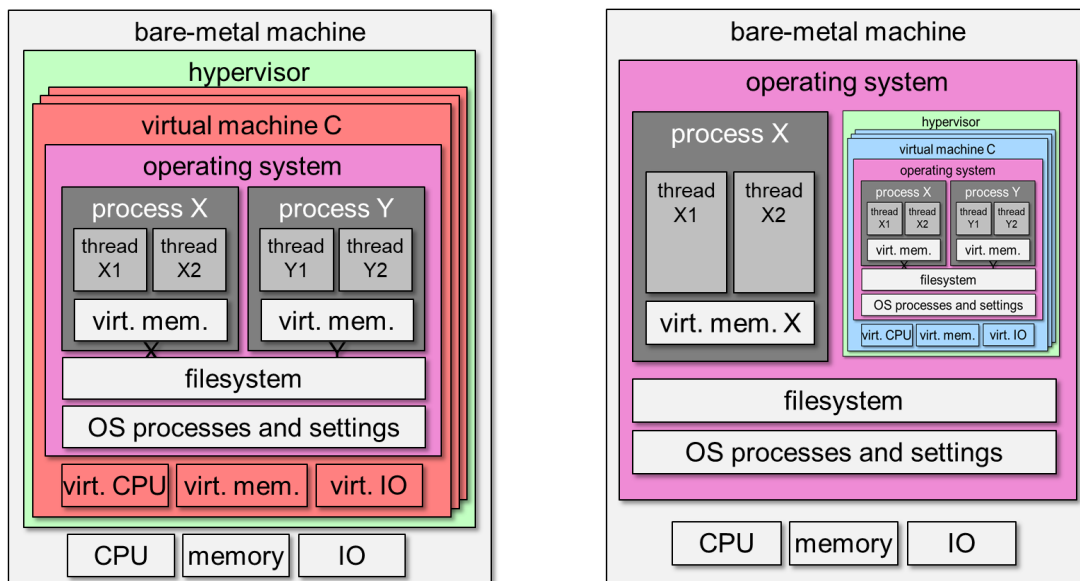


Figure 106: Native hypervisor (type 1, left) and hosted hypervisor (type-2, right)

A **hosted hypervisor**, a.k.a. **type-2 hypervisor** is installed on top of a common host operating system as depicted on the right side of Figure 106. A hosted hypervisor creates a process in that **host** operating system for each virtual machine with its **guest** operating system. Of course, normal processes can still run in the host operating system side-by-side with the hosted hypervisor. This is the main advantage of a hosted hypervisor compared to a native hypervisor. "Oracle VirtualBox" is a popular hosted hypervisor.

25.1.4 Bare-Metal Machines

On the other end of the spectrum, one could allocate a dedicated **bare-metal machine** for each single process⁵³. This would provide a maximum level of isolation between processes but of course a waste of resources.

25.2 Virtualization Technologies Trade-offs

In machine virtualization, one has to trade off the following qualities.

	Isolation	Host Dependence	Waste	Provisioning
Processes	low	very high	small	very fast + easy
Containers	medium	high	medium	fast + easy
Virtual machines	high	low	big	easy
Bare-metal machines	very high	none (since no host)	very big	hard

Table 37: Virtualization technologies overall comparison

25.2.1 Isolation

The evolution of programming languages has yielded concepts like, e.g., namespaces with accessibility rules that particularly allow for isolating parts of a single program from each other. This isolation, e.g., avoids that certain program units can access certain variables and overwrite their values.

Not just variables in a single program might be subject for interference. Multitasking, i.e., concurrently running programs, is common in all contemporary operating systems. Such unsolicited interference between different programs may be due to the following reasons:

- **File System:** Different processes may read and write to the very same directories and files, particularly if they are instances of the same program or instances of different revisions of the same program.
 - **Variable Data:** Two web server processes expect their respective data in `/var/www`
 - **Log Data:** Two web server processes may log to `/var/log/nginx/access.log`
 - **Configuration Data:** Two Nginx web server processes expect their respective configuration file in `/etc/nginx.conf`
 - **System Libraries:** A system library has a standardized path that point to a certain revision of that library.
 - For example, `/lib/x86_64-linux-gnu/libc.so` may link to `/lib/x86_64-linux-gnu/libc.so.6` but some processes may depend on a different revision.
 - Other recurring conflicts include database driver like `/usr/lib/mysql/libmysqlclient.so` or utility libraries like `/lib/x86_64-linux-gnu/libjpeg.so` or `/lib/x86_64-linux-gnu/libssl.so`.

⁵³ Of course, this is a simplified view since one practically cannot prevent common operating systems from starting several processes on their own.

- **Runtime Environments:** Two Java or PHP processes depend on different Java Runtime Environment or PHP CLI revisions but `/usr/bin/java` or `/usr/bin/php` points to just one revision at a time.
- **Environment Variables:** Processes may expect different values of the same environment variable. To name some common suspects:
 - `PATH` might include `/opt/git-2.20/bin` but a process expects a new version of Git.
 - `JAVA_HOME` might point to the wrong Java Runtime Environment; same for `PYTHONPATH`, `PERL5LIB`, `RUBYLIB`
 - `LANG` and `LC_*` configure the current locale and character encoding, e.g., `LANG=de_DE.UTF-8`
- **Port Mappings:** A target port can only be mapped to a single process. Yet, protocols have a certain default target port. Process that expose web UIs or APIs use HTTP(S). Thus, HTTP(S) default ports and development default ports 80, 443, 1443, 3000, 8080 are likely to be mapped multiple times yielding a conflict.
- **User / Groups:** Two instances of the same program protect their data by restricting file permissions to a certain user or group but use the very same user or group for that so they cannot protect the data from each other.

Such conflicts or interference cannot be avoided by means of programming languages concepts. Here, isolation must care for sharing and managing resources in a fair manner, such that processes get the "illusion" of complete access to these resources although they actually are limited and compete with others.

Besides rather inadvertent conflicts or interference, there is also a security aspect: Isolation also protects programs and data from intentional (malicious) executions and accesses, respectively.

As Table 37 depicts, **processes** are rated comparably "low" with respect to isolation as just have their own virtual memory address space. Moreover, processes are not limited in resource consumption.

Containers additionally have their own root file system, hostname, users, groups and process id namespaces, network interfaces (thus port number namespaces) as well as hostnames which lowers the risk of conflicts here. However, containers run in protection ring 3 (user mode) while virtual machines run in protection ring 0 (kernel mode) [116]. Therefore, they are particularly susceptible to kernel vulnerabilities like "Dirty COW" [117] that break isolation of user land processes.

Virtual machines almost provide the same degree of isolation as bare-metal machines do. However, virtual resources must finally be mapped to hardware resources and thus compete. For example, a process in one virtual machine could consume an unproportionable big share of the host's network interface maximum throughput. Processes in other virtual machines on the same host might suffer from this so-called **noisy neighbor**.

25.2.2 Host Dependence

Machine virtualization requires a host machine. Such host machines may constrain the flexibility for machine virtualization, i.e., what might actually run in a virtualized machine.

In containers, processes must be compatible with the Linux Kernel of the container's host. Processes in a virtual machine must be compatible with the operating system running in that virtual machine but not necessarily with the operating system of the virtual machine's host.

25.2.3 Waste

Waste of resources basically occurs due to two factors.

- **Redundancy**: Different kinds of virtualizations require different extends of data duplication and therefore waste of storage and memory.
- **Reservation**: Different kinds of virtualizations allocate resources in advance due to efficiency reasons. These are then mostly left unused but reserved and blocked from being used for other purposes. This is particularly true for memory and storage. Conversely, **utilization** of resources is the opposite of wasting resources due to premature allocation. Cloud providers in particular are interested in avoiding such kinds of waste in their bare-metal machines.

Processes might waste memory if the respective programs allocate memory (by means of, e.g., `malloc()`) without actually using it. However, this is somewhat inevitable and up to the developer to prevent.

Since **containers** are just process hierarchies, they are comparable to mere processes with respect to waste. Just if the layering as explained in Section 25.3.3b) is not used properly, containers might produce redundant copies of files in the host's file system.

During start-up, a **virtual machine** allocates a certain amount of memory, e.g., 4 gigabytes, independently of what the processes running in that virtual machine allocate in sum from that virtual machine. Modern virtual machines allow for certain dynamic adaptations of the allocated memory, but these are usually bounded by certain limits that have to be guessed and configured in advance (minimum memory, startup memory, maximum memory). Moreover, such dynamic adaptations lower performance. The same is true for storage: Virtual machines have to allocate storage in advance in the host for their mounted filesystems independently how much space in these filesystems is actually used. Moreover, the boot disk image of each virtual machine carries a whole copy of an operating system with all its files. This results in a storage footprint in the gigabytes for each single virtual machine boot disk image.

25.2.4 Provisioning

Given that the level of isolation suffices, different virtualization techniques differ in how fast and easy a program can be provided with a run-time environment.

Of course, creating a **process** for a program in a ready-to-use operating system is much faster and easier than ordering and install and configure a dedicated **bare-metal machine**. Moreover, practice shows that it can be hard to uphold the immutability of bare-metal machine and, mainly due to lack of thorough documentation, reproduce a certain configuration, e.g., for horizontal scaling or for setting up an identical system cluster as an additional deployment stage in a continuous delivery pipeline. Machines that have such a unique and virtually irreproducible configuration are often **called snowflake machines**.

The small storage footprint leverages ecosystems of freely available **container** images. Moreover, container images are small enough to be versionable and transferrable deployment artifacts. It is faster and easier to provision a container than a virtual machine, let alone a bare-metal machine.

As demonstrated in Section 24.9.8a), provisioning of identical **virtual machines** by means of virtual machine images is quite easy but neither fast nor slow.

25.3 Container-based Virtualization

Containers are another virtualization technology. With respect to the trade-offs in Table 37, containers are a compromise between OS processes and virtual machines. Containers are hierarchies of processes running in the host's operating system, which are isolated from each other and whose lifecycle is managed by a container engine (cf. Section 25.1.2).

25.3.1 Isolation via Namespaces

By means of Linux [namespaces](#) [118], processes in different containers are much more isolated from each other than those within the same container. Namespaces establish a view for processes in a container on certain resources and settings like the process list, users, mounts, hostname that is different from each other and that of the host.

Namespace	Purpose
inter-process communication (IPC)	Restricts communications of processes in separate namespaces
process identifier (PID)	Processes that are visible in a container are just a subset of processes running on the host. Moreover, the host's process identifiers (PID) of these processes are mapped to PIDs starting from 0 within the container. Consequently, every container constitutes a namespace for environment variables, which are nothing but key-value pairs. If a process in container sets, e.g., <code>JAVA_HOME=/usr/bin/java</code> , processes of other containers cannot see this environment variable setting. Moreover, environment variables that are exported by a shell which starts a container are not exported into that container. (This could be expected since exported environment variables are normally visible by child processes which a shell spawns.) Instead, they have to be set explicitly on container startup. The only environment variables that are defined by the container runtime in a new container are <code>HOSTNAME</code> , <code>HOME</code> , and <code>PATH</code> .
user (USER)	Like for PIDs, a container might be provided with just a subset of the host's users and groups. Likewise, the respective user ids (UID) and group ids (GID) are mapped to different ids which a process in a container sees. In particular, one can map the root user of a container to a non-root user in the host.
mount (MNT)	Mount points are set namespace-wide and not system-wide. In combination with a <code>pivot_root</code> system call (change root), each container can be assigned a different root directory.
network (NET)	Network interfaces are visible just in one network namespace at some point in time but can be moved between namespaces. Virtual network interfaces pairs in two different namespaces can be used to create a bridge to physical interfaces in a certain namespace. Each network namespace has its own network stack with a private IP address, routing tables, firewall rules etc.
UTS (UTS)	A UTS namespace contains certain properties (observable via Linux command <code>uname -a</code>) which particularly includes the hostname.
control groups (CGROUPS)	Isolates subhierarchies in cgroup hierarchies from each other. Processes within a CGROUP namespace cannot collectively exceed the resources limits that have been set from outside.

Table 38: Linux namespaces

25.3.2 Resource Sharing with Cgroups

So, namespace mostly control *what processes see and can access*. Containers also make use of Linux [control groups \(cgroups\)](#) [119]. These are used to justly apportion *how much processes can consume*, i.e., the allocation of certain resources like memory, CPU, or network I/O.

The connection between cgroups and the CGROUP namespaces of Table 38 is that by isolating cgroup subhierarchies into namespaces, processes in a single namespace must share resources assigned to that namespace and cannot "steal" from other namespaces.

EXAMPLE 108: MEMORY LIMITING CONTAINERS WITH CGROUPS

On its creation, one can set the memory limit for a Docker container. In the host's filesystem, this limit is stored in the file `/sys/fs/cgroup/docker/$CID/memory/memory.limits_in_bytes`. In the container (with some container id CID), the same file appears at `/sys/fs/cgroup/memory/memory.limits_in_bytes`. So, the limits for docker process group but also other Docker containers are out of reach for the container as these are just not visible in the container's file system.

25.3.3 Docker

a) Basic Docker Concepts

In Figure 107, we see basic concepts of Docker. The Docker engine is a set of processes which run on a Linux based host. [Containers](#) are created based on [images](#) which are local to the host. Missing images can be downloaded from remote [registries](#). The download or creation of images or containers is done by means of a command line tool [Docker CLI](#), which can be executed on a different machine than the host.

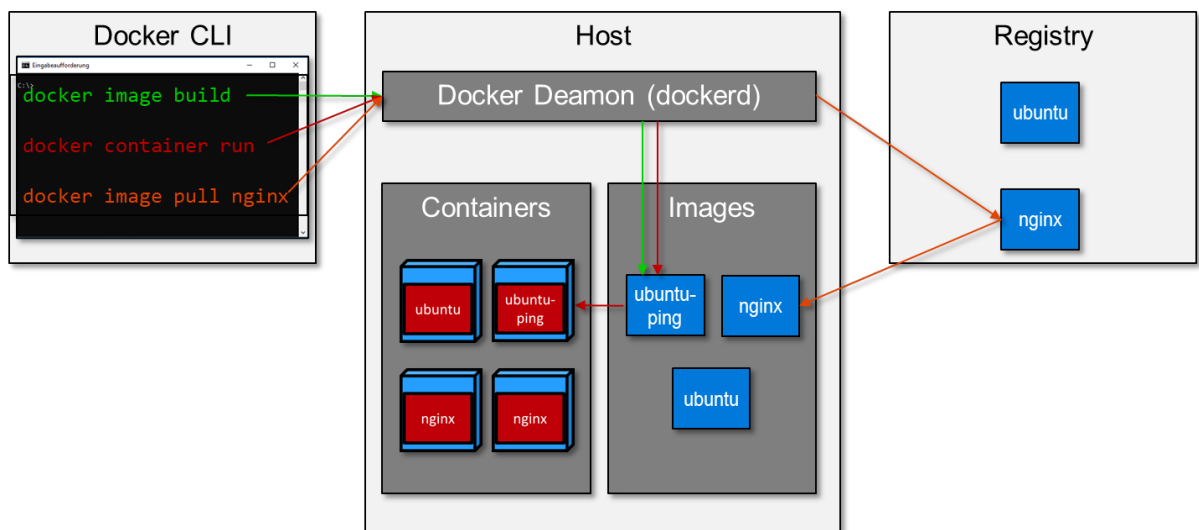


Figure 107: Basic docker concepts

b) Layers

Images as specified in [120] are comprised of linearly prioritized [layers](#). Each layer contains a partial file system, i.e., files and directories. A container cannot access these partial filesystems directly but a filesystem that results from merging the layers. In case of a conflict, i.e., if two layers have each a file with the same path and filename but different content, the layer with the higher priority wins. These kinds of file systems are also called [union file systems \(UnionFS\)](#).

EXAMPLE 109: UNION FILE SYSTEM

In Figure 108, both layers have a file `/dirA/fileA2`. In the resulting **effective file system**, the file is taken from layer 2 due to its higher priority 2.

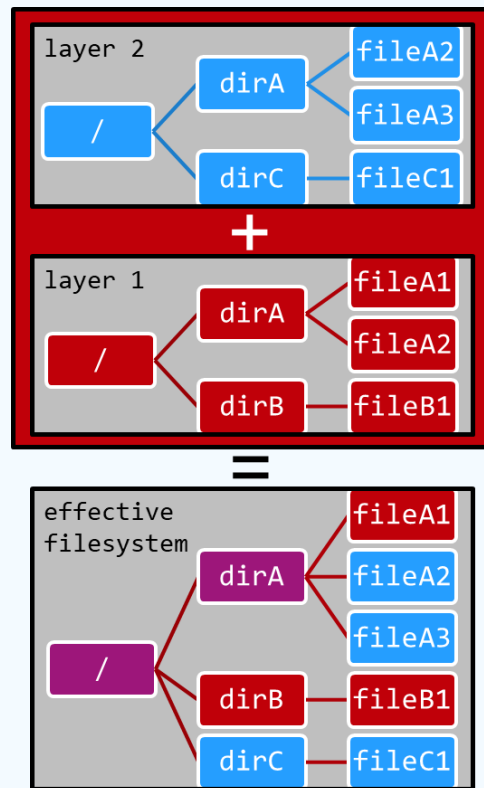


Figure 108: Layer example

Writes in the resulting filesystem are redirected to another layer with highest priority, called **thin-read-write layer**. Modifications to an existing file result in a **copy-on-write** of that file into the thin-read-write layer.

c) Images

Images contain lists of read-only layers. In practice, images are not entirely different from each other: They often use the same layers which contain libraries and executables of operating systems or middleware. Since each layer need to be transferred to and stored once per docker host, this leads to a much smaller storage footprint compared to virtual machine images. Layers and images can be uniquely identified by hash values, which are computed based on a layer's contents.

d) Dockerfiles

Dockerfiles contain instructions for the Docker CLI that specify how to build a new image. The `FROM` instruction basically includes all layers from another image. In the example of Figure 109, layers of the image `ubuntu:18.04` are included. (This figure depicts the layers and attributes of an image together with the Dockerfile instructions that yield layers.) Other instructions like `RUN` create new layers by executing Linux commands like `apt-get` or `copy`. In the example of Figure 109 (left), the two additional layers add the `ping` command to the commands already included in the `ubuntu:18.04` image resulting in a new image `ubuntu-ping:1.0`.

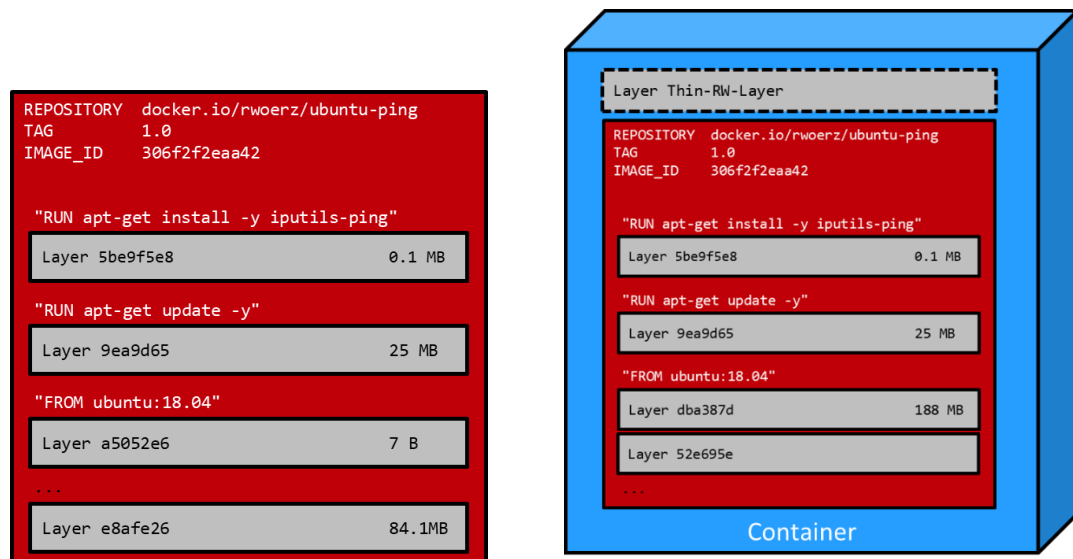


Figure 109: Example image with Dockerfile instructions (left) and its container instantiation (right)

e) Tags, Repositories and Registries

Like layers, images can be uniquely identified by their SHA-256 hash value. Since these identifiers are not quite mnemonic, one can identify images by `<repository>:<tag>`. A **repository** contains one or more revisions of an image, and a **tag** identifies a specific revision like `1.0`. If just `<repository>` is provided, `<tag>` defaults to the value `"latest"`. A value for `<repository>` is not just an identifier but also a locator for a repository. It consists of the following segments:

`<registryHost>:<registryPort>/<namespace>/<name>`.

Here, `<registryHost>:<registryPort>` identify a server called **registry** that stores images. It is optional and defaults to `docker.io:443`. An optional **namespace**—totally different from the namespace of Section 25.3.1—is a means to group different repositories in a single registry and is often equal to a username in a public registry. The **name** is mandatory⁵⁴.

EXAMPLE 110: IMAGE TAGS AS LOCATORS

For example, and putting it all together, a certain image might be identified and located by `localhost:5000/rwoerzbe/ubuntu-ping:1.0` which means that `dockerd` would pull it from registry `localhost:5000` within the namespace `rwoerzbe` and name `ubuntu-ping` with tag `1.0`

f) Containers

Just like for virtual machines, a Docker **container** can be created based on a docker image. At the bottom line, a container is just a hierarchy of user space processes that run on the docker host and to system calls to the docker host's kernel⁵⁵ like any other process.

What makes processes in a container different from other processes is that they are isolated from processes outside that hierarchy by means of namespaces and cgroups (cf. Section 25.3).

⁵⁴ Unfortunately, Docker's terminology around registries, repositories, names, namespaces, and tags is flawed even in the official Docker documentation. That is why the `docker` command `tag` sets a `<repository>:<tag>` instead of just a `<tag>`. Moreover, `<repository>` is sometime referred to as `<image>` or `<name>`.

⁵⁵ This is an important difference from virtual machines. Since images usually are based on some Linux derivative image, one might think that each container runs its own kernel. Those Linux images, however, are rather included in order to provide Linux tools like `bash` and libraries.

Figure 109 (right) exemplifies the instantiation of the image from Figure 109. Since processes in a container might write to files, each container has a thin read-write layer. Writes to this thin read-write layer get lost each time the container is removed.

g) Docker Processes

Actually, containerized processes are not created and treated like normal host processes. Docker has a bunch of processes with certain responsibilities:

The Docker Daemon `dockerd` exposes an API via a local UNIX socket (`/var/run/docker.sock`) for remote clients via TCP (default port 2375). If necessary, the `docker-proxy` maps ports of the docker host to ports which a container exposes. In a single host, all container lifecycles are managed by `containerd` [121]. This includes "the complete container lifecycle of its host system, from image transfer and storage to container execution and supervision to low-level storage to network attachments and beyond". Here, `containerd` delegates the actual creating, starting, and stopping of containers to the command line tool `runc`. When containers are started by `runc`, it creates a child process named `containerd-shim`. Child processes of `containerd-shim` are exactly those that are considered to be the container processes. Then, `runc` reparents `containerd-shim` to its own parent `containerd`. A `containerd-shim` process keeps the container processes alive after `runc` has exited and even if `containerd` or `dockerd` terminate. Moreover, it reports back container process' (exit) status to `containerd`.

EXAMPLE 111: DOCKER PROCESSES

Figure 110 depicts a cut-out of a Linux process hierarchy within a docker host. We can see three processes which are supposed to be instantiated from a `nginx` image. The leftmost one has an additional `bash` process, e.g., due to an administrator, who has logged into this container via `docker exec`. The container just runs `nginx`. The rightmost container is a container in the making, i.e., `runc` is still running and has not yet re-parented `containerd-shim` to `containerd`.

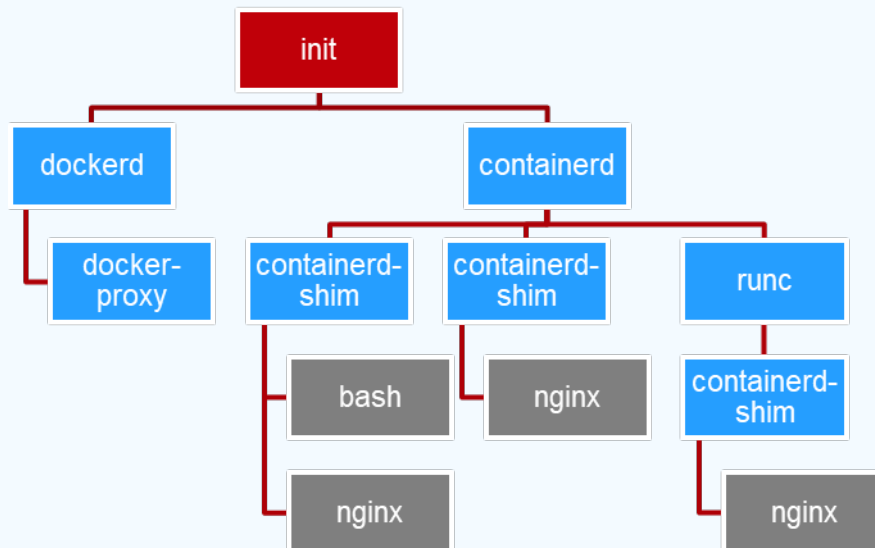


Figure 110: Docker processes

h) Mounts

Files in a Docker container can be stored in four different ways in the filesystem. Figure 111 and Table 39 summarize them.

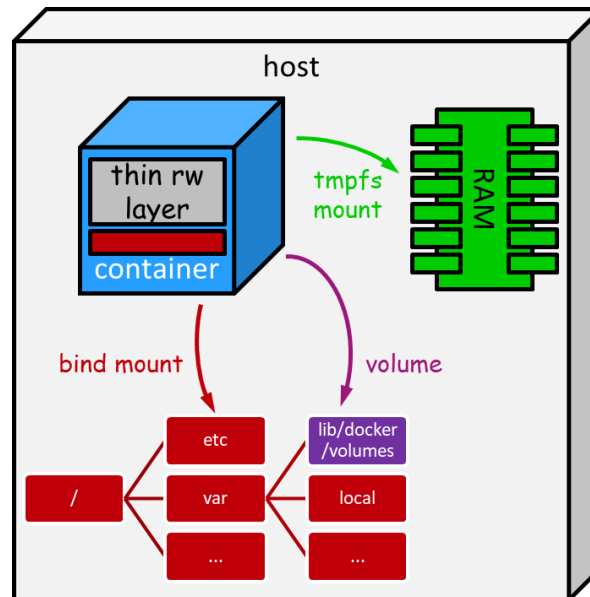


Figure 111: Docker storage options

Storage option	Durability	Sharable	Description
thin read-writable layer	until container removed	no	If a process in a container writes to a file in an arbitrary directory of the container's filesystem, those writes are earthed at a thin read-writable layer owned by that container. Such thin read-writable layers have higher priority than any other layer in a container's image. Existing files from lower layers are copied to this writable layer on write access (copy-on-write). However, a read-writeable layer is bound to its specific container. If the container is removed, so is the thin read-writable layer.
tmpfs mount	until container stopped	no	With tmpfs mounts, one can create memory disks in the host and mount them in a specific directory in the container's filesystem. Of course, writes in these tmpfs mounts are very fast. However, contents of these directories again do not survive a container restart. Moreover, tmpfs mounts are not shareable, i.e., no other container nor the host can access a tmpfs mount.
bind mount	until deleted on host	among containers and host	A bind mount is a directory in the host, which is mirrored as a directory in the filesystem of one or more containers.
volume	until deleted on host	among containers	Like bind mounts, volumes appear just as directories in the container's filesystem. On the host, the volume is directory at a specific location and a specific subdirectory structure, i.e., it is not meant to be accessed by processes on the host. Different volume drivers allow for storing the volume not only on a host's disk but also on other types of storages, e.g., cloud storage.

Table 39: Docker storage options

i) Docker Command Line Interface (CLI)

Like `gc1oud` (cf. Section 24.9.3), Docker can be managed by a single executable `docker`. Docker commands are grouped into commands that revolve around images, containers, and volumes just to name some of them. Figure 112 shows the most important commands. Please refer to the official Docker documentation for an exhaustive documentation about the Docker CLI.

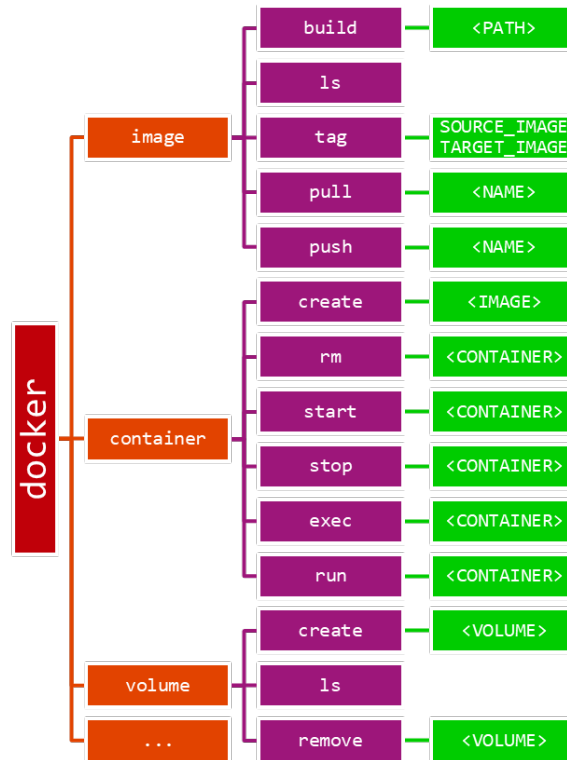


Figure 112: Docker command-line interface

25.4 Container Orchestration

25.4.1 Container Orchestration Features

Based on container-based virtualization, **container orchestration** typically provides the following additional features:

Feature	Description
Optimized resource utilization	Just with container-based virtualization, an administrator has to decide which container should run on which host. Container orchestration automates the scheduling of containers to hosts using some kind of binpacking algorithm and thereby optimizing the overall host utilization.
Self-healing	Just with container-based virtualization, if a host dies, all of its containers die with it forever. With container orchestration, containers are automatically recreated on healthy hosts.
Horizontal scaling	Containers with the same image can be created automatically due to increased workload even in an automatic fashion.
Service discovery	Tiers of containers with the same image serve as endpoints for a so called "service". For clients it would be infeasible to keep track of the actual internal IP addresses of these endpoints. Therefore, container orchestration uses an internal domain name system (DNS) such that services can be assigned with a constant domain name that resolves to a load balancer for the tier or to the change-prone IP addresses of one of the endpoints.
Load balancing	Container orchestration provides load balancing mechanisms in order to distribute workload to containers with the same image.
Rollouts/-backs without	Containers based on the same image can be automatically updated to a new image

Feature	Description
observable downtimes	version one by one. Together with load balancing, the whole tier of containers seem to be completely reliable, i.e., has no downtime that is observable from their clients.
Secret- and configuration-management	Software systems under development normally use distinct clusters for development, tests and production. Configuration data and secrets like password in particular differ between these clusters and thus better be stored and versioned apart from other data.
Storage orchestration	Cloud providers differ in what kinds of storage options they offer (cf. Section 24.9.7) and how these storages can be accesses. Container orchestration provides a layer of abstraction above these cloud provider specific storage options thus making containers independent of a certain provider specific storage option.

Table 40: Container orchestration features

Moreover, cloud orchestration eases the provisioning of certain system resources, like, e.g., identity providers. Therefore, cloud customers can avoid lock-ins into managed PaaS services. Ideally by means of container virtualization and orchestration, systems are portable between clouds of different providers without any adaptation.

25.4.2 Basic Kubernetes Concepts

In 2019, **Kubernetes (k8s)** [122] is the predominant container orchestration tool. Figure 113 depicts Kubernetes' main concepts: Kubernetes manages **Kubernetes clusters** of nodes that are either master nodes or worker nodes.

a) Nodes and Processes

Worker nodes⁵⁶ are (virtual) machines that run system specific processes realizing system resources of Chapter 23. These processes run in containers that are managed by the container runtime **containerd** (cf. Section 25.3.3g). Each container is assigned to exactly one pod. A **pod** is the smallest unit of computation that Kubernetes manages and can contain one or more containers. The **kube-proxy** on a worker node cares for mapping ports that are exposed by containers to exposed ports of the node. A kubelet is a process on each worker node that realizes the control plane of Kubernetes: It communicates with the **apiserver** in order to receive pod lifecycle commands and report back status of the worker node.

Master nodes manage the state of a Kubernetes cluster. The **controller manager** is a process with different **controllers**, which are basically control loops, e.g., program loops that regulate the state of the Kubernetes cluster. It cares for noticing and reacting: If a worker node fails, it creates the desired number of pods, creates and updates endpoints which connects Kubernetes services to pods and manages security tokens. The **kube-scheduler** assigns newly created pods to worker nodes. Administration data like the desired state is persistently stored in the key-value store **etcd**. Communication between a master node and a worker node takes place via the **apiserver**. This is also true for the command line tool **kubectl**, which sends admin commands to master nodes.

⁵⁶ Note that we defined a node to be a process (or coherent set of processes) in a network (cf. Section 2.3.8). In contrast, a Kubernetes worker node is a (virtual) machine.

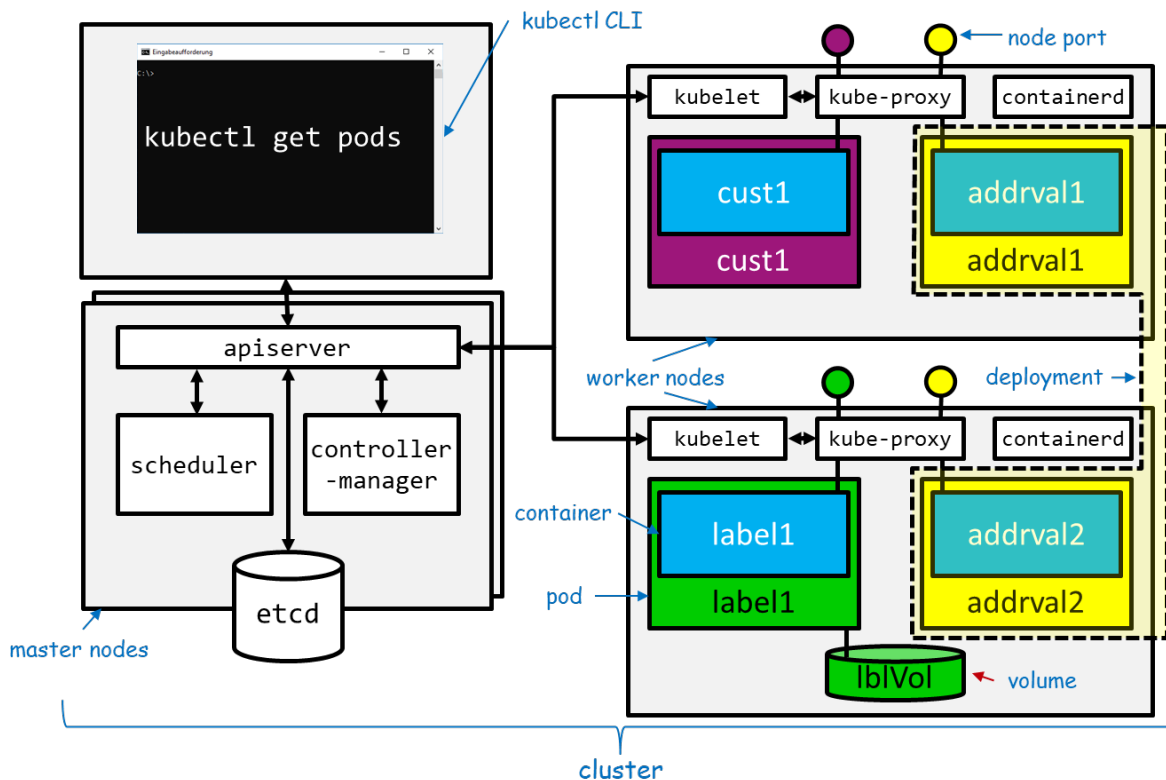


Figure 113: Basic Kubernetes concepts

b) Kubernetes Objects

In Kubernetes the constituents of cluster's state are defined by an object model. Figure 114 depicts a cut-out of the **object meta model** which specifies how objects of different types are related to each other. This is similar to a GCP project which is also **software-defined** and comprised of interrelated resources (cf. Section 24.8).

A Kubernetes **state** is modeled by an instance of this meta model. It conforms to the object meta model and so do Kubernetes **specifications** which define partial object models in the format of "Yet Another Markup Language (YAML)" or "JavaScript Object Notation (JSON)". Moreover, the object meta model is reflected in the commands of the **kubectl** command line interface, which is just a client of the RESTful API that also speaks in terms of the object meta model.

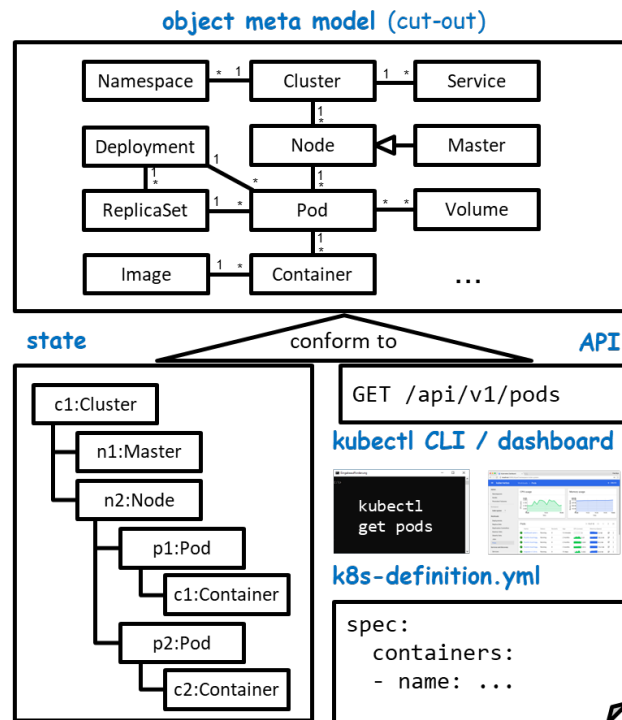


Figure 114: Kubernetes' object model

c) States

In Kubernetes, we talk about three states of a Kubernetes cluster as depicted in Figure 115. A **desired state** is a set of properties, to which an **actual state** is supposed to adhere. Such properties typically are the number of pods with containers of a certain image.

The desired state is modified intentionally via calls to the API, e.g., by means of the kubectl CLI or via autoscaling. If it changes, the scheduler and controller manager cares to make the actual state consistent again with the desired state. If the actual state deviates from the desired state due to a worker node failure, the controller manager realigns the actual state with the desired state by creating the concerned pods on other worker nodes.

Sometimes the **observed state**, i.e., what kubectl or the dashboard outputs, lags a little behind the actual state.

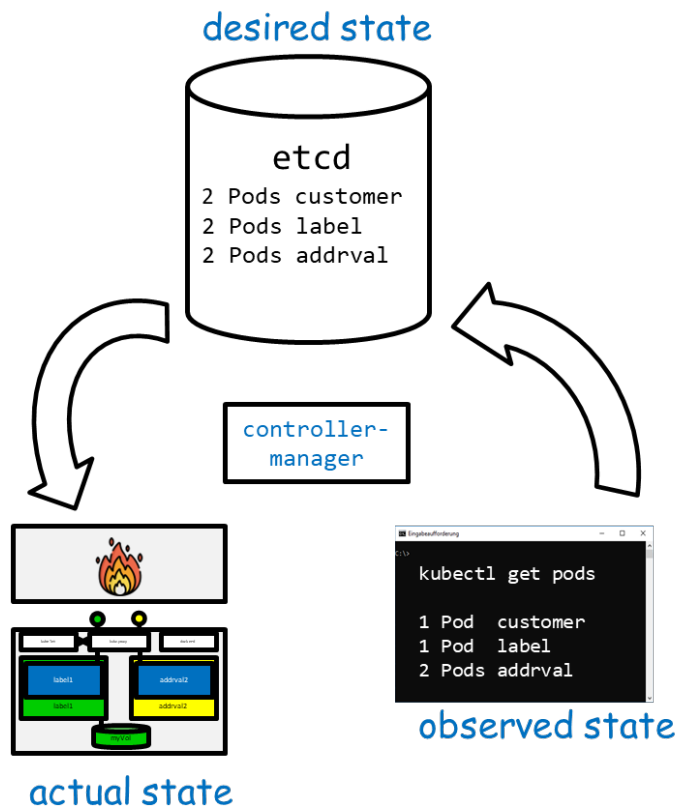


Figure 115: States in Kubernetes

d) Manifests

One can successively build a desired state by repeatedly calling the API via lengthy `kubectl` commands. A better practice is to define the desired state in definition files, a.k.a. **manifests**, which are normally formatted in the Yet Another Markup Language (YAML). Such manifests have sections, each of which define the properties of an object of a certain **kind**, i.e., meta object in the meta object model. **Names** and **labels** can be defined beneath metadata and are relevant for selecting the respective object later. The **spec** is specific to the kind. Normally, there is just a definition for a single object per file. If there are more than one, these are separated by a triple dash ---.

EXAMPLE 112: KUBERNETES POD MANIFEST

```
apiVersion: v1
kind: Pod
metadata:
  name: label
  labels:
    app: parcer-label
spec:
  # definition specific to the "kind"
  ---
apiVersion: v1
kind: Service
metadata:
  ...
spec:
  ...
```

For example, a definition for a deployment could look like the following:

EXAMPLE 113: KUBERNETES DEPLOYMENT MANIFEST

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: addrval-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: addrval
  template:
    metadata:
      labels:
        app: addrval
    spec:
      containers:
        - name: addrval
          image: addrval:1.1
          ports:
            - containerPort: 8083
```

Here, we define a Deployment which has two pods each of which containing a container based on the image `addrval:1.1` and that expose a port `8083`. If this specification is applied by means of `kubectl apply -f filename.yml` then two pods are created on the worker nodes.

25.5 Service Meshes

Service meshes focus on the network traffic in a distributed system. They are typically used as an addition to container orchestration. Service meshes add sidecar containers into each pod, which implement the following cross-cutting features:

Feature	Description
Securing network traffic	A node and its dedicated proxy are local to each other. Their network communication thus can be considered secure. Consequently, a node can offload securing network communication with other nodes to its proxy. This includes traffic encryption and signing but also mutual authentication with TLS (cf. Section 30.3), also called mTLS.
Observability of network traffic	Proxies can log received messages on behalf of the nodes in a standardized way. Such standardized logs then can easier be aggregated, analyzed and displayed.
Routing of network traffic	Proxies can redirect messages. This is particularly useful if a redeployment on their respective node is currently underway. Redeployment strategies (cf. 18.7) or internal load balancing strategies are therefore easier to implement.
Resilience of distributed systems	Proxies can act as circuit breakers as discussed in Section 21.6 and thereby increase the resilience of a distributed system. Proxies in a service mesh can use fault injection , e.g., delay responses or respond with error responses, in order to test configured circuit breakers. Please note that these features mitigate problems that arise only in synchronous network communication (cf. Section 21.4.1). For asynchronous communication and messaging, service meshes are therefore less useful and uncommon.

Table 41: Service Mesh features

In summary, these features leverage strengths of scaled systems, e.g., different redeployment strategies, and mitigate some weaknesses of distributed systems and microservice architectures in particular like insecure network communication.

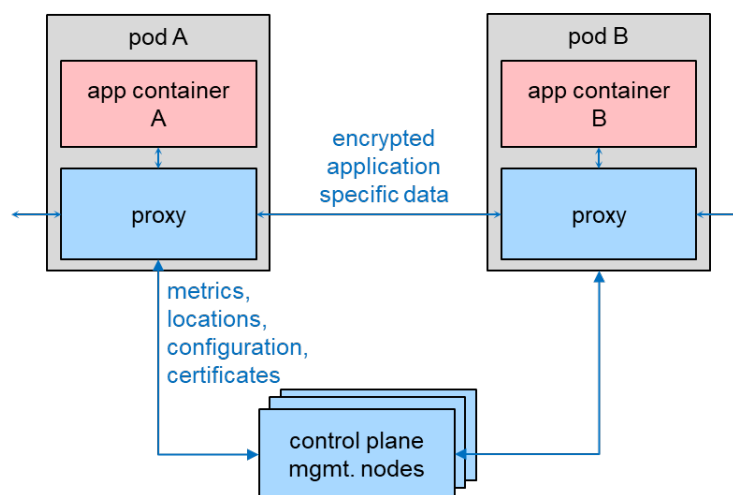


Figure 116: Service mesh concepts

Figure 116 depicts how service meshes conceptually separate network traffic into two distinct so-called [planes](#):

- The [data plane](#) contains all messages that sent and received by non-proxy nodes. These messages contain application specific data.

- The **control plane** refers to the messages between proxies and dedicated service mesh management nodes. These messages contain configuration data for the proxies, certificates for mTLS and the like.

A comparison of concrete service mesh products, like Istio and Linkerd, can be found in [123].

26 Messaging Systems

This chapter is about technologies that support the messaging communication pattern of Section 20.6 and go beyond mere relaying of messaging.

Criteria

Receivers	single	multiple
Retention	delete	retain
Communication	pull	push
Storage	memory	disk
Batch size	time-sliced chunks	single message
Processing	none	complex validation, transformation, augmenting, joining
Horizontal scaling	no	yes, via partitions and replicas

Table 42: Messaging system characteristics

As Table 42 summarizes, different **messaging systems** (a.k.a. **message-oriented middleware (MOM)**) differ in

1. if they allow for **multicasting** a single message to **multiple receivers** or just forward a single message to a **single receiver**,
2. if they **delete** forwarded messages or **retain** them at least for a while, which is particularly useful if receivers are added to a multicast group that need to catch up,
3. if forwarding means that messages are **pushed** to the receivers or **pulled** from the receivers,
4. if they store messages in **memory** or on **disk**,
5. if they accumulate received messages for the duration of a certain time-slice, e.g., one day, into **chunks (batches)** or if they process or forward messages as soon as possible, i.e., keeping the batch size equal to a single message,
6. if they just **forward** messages or allow for **processing** them, e.g., validating, transforming, augmenting them with additional data or joining messages from different streams,
7. if they allow for **horizontal scaling** via partitioning and replication of streams in order to increase throughput and availability.

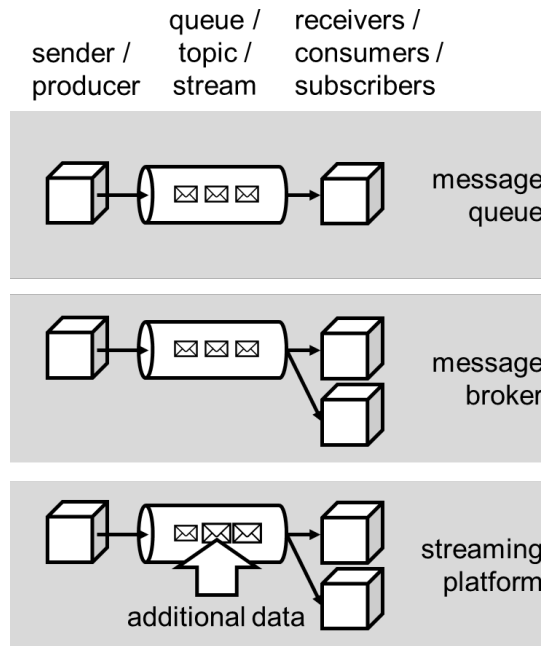


Figure 117: Message queues vs. message brokers vs. streaming platforms

Message queues, message brokers, enterprise service buses (ESB) and streaming platforms are special kinds of messaging systems. **Message brokers** typically differ from **message queues** as they allow for multicasting (cf. list item 1). **Enterprise service buses** differ from message brokers basically in that they additionally allow for processing (cf. list item 6) and **streaming platforms** are enterprise service buses with a focus on scalability (cf. list item 7). Please take this not as a normative delineation of these terms. They are often used interchangeably.

26.1 Message Brokers

Even without processing capabilities, message brokers allow for a wide range of uses cases. (The following mostly summarizes [24, pp. 439–487].)

26.1.1 Load-balancing and Fan-Out

It is common that a single broker has multiple receivers, a.k.a. consumers (cf. list item 1). According to [24, pp. 444–445], one can identify two main patterns that determine which messages are forwarded to which receivers (cf. Figure 118).

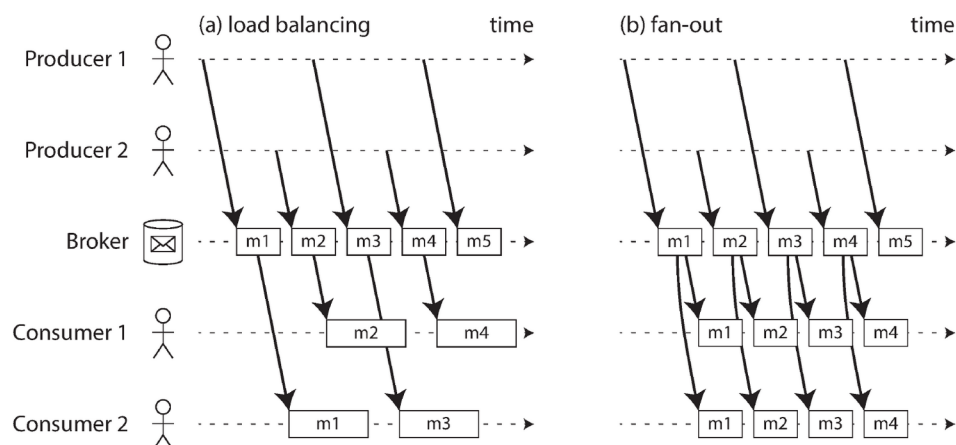


Figure 118: Load balancing vs. fan-out [24, p. 445]

A message broker can be used as a **load balancer** (cf. Chapter 19) with additional capabilities. If processing takes place within the consumers and requires a lot of resources, each message could be forwarded just to one consumer.

If the intend to distribute data, then it is possible to use a **message broker** to **fan out** messages, i.e., forward identical copies of a message to multiple consumers.

26.1.2 Redelivery

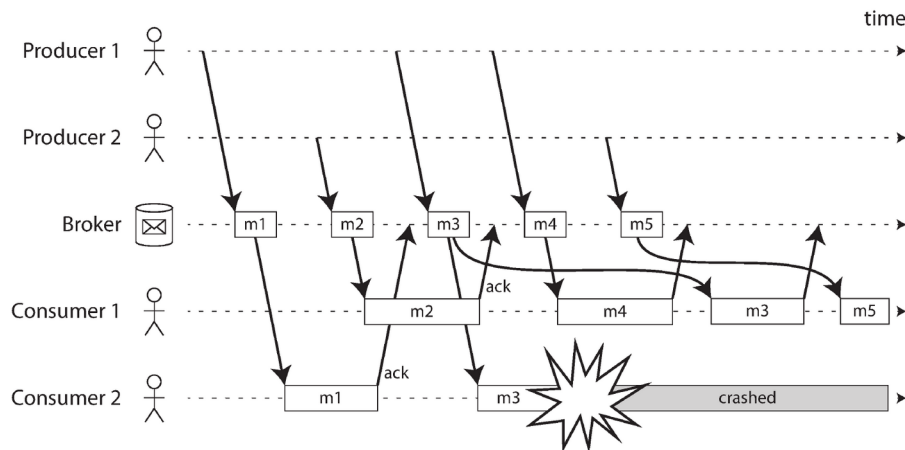


Figure 119: Re-ordering of messages due to consumer failure [24, p. 446]

Problems arise in load balancing if ordering of messages is of importance. In the example of Figure 119, messages are sent before the acknowledgments of a preceding message arrives at the broker due to performance reasons. Consequently, message *m4* is sent to Consumer 1 before the broker notices that *m3* could not be delivered due to a failure of Consumer 1. The broker re-delivers *m3* to the still healthy Consumer 1 at the price that in the ordering of messages on Consumer 1 now *m4* comes before *m3*.

26.1.3 Event Sourcing

Often, messages basically contain **event data**. E.g., if a relevant datum in the sender changes, the sender emits a message that reflects the event (timestamp, original value, actual value, etc.)

EXAMPLE 114: EVENT DATA IN ReWo-RATE

ReWo-Rate might emit a message each time a rating is created or updated.

Architectures that extensively use messages containing event data for communication between systems or microservices are also called **event-driven architectures**.

Event sourcing is an architectural pattern for **Domain-driven design** and **Event-driven architectures**. It promotes storing not just the current state of an application but also its entire change history in an **event log** comprising all events that led to changes in the stored entities (data records). Basically, an event is a data record containing timestamps of the event's occurrence and recording, the "before" data value that is supposed to be changed and the "after" data value after the change.

EXAMPLE 115: EVENT SOURING IN ReWo-RATE

In ReWo-Rate, ratings pose change-prone entities. Using event sourcing, changes to ratings can be recorded, audited, and replayed if needed.

Of course, a raw stream is unsuitable for certain use cases like complex queries concerning the current application state. However, streams can be used as input for database, e.g., a consumer's job could be transforming events into data manipulation queries to some database. After all, a database binary log (cf. Section 24.9.9) is some kind of log in a log-based messaging system with the restriction to database (replicas) as producers and consumers.

26.2 Streaming Platforms

Streams are ubiquitous in computing. A **stream** is a data structure with the following characteristics:

- A stream is considered to be a quasi-infinite array of data records, often considered to be messages or events.
 - It is natural to start processing of existing data records if they are still being added to the stream.
 - If all data records are there before processing starts but the overall batch of data records is too large to fit in memory entirely, i.e., in the terabytes or petabytes, the batch is considered to be a stream that is loaded from storage into memory and processed part by part.
- Due to their quasi-infinite size, handling streams comes along with some limitations. In general, there is no random access to the entirety of a stream. At least a consumer can read the latest data record on a stream and at best some of the latest data records.

EXAMPLE 116: STREAMS IN JAVA

Java provides foundation classes for handling streams since its inception. Since version 8 streams became a first-class citizen in Java as they suit lambda functions quite well. The following outputs squared values of values that come from a pseudo stream ("pseudo" because of its obvious finiteness).

```
int[] numberArray = {1,2,3,4,5};
IntStream pseudoStream = Arrays.stream(numberArray);
pseudoStream.forEach(n -> System.out.println(n*n));
```

Instead of being constructed from a finite array, a Java stream object can also wrap inputs from an I/O device (e.g., disk or network).

Streaming platforms like Apache Kafka [124] are modern messaging systems with can be characterized according to the list at the beginning of this chapter as follows: They (1) allow for multicasting messages, (2) retain messages, (3) lets consumers pull messages, (4) store messages on disk, (5) are stream-oriented, (6) allow for processing (called "**stream processing**"), and (7) allow for horizontal scaling. Particularly (6) gave rise to the new term streaming platform.

If we add processing capabilities to stream-oriented messaging, new use cases emerge.

26.2.1 Complex Event Processing

Complex event processing systems (CEP) have persistent queries in place that match on event in streams and emit other complex events in case of a match. CEPs are used for, e.g., fraud detection in credit card usage event streams, detecting trends in stock markets event streams or detecting (upcoming) machine malfunction of mechanical machines from sensor data.

26.2.2 Event Stream Processing

Event stream processors (ESP) enrich, transform, filter and aggregate event data. For example, one could consider incoming emails as a stream. Such emails could be enriched by contact data of

the sender thereby making the stream of emails more "valuable". "Real-time" analytics is another common use case: For example, user interactions (clicks) with a website could be considered a stream. An ESP could count the clicks on certain buttons like the checkout button. If this number goes down (due to a buggy release) this can be detected timely by help of an ESP. Post-hoc analytic requests to a data warehouse on day after the event would not be as helpful.

26.3 Scaling Messaging Systems

One way to implement a message broker is to use **log-based messaging**, which uses **log files** that are distributed. Produced messages are sequentially and persistently appended to these log files. Consumers can consume messages by sequentially⁵⁷ reading⁵⁸ from these files. In particular, new consumers or consumers that just have recovered from a failure can still catch up by reading not just the most recent messages but a certain **offset** before the most recent one.

Although consumers read from the same log files, they do not interfere with each other as data is not destroyed by consumption⁵⁹. This makes log-based messaging particularly suitable for fan-out of messages. Eventually, even log files are purged regularly, i.e., messages that are smaller than the minimal offset of all partition's consumers can be deleted safely. In rare cases, consumers cannot keep pace with producers, e.g., if they suffer from frequent downtimes. Since messages are appended to files on disk in log-based messaging in a ring buffer, disk space might run out in these cases and data gets lost.

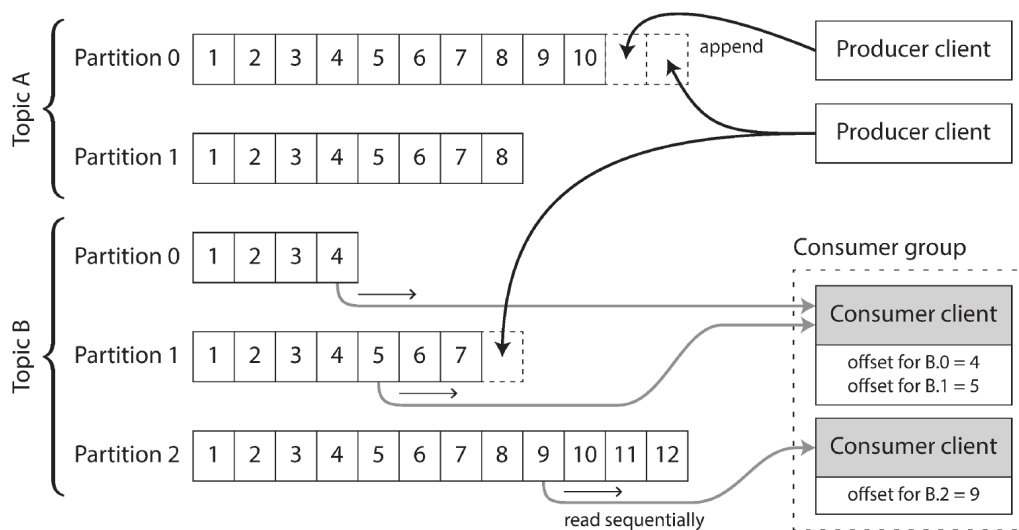


Figure 120: Partitioned logs [24, p. 448]

Load balancing is leveraged by splitting up a single stream (topic) into several **partitions** (cf. Figure 120). These are again just files and are at best distributed among several message brokers. Each partition can be read by several consumers, and each consumer can read several partitions. Consumers store the last read message in the current offset.

⁵⁷ Please note that sequential disk IO has a much higher throughput than random disk IO with many disk seeks and is comparable to random memory access [125, p. 6].

⁵⁸ Since consumers are distributed and remote to the log files, they actually do not access these files directly but consume network messages sent to them which contain log file records.

⁵⁹ Even log files are purged eventually and regularly according to some retention strategy. But this is merely an optional housekeeping job and normally affects only those messages that have been consumed long before.

IV. Web Technology

27 Web Application Layer Protocols

It is important to know, what to run on nodes and how to do that in a cloud environment (cf. Chapter 23) and how to virtualize nodes (cf. Chapter 25). For large scale systems that are on the World Wide Web it is also important to know how to expose nodes properly, e.g., how to shape their interfaces and what communication protocol serves which purpose. In this chapter, we outline important properties and terms of HTTP (cf. Section 27.1), its proper “RESTful” use for web application (cf. Section 27.1.9), and protocols that are based on HTTP (cf. Sections 27.2ff).

27.1 Hypertext Transfer Protocol

In a distributed system, quality attributes like maintainability, reliability and performance do not only depend on what programs run in nodes but also how nodes communicate. This is particularly true for client-server systems, where clients (a) might be developed by independent parties, e.g., customers of the company that implements the server-side, and (b) are connected to the server-side via the internet, which is notoriously unreliable, insecure, and susceptible to high latencies depending on the network proximity of the client and the server.

Using the right protocol properly and designing application programming interfaces (API) in a proper way is crucial especially for large systems. With “protocols”, we mean protocols on the ISO/OSI application layer [11, p. 32] and their influence on architectural styles. Lower level protocols like Ethernet are not being discussed.

As we have seen in Section 5.1, there are different types of workloads. Web application workloads are usually conveyed by means of the Hypertext Transfer Protocol (HTTP) as specified in [85].

27.1.1 Resources

HTTP is about transferring data records in request and response messages

- which are called **web resources** (or just resources),
- which are **distributed** over the so-called World Wide Web, and
- which are identified, addressable, and can refer to each other by **uniform resource locators**.

Web resources are abstract entities. They might be anything, e.g.,

- a geographic location or
- some record in a database or
- a program that converts a number to its square number.

As abstract entities, resources need a concrete **representation** that can be embedded in a message. Technically, a concrete representation adheres to a certain encoding and data format. Standardized data formats are called **media types** (formerly: MIME type) and are identified by values like `text/html`, `text/xml`, `application/pdf`, `text/javascript`, `text/css`, `application/octet-stream`.

EXAMPLE 117: MEDIA TYPES

Here are two HTTP responses (with some omission) that carry the same payload but with different media type representations.

```
Content-Type: application/json; charset=utf-8
```

```
{  
  "name": "Jane Doe",  
  "email": "jane@example.com",  
  "skills": ["Python", "Web Development", "Data Analysis"]  
}
```

```
Content-Type: application/xml; charset=utf-8
```

```
<?xml version="1.0" encoding="UTF-8"?>  
<user>  
  <name>Jane Doe</name>  
  <email>jane@example.com</email>  
  <skills>  
    <skill>Python</skill>  
    <skill>Web Development</skill>  
    <skill>Data Analysis</skill>  
  </skills>  
</user>
```

27.1.2 Request and Response Messages

Request and response messages have two parts: a mandatory **header** and an optional **body**. The header contains numerous lines which are mostly key-value pairs which constitute meta data. Particularly, a server can infer an **effective request URI** from the header. The body contains the actual representation of a resource.

27.1.3 Uniform Resource Identifiers

A **uniform resource identifier (URI)** [126] is a character string that uniquely identifies a web resource like the examples below.

EXAMPLE 118: UNIFORM RESOURCE IDENTIFIERS (URI)

```
ftp://ftp.is.co.za/rfc/rfc1808.txt  
http://www.ietf.org/rfc/rfc2396.txt  
ldap://[2001:db8::7]/c=GB?objectClass=one  
mailto:John.Doe@example.com  
news:comp.infosystems.www.servers.unix  
tel:+1-816-555-1212  
telnet://192.0.2.16:80/  
urn:oasis:names:specification:docbook:dtd:xml:4.1.2
```

A **uniform resource locator (URL)** is a URI that is also an address, i.e., contains information about how to look up a certain resource. It contains multiple segments as shown in Figure 121.

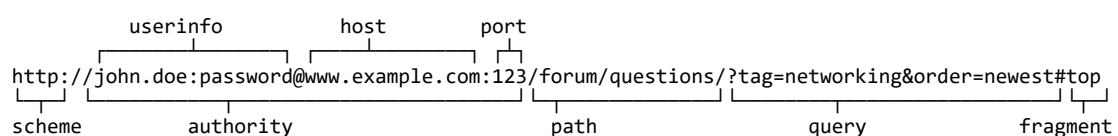


Figure 121: The Uniform Resource Locator (URL)

As the name suggests, the path segment reflects a path in a resource hierarchy. Modeling a decent resource hierarchy is a virtue on its own: Such a hierarchy has to suit a client developer's mental model (cf. Section 10.1.2c)). Moreover, paths are often used as subjects for pattern matching in authorization frameworks (cf. Section 30.4.5e).

EXAMPLE 119: URL PATH

For example, end-users with a role ADMIN would be authorized to access all resources "under" a resource admin, e.g., with a path prefix /admin/.

EXAMPLE 120: URL QUERY PARAMETER UTM_SOURCE

The query parameter utm_source is often used in URL that are links to let the target site know about where the user came from, i.e. a link embedded in a web page with origin originsite.com would look like `https://targetsite.com/path/to/resource?utm_source=originsite.com`

27.1.4 Request Methods

Each request has a mandatory **request line** in its header. In this request line, the **request method** (sometimes called **verb**) is a mandatory segment.

Some request methods are **idempotent**, that is, in a sequence of identical requests with such methods just the first request modifies a resource but not the subsequent requests—provided that there are no concurrent and modifying requests to the same resource.

EXAMPLE 121: HTTP PUT REQUEST METHOD

The PUT method often updates a resource identified by an URI. Subsequent identical updates via PUT do not alter the resource anymore.

Some idempotent methods are said to be even **safe**, i.e., they do not modify resources at all thus do not modify the server-side state of a web application. It is particularly safe to do a second identical request with such methods if the first had no response (timeout). Safety implies idempotency but not the other way around.

EXAMPLE 122: HTTP GET REQUEST METHOD

The GET method merely reads a resource but does not modify it.

A method is said to be **cacheable** if the respective response might be served from a cache in a subsequent (GET) request.

Method	Safe	Idem- potent	Cache- able	Description
GET	s		c	Used to transfer a resource representation to the client. The following prints out the response for the given resource. <pre>curl -X GET http://httpd.apache.org/docs/2.4/en/license.html -is</pre>
POST		(c)		Requests the server to process the resource representation enclosed in the POST request body. The kind of processing is identified by the request URI. A common "kind of processing" is just to add a resource to an existing collection of resources. The request body carries the resource representation to add, the request URI identifies the collection, and the response body might return the resource representation with additional attributes like an internal ID. The following adds a resource to a collection <pre>curl -X POST -d "key=v" https://ptsv2.com/t/1zvfr-1561032323/post</pre>

Method	Safe	Idem- potent	Cache- able	Description
				-is
PUT		i		Requests to create or, if already existent, replace a resource targeted by the request URI by the resource represented in the request's body.
PATCH				PATCH requests modify an existing resource that is targeted by the request URI. The request body contains a representation of the change that should be applied to the respective resource but not an entire representation of the new resource version as a whole. The PATCH HTTP method is not part of the original HTTP semantics specification but an addendum specified in [127]
DELETE		i		Requests the origin server to unmap the request URI the respective resource. From a client's point of view, the resource is then inaccessible by means of that URI. However, it is up to the origin server how to implement this unmapping.
HEAD	s		c	Identical to GET except the body in the response is omitted. curl -X HEAD http://httpd.apache.org/docs/2.4/en/license.html -is
TRACE	s	i		Just echoes a resource representation in a request in the respective response. This method is for mere debugging purposes since the response also contains potential (header) modifications made by intermediate proxy servers. This method is often disabled due to security reasons.
OPTIONS	s	i		Response options and requirements (as key-value pairs) that are available and have to be considered by a client when dealing with a resource but not the response (in some representation) itself. In particular, valid request methods are included in the response. The following command line yields a response for the given resource curl -X OPTIONS http://httpd.apache.org/docs/2.4/en/license.html -i
CONNECT				Used to tell a proxy to establish a TCP/IP connection with the given server and forward any subsequent TCP packets with the server IP to that server. This is particularly useful if a client is aware that it communicates with a server via an outbound proxy and the communication is supposed to be TLS encrypted (https).

Table 43: HTTP request methods (cf. [85, Ch. 9])

27.1.5 HTTP Caching

In the World Wide Web, client and server may be up to 20 000 km apart and have network routes of limited capacity. Therefore, response time (cf. Section 5.2) is an intrinsic problem which can be mitigated using caches (cf. Section 23.9). However, caching itself has an intrinsic problem, too: One has to trade off the benefits, i.e., reduced response times, with the downsides, i.e., a higher risk for inconsistencies due to stale caches and security issues due to unauthorized use of cache contents.

HTTP has built-in support for web caching [99]. Conceptually, a **HTTP cache** (a.k.a. web cache) is a local store for HTTP response messages that came from another cache or the **origin server**. This local store is attached to a server that controls the storage, retrieval, and eviction of such response messages. A HTTP cache is either a **shared cache** that serves responses for multiple clients or a

private cache that serves responses just for a single user of a single client. Browsers⁶⁰ are the predominant clients for HTTP. They have a private cache for each of their (operating system) users.

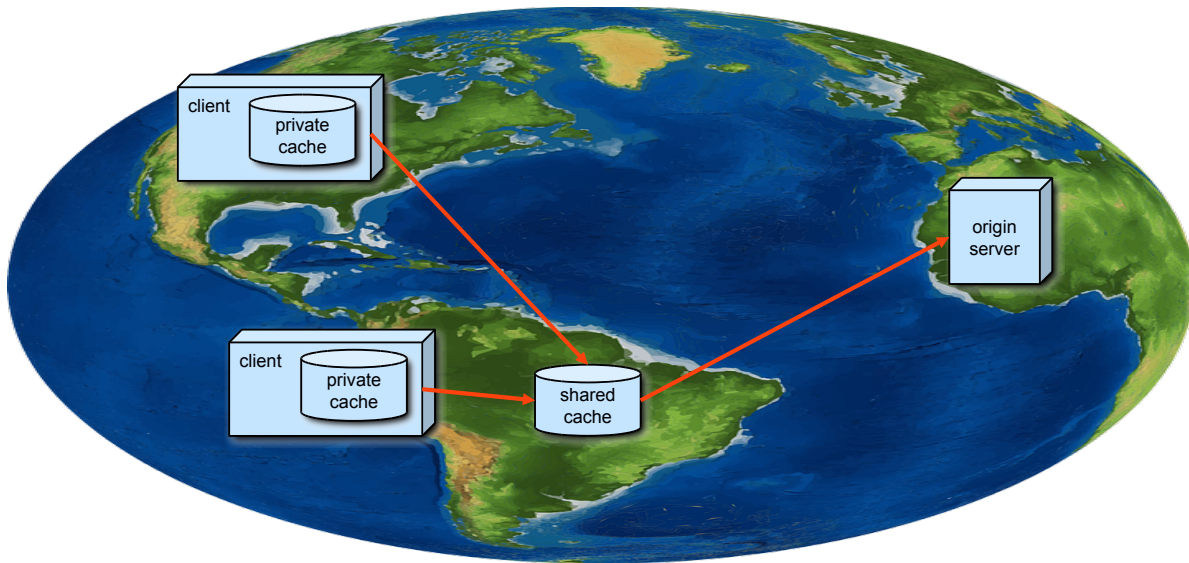


Figure 122: HTTP caches

The caching behavior of a response and thus the trade-off between network latency and transmission time on the one hand and consistency and security on the other hand can be controlled by numerous **cache directives**. These are key-value pairs in the HTTP header field `Cache-Control` (like `max-age`, `s-maxage`, `no-store`, `private`, `public`, `must-revalidate`) and a date-time for `Expires`. These directives determine (a) if a response is stored in the first place by a HTTP cache and (b) whether it can be used, i.e., is fresh enough to be a **cache hit** for a request, or cannot be used, i.e., is a **cache miss** due to the response's staleness.

However, it is notoriously hard to define a fixed time in advance for a resource to be considered fresh. **Validation requests** provide a compromise between freshness guarantees and short response times: If a response contains the directive `no-cache` (alternatively `must-revalidate` and `max-age=0`) then the response may indeed be cached but not used unconditionally in subsequent requests for the same resource. Instead, a subsequent request for the same resource causes a validation request that includes an ETag value of the cached response in the `If-None-Match` field. ETag values are checksums (often hash values) for the resource's contents. The origin server or upstream cache compares the ETag value in the request with that of the resource it holds. If both are equal, it responds positively with a small response that basically includes the status code 304 Not Modified which allows the client to use its cached response (cache hit).

The response time for validation requests mostly consists of propagation delays (cf. Section 5.2.2). Transmission delays are short due to the small messages exchanged in the validation. Therefore, validation requests are particularly helpful for big resources because that would cause long transmission delays.

⁶⁰ HTTP specifications rather speak the more generic term "user agent" (cf. Section 2.3.12).

EXAMPLE 123: CACHE VALIDATION IN HTTP

Figure 123 shows the initial request and respective response for the resource `/file`. The response headers particularly include a `max-age` of two minutes after which the response needs to be renewed. Until then, validation requests suffice.

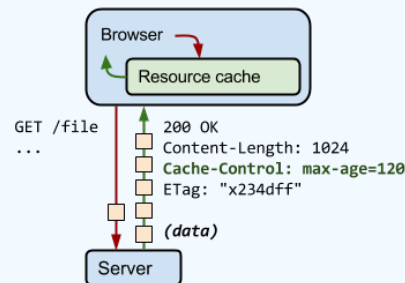


Figure 123: Initial request for resource `/file` [128]

Figure 124 exemplifies a validation request. Here, the private cache (resource cache) already contains a response with `ETag x234dff`.

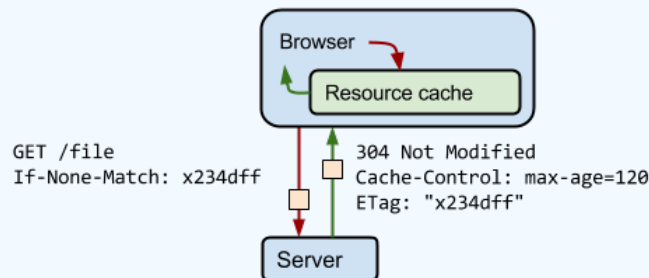


Figure 124: Validation request and response [128]

For a full explanation of all directives please refer to [114].

A typical setting for `Cache-Control` looks like the one in Figure 125. The first response often contains a server-side generated HTML resource. Since this often is highly dynamic, it should better not be reused from a cache without validation. However, the HTML contains URL references to other resources, which have to be fetched as well. The `style.3da37df.css` is a more stable resource and bears no secrets. The filename and thus the URL to the resource contains [fingerprints](#), `3da37df` in this case. This is a hash value that depends on the contents of that file. Therefore, if the CSS changes to a new revision, this hash value changes and therefore the URL for this resource. Since a resource revision (with a constant URL) never changes by design, the `max-age` could be set to a very high value, e.g., one year. The same is true for the `script.8sd34ff.js`. However, this script might contain confidential data and should not be stored in shared caches, hence the private. The `photo.jpg` does not use fingerprinting. However, one can live with a stale `photo.jpg` for one day at most, so the `max-age` is one day here.

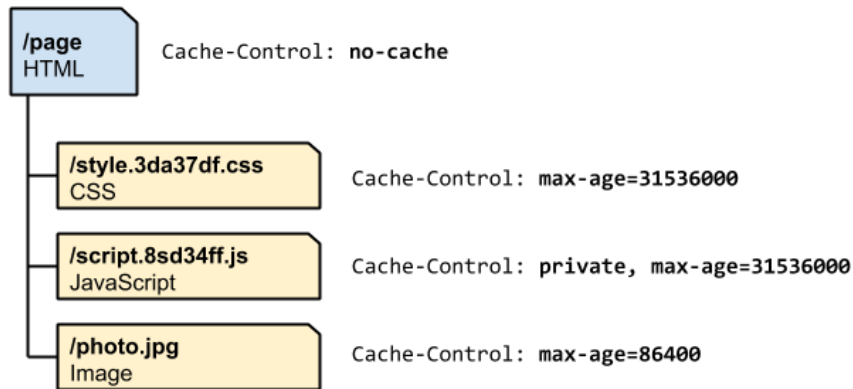


Figure 125: Cache-Control example [128]

27.1.6 HTTP 1.1 Shortcomings

In HTTP 1.1 multiplexing is delegated to the transport layer (TCP). However, TCP connections

- each allocate memory of about 150 KB, mostly for its read and write socket buffers⁶¹,
- open a file descriptor on Linux systems, which are limited per process typically to 1024 (check that with `ulimit -n`),
- require numerous request and responses to establish particularly with TLS encryption between TCP and HTTP, and
- start and accelerate slow (additive) but break hard (multiplicative) in case of package losses due to TCP congestion control (cf. Section 5.2.2).

Browser thus establish no more than six TCP connections per origin (domain) which then often turn out to be bottlenecks: Although with [HTTP 1.1 Pipelining](#) [129, p. 54] several SAFE requests $a, b, c, \dots$ can be sent via the same TCP connection before any response arrives, the order of responses must match those of the requests, i.e., the response for b cannot be sent before the response for a . If a and b are processed in parallel on the server but a takes much longer, the response for b has to wait. This effect is also called [head-of-line blocking](#) [20, p. 194].

27.1.7 HTTP/2 Multiplexing

HTTP/2 does multiplexing on its own. HTTP/2 messages of arbitrary size are broken down to a number of HTTP/2 frames, at least one for the HTTP header and arbitrarily more for binary encoded chunks of the message payload. These frames are sent over HTTP/2 streams, which resemble TCP connections but are not that limited by their number and can be prioritized.

⁶¹ For TCP, read socket buffers constitute the final packet destination. After being placed in a read socket buffer, a TCP packet is acknowledged (ACKed). On Linux, you can check the minimal, default, and maximal buffer sizes for TCP socket buffers with `sysctl net.ipv4.tcp_rmem` and `sysctl net.ipv4.tcp_wmem`.

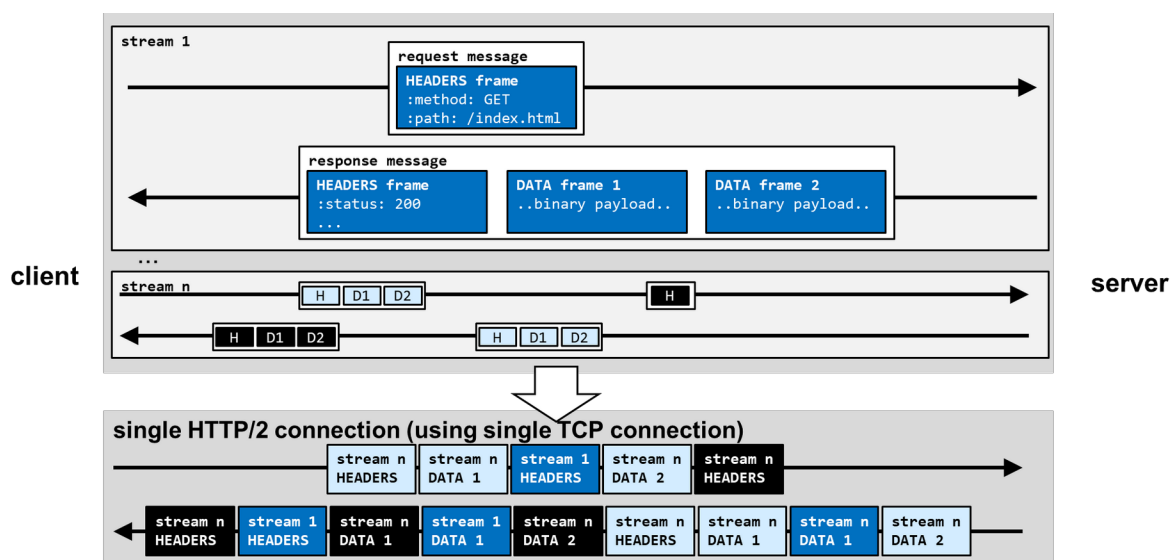


Figure 126: HTTP/2 multiplexing

As one can see by Figure 126, frames within a stream remain their ordering when put on a HTTP/2 connection but frames might be arbitrarily interleaved with frames from other streams. One HTTP/2 connection uses exactly one TCP connection. In contrast, the HTTP/2 frames are further broken down and wrapped into fixed-sizes length TCP packets.

Establishing just one TCP connection per server of course has one mayor drawback: The multiplicative throughput decrease due to an IP package loss affects all streams.

27.1.8 HTTP/3: QUIC over UDP

Since HTTP/2 multiplexes multiple streams over a single TCP connection, all streams are affected by a packet that gets lost or needs to be reordered. More precisely, all frames in all streams that come after such an erroneous frame are affected. This effect is known as head-of-line blocking. TCP itself cannot help here since it is agnostic of the different HTTP/2 streams.

Moreover, with TLS over HTTP/2 the TCP connection and the TLS connection are basically established in sequence.

HTTP/3 is therefore no longer based on TCP but UDP. Since UDP alone lacks important features like congestion control, retransmission and reordering of packets, QUIC [83] supplements UDP on the transport network layer.

Network Layer	HTTP 1.1	HTTP/2	HTTP/3
Application	HTTP 1.1	HTTP/2	HTTP3
Security	TLS 1.3	TLS 1.3	QUIC
Transport	TCP	TCP	UDP
Network	IP	IP	IP

Table 44: HTTP Versions and Network Stacks

QUIC's takes different HTTP streams into account. In case of a packet loss, only the respective stream is affected by the head-of-line blocking. Moreover, TLS protocol handshake data can be put on the very first messages, i.e., the transport and security layer handshakes happen in parallel. Actual HTTP/3 request message data can already be put on the third exchange.

27.1.9 HTTP Omissions

The HTTP standard leaves aspects unspecified like

- the structure of not just syntactically well-formed but also valid URLs, i.e., those that do not provoke a response with a 404 (Not Found),
- the structure of valid requests payloads, e.g., valid key names and,
- the structure of responses payloads,
- detailed meanings of HTTP status codes in responses, e.g., that a 403 (Forbidden) may result from a missing authorization,
- valid media types for requests and responses like `application/json`,
- accepted authentication methods like OpenID Connect.

These are refined by standards that are based on HTTP and are described in the following sections and chapters.

27.2 HTTP Streaming

HTTP is mostly used for request-response oriented communication. However, it supports a weak form of streaming that is prone to disconnections if messages are not produced rapidly on the server-side.

EXAMPLE 124: HTTP STREAMING FOR LLM RESPONSES

HTTP streaming is used, e.g., in prompts for Large Language Models like GPT [130], where the whole response could take minutes to be computed but single tokens can be sent rapidly as chunks right after they have been generated to the client for the user to read.

In HTTP/1, streaming is implemented via chunked transfer encoding (CTE) [129, p. 36]. Instead of a-priorily declaring the size of a response via HTTP response header attribute `Content-Length`, the server indicates via header attribute `Transfer-Encoding: chunked` that the response is being delivered as an—potentially indefinite—stream of so-called chunks that arrive one by one. Each chunk may contain arbitrary texts preceded by their length in hexadecimal numbers. The (potentially never occurring) end of a stream is signaled by a `0\r\n\r\n` zero chunk.

In HTTP/2 and HTTP/3, no extra HTTP header attribute is required nor extra text in HTTP bodies that precedes chunks in order to encode the chunking. Instead, every response comes as a stream of frames, even if the size of the response is very well known right away. The end of stream is signaled by a frame with header flag `END_STREAM` which might be even in the very first frame or never come.

27.3 Server-sent Events (SSE)

HTTP streaming is only a weak form of streaming because response time timeout settings that are set at the client side and in intermediate proxies still apply. Common browsers have a preset timeout period of 5 minutes. If the time that passes between two chunks is longer than this timeout period, the underlying TCP connection is closed, and later chunks cannot be delivered.

Thus, with Chunked Transfer Encoding alone, HTTP provides no real streaming and is useful only for use cases like the one in Example 124:.

Server-sent Events (SSE) [131] are based upon HTTP streaming. With SSE, connections are automatically reestablished. Moreover, instead of just plain text, with SSE each chunk text contains a list of key-value pairs with keys like data, event, or id. Such an SSE chunk text poses an “event” pushed by the server, hence the name SSE.

27.3.1 SSE Protocol

There is no special SSE protocol. SSE just refines what is covered in the HTTP 1.1 standard, particularly the chunked transfer encoding

EXAMPLE 125: SERVER-SENT EVENT MESSAGES

In order to initiate a stream from a server, a client (browser) has to send a request like this [20, p. 282]

```
GET /path/to/stream-url HTTP/1.1
Host: example.com
Accept: text/event-stream
```

A server would then respond with something like

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Type: text/event-stream
Transfer-Encoding: chunked

retry: 15000

data: First message is a simple string.

data: {"message": "JSON payload"}

event: foo
data: Message of type "foo"

id: 42
event: bar
data: Multi-line message of
data: type "bar" and id "42"

id: 43
data: Last message, id "43"
```

Here, we have five chunks each of which is delimited by a blank line. With SSE, chunks are not arbitrary textual (UTF-8 encoded). Instead, each line is a key value pair where keys are defined by SSE.

27.3.2 SSE API

The API for SSE is defined in [131].

EXAMPLE 126: SERVER-SET EVENT CLIENT

A client-side JavaScript that uses the SSE API could look like this [15, p. 280]

```
var source = new EventSource("/path/to/stream-url");
source.onopen = function () { ... };
source.onerror = function () { ... };
source.addEventListener("foo", function (event) { processFoo(event.data); });
source.onmessage = function (event) {
  log_message(event.id, event.data);
  if (event.id == "CLOSE") {
    source.close();
  }
}
```

Note, that all the chunks in the response from above would be passed to `onmessage` but only those with key-value event: `foo` would be passed to function in the `addEventListener`. (“foo”, ...

27.4 gRPC

[gRPC Remote Procedure Calls \(gRPC⁶²\)](#) use a binary format called [Protocol Buffers](#) for messages and an overall approach which resembles the Common [Object Request Broker Architecture \(CORBA\)](#) [132], a protocol for distributed systems that was widespread in the 1990s. In contrast to CORBA, gRPC is built on top of HTTP/2 instead of the Internet Inter-ORB Protocol (IIOP).

While REST promotes ease of use through self-descriptiveness of messages, gRPC favors small footprint and thus performance.

27.4.1 Protocol Buffer Messages

Protocol Buffers are “a language-neutral, platform-neutral, extensible way of serializing structured data for use in communications protocols, data storage, and more” [133]. Messages contain mere values in a binary encoding with minimal meta data, particularly field type, field length and field number (position). Such messages can be (de-)serialized only with [message types](#), which are defined in `.proto` files. Such message types are used to automatically generate proxies and stubs for the respective programming languages.

⁶² This is a recursive acronym.

EXAMPLE 127: PERSON ATTRIBUTES IN PROTOCOL BUFFER MESSAGES

Attributes of a person could be defined in a message type in a `person.proto` file like this:

```
message Person {  
  string name = 1;  
  int32 id = 2;  
  string email = 3;  
}
```

Using the tool `protoc`, one can generate a class `Person.java` for a sender, which can be used like this:

```
Person person =  
  Person.newBuilder()  
    .setName("Max Mann")  
    .setId(123)  
    .setEmail("max@gmail.com")  
    .build();
```

The values `Max Mann`, `123`, and `max@gmail.com` are then encoded to binary, prefixed by type and length and concatenated to

`0A 08 4D 61 78 20 4D 61 6E 6E 10 7B 1A 0D 6D 61 78 40 67 6D 61 69 6C 2E 63 6F 6D`

according to [121] and tested with <http://yura415.github.io/js-protobuf-encode-decode/>. The underlined `1A16` is `000110102` in binary, where the left `000112 = 310` encodes the field number and the right `0102 = 210` indicates the type string. The underlined `0D16` is `1310` is the length of that string in bytes.

27.4.2 Protocol Buffer Services

`.proto` files can also contain service definitions, which are like interface definitions, and which refer to message types.

EXAMPLE 128: PROTOCOL BUFFER SERVICES

The following content of a `.proto` file defines a service `Greeter`.

```
service Greeter {  
  rpc SayHello (HelloRequest) returns (HelloReply);  
  rpc BiDiSayHello (stream HelloRequest) returns (stream HelloReply);  
}  
  
message HelloRequest {  
  string name = 1;  
}  
  
message HelloReply {  
  string message = 1;  
}
```

`SayHello` defines a simple request-response communication pattern, i.e., one request is followed by one response. Other communication patterns can be defined as well, like full-duplex messaging comparable to WebSockets. `BiDiSayHello` exemplifies this.

27.5 WebSockets

27.5.1 WebSocket Protocol

The [WebSocket Protocol](#) [78] supports full-duplex communication protocol (cf. Figure 72). In particular, a server can proactively push messages to a client provided that a client has established a WebSocket connection and thereby a callback beforehand.

In essence, the WebSocket Protocol let clients communicate with servers via TCP, i.e., on network layer 4 instead of 7, yet with some abstractions that simplify the communication:

- **HTTP Handshake for Initial Connection & Firewall Compatibility:** Instead of requiring a completely new port or protocol for the initial setup, WebSockets start with a standard HTTP/1.1 GET request with special Upgrade and Connection headers (cf. [85, Sec. 7.8]). Therefore, they can reuse 443 as the target port for messages from client to server, which is likely to be not blocked by a firewall. WebSockets can also reused an already established TLS connection over that port and encrypt the payload of the WebSocket messages.
- **Subprotocol Negotiation:** With the Sec-WebSocket-Protocol header in the initial HTTP request, clients can directly negotiate a more specific protocol that relies on the WebSocket protocol like the Web Application Messaging Protocol (WAMP). A registry of such protocols can be found in [134].
- **Message Framing:** As opposed to raw TCP, it supports messages of arbitrary size instead of fixed sized frames. Splitting and reassembling messages from TCP frames is transparent to the clients and servers.
- **Control Frames:** Ping-Pong messages and Close messages allow for keeping connections alive and for controlled shutdown, respectively.

27.5.2 WebSocket API

Contemporary browsers implement the [WebSocket API specification](#) [135] that allow for JavaScripts running in clients (browsers) to communicate with servers via the WebSocket Protocol, which also implement a WebSocket API like with Spring WebSocket Support [136]. In particular, such browser-side JavaScripts may define a JavaScript function `onMessage(message)` that is invoked if a message is pushed from the server, i.e., `onMessage(message)` then works as remote callback function.

EXAMPLE 129: WEB SOCKET CLIENT IN JAVASCRIPT

The following HTML with embedded JavaScript is taken from <https://www.websocket.org/echo.html> and demonstrates the usage of the WebSocket API. Please note that normally the `onMessage(message)` method does not close the WebSocket but is called numerous times due to messages sent by the server.

```
<!DOCTYPE html><meta charset="utf-8" />
<title>WebSocket Test</title>
<script language="javascript" type="text/javascript">
    var wsUri = "wss://echo.websocket.org/";
    var output;

    function init() {
        output = document.getElementById("output");
        testWebSocket();
    }

    function testWebSocket() {
        websocket = new WebSocket(wsUri);
        websocket.onopen = function(message) { onOpen(message) };
        websocket.onclose = function(message) { onClose(message) };
        websocket.onmessage = function(message) { onMessage(message) };
        websocket.onerror = function(message) { onError(message) };
    }

    function onOpen(message) {
        writeToScreen("CONNECTED");
        doSend("WebSocket rocks");
    }

    function onClose(message) {
        writeToScreen("DISCONNECTED");
    }

    function onMessage(message) {
        writeToScreen('<span style="color: blue;">RESPONSE: ' +
message.data+'</span>');
        websocket.close();
    }

    function onError(message) {
        writeToScreen('<span style="color: red;">ERROR:</span> ' + message.data);
    }
}
```

```

function doSend(message) {
    writeToScreen("SENT: " + message);
    websocket.send(message);
}

function writeToScreen(message) {
    var p = document.createElement("p");
    p.style.wordWrap = "break-word";
    p.innerHTML = message;
    output.appendChild(p);
}

window.addEventListener("load", init, false);
</script>
<h2>WebSocket Test</h2>
<div id="output"></div>

```

27.6 Comparison of Protocols

It is not quite easy to delineate the HTTP-based web application layer protocols. Table 45 sums up all properties and compares them.

Feature	WebSockets	Server-Sent Events (SSE)	gRPC	HTTP/1.1	HTTP/2	HTTP/3
Communication Type	Full duplex, bi-directional	Unidirectional (server to client)	Full-duplex, bi-directional (with streams)	Request/response	Request/response	Request/response
Use Case	Real-time updates, chat applications, gaming	Real-time updates like news feeds, notifications	Microservices communication, RPC	General web browsing and APIs	Improved performance for web browsing and APIs	Further improved performance and faultlessness
Protocol State	Stateful	Stateless	Stateful (with streams)	Stateless	Stateless	Stateless
Underlying Application Layer Protocol	WebSocket protocol	HTTP/1.1	HTTP/2	HTTP/1.1	HTTP/2	HTTP/3
Underlying Transport Layer Protocol	TCP	TCP	TCP	TCP	TCP	QUIC + UDP
Transport Layer Connection Persistence	Persistent	Persistent (long-lived HTTP connection)	Persistent (cf. HTTP/2)	Per request (new TCP connection per request)	Persistent (multiplexed streams on persistent TCP)	Persistent (multiplexed streams over QUIC)
Message Format	Text (UTF-8) or binary	Text (UTF-8)	Binary (Protocol Buffers)	Text (headers and body)	Binary headers, text or	Binary headers, text or

Feature	WebSockets	Server-Sent Events (SSE)	gRPC	HTTP/1.1	HTTP/2	HTTP/3
					binary body	binary body
Reconnection	Requires client-side logic	Built-in automatic reconnection	Built-in with retry policies	New connection per request anyway	New connection on first request + connection reuse	New connection on first request + connection reuse
Network Latency Increase	Low	Low to moderate	Very low	Higher due to connection setup for each request	Lower latency with multiplexed connections	Lower latency with QUIC
Scalability	Good, but needs sticky sessions or state management	Good, simpler without sticky sessions	Excellent, designed for high performance and scalability	Moderate due to connection overhead	Excellent due to multiplexing and header compression	Excellent HTTP/2 + improved transport
Interaction with Load Balancers	Requires support for WebSocket protocol, typically uses sticky sessions	Long-lived connections need proper timeouts	Requires support for HTTP/2, benefits from load balancing features	Requires load balancing for each connection	Efficient load balancing with multiplexing	Efficient load balancing with QUIC and multiplexing
Browser Support	98.14% on caniuse (May 2024)	97.54% on caniuse (May 2024)	Limited, primarily for backend services	100% presumably	97.49% on caniuse (May 2024)	78.37% on caniuse (May 2024)
Security	Secure with WSS (WebSockets over TLS)	Secure with HTTPS	Secure with TLS	Secure with HTTPS	Secure with HTTPS	Secure with HTTPS over QUIC
Specific Security Risks	Cross-Site-Web-Socket-Hijacking (CSWH) [137]	Like HTTP	Like HTTP	Like HTTP	Like HTTP	Like HTTP
Implementation Complexity	Moderate	Low	High	Low	Moderate	Moderate to high (due to QUIC)
Library and Framework Support	Wide support across various languages and frameworks	Native browser support, easy to implement	Strong support with official libraries for multiple languages	Wide support	Wide support	Emerging support
Message Ordering	Preserved	Preserved	Preserved	Preserved	Preserved	Preserved
Con-currency	Good for many	Handles many	Excellent with	Limited,	Excellent,	Excellent,

Feature	WebSockets	Server-Sent Events (SSE)	gRPC	HTTP/1.1	HTTP/2	HTTP/3
Handling	concurrent connections	clients well	built-in support for concurrency	each request opens a new connection	supports multiplexing	supports multiplexing over QUIC
Resource Efficiency	Efficient for maintaining active connections	Can be resource-intensive on the server side	Highly efficient due to Protocol Buffers and HTTP/2	Less efficient due to connection overhead	Highly efficient due to multiplexing and compression	Highly efficient with multiplexing and QUIC
Client Libraries	Widely available	No special libraries needed (native support in browsers)	Official libraries for many languages	Widely available	Widely available	Emerging availability
Mobile Support	Well-supported on most platforms	Well-supported on most platforms	Supported, but primarily used in backend services	Well-supported	Well-supported	Emerging (supported in modern mobile browsers)
Firewall/ NAT Traversal	Good, but may require special handling for certain environments	Good, as it uses standard HTTP/HTTPS	Excellent, uses HTTP/2 which works well through firewalls	Good, but each new request might need traversal	Excellent, works well through firewalls	Excellent, designed to work well through firewalls
Data Compression	Supported	Supported via HTTP compression	Supported with Protocol Buffers and HTTP/2 compression	Supported via HTTP compression	Supported with header compression	Supported with header compression and QUIC
Error Handling	Requires custom implementation	Standard HTTP error codes	Advanced, with gRPC-specific error codes and retries	Standard HTTP error codes	Standard HTTP error codes	Standard HTTP error codes

Table 45: Comparison of WebSockets and Server-sent Events

28 Representational State Transfer (REST)

Representational State Transfer (REST) as defined by [138, p. 76 ff.] is an architectural style for web applications and their web APIs that builds upon HTTP. This architectural style is defined by the following principles.

28.1 Client-Server Architecture

REST is just applicable for client-server architectures with a request-response communication style. Peer-to-peer architectures or unidirectional messaging cannot be subject of REST. Please note that although we tend to think of generic clients like browsers in the context of the web. However, REST also applies to other kinds of clients (cf. Section 23.2).

28.2 Statelessness

REST relies on HTTP, which is stateless (cf. Section 21.2). Therefore, server-side connection states are not an issue. For application states, REST generally advocates client-side sessions [138, p. 79]. Navigational states can easily be expressed in resource representations by means of [hypermedia](#): A resource representation may contain [links](#) that trigger actions which are valid in the current navigational state.

EXAMPLE 130: NAVIGATIONAL STATE

For example, a page in a paginated list is such a resource which typically contains links for forward and backward navigation.

Authentication states can be kept in [cookies](#). [JSON Web Token \(JWT\)](#) is a commonly used standard format particularly for authentication information that can be transferred via a cookie (cf. Section 30.4.6). Please note that a JWT cookie is not just a reference to some server-side data record. Instead, it is a self-contained data record (i.e., a value, not a reference) that is supposed to be understood by every application server of the same kind in an out-scaled tier.

In theory, even [preliminary transaction data](#) could be kept in client-side sessions and transmitted back and forth in each request and response. Since these data can become large, session centralization is the usual strategy here, although not really mandated by REST.

EXAMPLE 131: PRELIMINARY TRANSACTION DATA

Contents of a web shop's shopping cart or user's input like receiver addresses or payment information during checkout are an example for preliminary⁶³ transactional data. These data become final transactional data after a legally binding purchase. Although it would not be catastrophic but nonetheless annoying if these data get lost along with a session.

In [53, pp. 81–86, 455–464] Fowler provides a similar discussion about session states from a pattern-oriented viewpoint.

28.3 Uniform Interface

Uniform interfaces particularly support loose coupling (cf. Section 14.5) between clients and servers. They are delineated by the following REST principles.

⁶³ The term "preliminary" is from a business perspective, e.g., before a legal transaction took place. From a technical database point of view, it does not matter if a transaction contains preliminary or final data.

28.3.1 Resource identification in requests

Request messages refer to resources by means of URIs.

28.3.2 Resource manipulation

Representations of resources are transferred between client and server back and forth. A client has enough information about a resource to manipulate it properly and replaced the updated resource via request method PUT or delete it.

28.3.3 Self-descriptive messages

Messages, files, or data serializations in general are the more **self-descriptive** the less additional schema information is need to in order to process them. Such schemas include encoding information, delimiters for single values, mnemonic key names and even an internal (hierarchical) structure between single values.

EXAMPLE 132: SELF-DESCRIBING DATA

A data record with two values "René" and "1978" can be serialized in Java like this

```
dataOutputStream.writeUTF("René");  
dataOutputStream.writeInt(1978);
```

This produces a byte sequence which look like this in hex code:

00:05:52:65:6E:C3:A9:00:00:07:BA.

A (developer of a) receiver of this message has no chance to deserialize such a byte sequence without additional schema information. Such a byte sequence is the opposite of being self-descriptive.

Another common way to serialize such data is to put it into an XML or JSON document like this:

```
{"name" : "René", "yearOfBirth" : 1978}.
```

Here, encodings, delimiters and even meanings of the values are much clearer. A developer would know how to deserialize such a message and how to process the values in a reasonable way. Moreover, such self-descriptive messages are easy to compose by hand in order to, e.g., test or explore an API. Simple tools like the ubiquitous Linux command line tool `curl` oftentimes suffice to do so.

Self-descriptive messages come with the price of a higher data volume because the meta data, i.e., keys, delimiters and structures expressed by brackets and colons consume space on their own.

In the previous example, the byte sequence consists of 11 bytes of pure value (payload). Conversely, in the JSON just the underscored characters are actual values (payload). Therefore, this JSON consumes 40 bytes if encoded in UTF-8.

28.3.4 Hypermedia as the Engine of Application State (HATEOAS)

Use cases of typical web applications are often implemented in processes that cause a sequence of requests and responses between clients and servers. Valid follow-up requests depend on the state the process currently is in.

EXAMPLE 133: HATEOAS

In a web shop, at least one item has to be put in a shopping cart before the shopping cart can be checked out.

Hypermedia as the Engine of Application State (HATEOAS) [138, p. 86] goes one step further than just self-descriptiveness. A response not just includes data and schema data pertaining the resource itself but also **hypermedia**, i.e.,

1. links with URLs or URL templates to associated resources or embedded associated resources,
2. applicable HTTP methods (GET, PUT, POST, DELETE etc.) and schemas for representations of these associated resources that can be put in request messages.

HATEOAS also promotes the same properties for web APIs (with specific clients) like what the World Wide Web promotes for web UIs (where generic clients like browsers are the clients), i.e., documents that are linked with each other:

- A browser user needs just an initial resource, i.e., a “home page” and a basic understanding of how hypermedia works. He or she discovers web UIs via, e.g., links and forms and knows what to do with them, e.g., by filling out the required form text fields.
- Likewise, a specific client developer discovers a web API starting from an initial resource. He or she discovers concrete associated resources by links with URLs. Moreover, he or she discovers what can be done by additional schematic information.

The client can be kept mostly agnostic from the specifics of the web API that the server exposes. Ideally, even if the web API evolves and contains additional data and hypermedia, a client can be left untouched. For example, if the client renders a UI, e.g., is a native smartphone application, it could dynamically populate its UI with input fields and action buttons (like next, search, submit, cancel, delete etc.) depending on the responses.

Of course, there is a trade-off even in this “ideal” world: A client that dynamically renders a UI based on web API responses is somewhat close to a browser. However, web API response typically lacks layout information in contrast to web UI responses containing HTML, CSS etc. This is a key difference between UIs and APIs. Therefore, such clients would blissfully render whatever hypermedia is found in responses in a quite generic way (e.g., all links as buttons in a row as they appear in the response) which could harm user experience.

HATEOAS as defined by [138, p. 82] is quite vague and particularly leaves open, how hypermedia syntactically looks like for different kinds of interfaces like web UIs or web APIs and their associated media types like text/HTML or application/json.

EXAMPLE 134: HYPERMEDIA IN HATEOAS

For text/HTML, hypermedia looks like these

```
<img href="http://www.example.com/image.png">alttext</img>  
<a href="/anotherpage.html">Another Page</a>  
<form method="post" action="/someAction">...</form>
```

For web APIs that emit mime type application/json, there are also standards for hypermedia:

1. **Hypertext Application Language (HAL)** [139] basically defines a JSON-based media type application/hal+json for links to associated resources embedded in JSON representations and
2. **HAL-FORMS** [140], which particularly defines also defines a JSON-based media type application/prs.hal-forms+json for defining applicable HTTP methods of associated resources and schemas for sending PUT and POST requests to them.

EXAMPLE 135: HYPERTEXT APPLICATION LANGUAGE

Figure 127 provides an example for HAL. The example is a logistics application called Parcer (for Parcel Services) that allows for managing customers, their shipments and associated PDF labels. Responses of the Parcer WebAPI are of media type `application/hal+json` and include some hypermedia, i.e., links to associated resources.

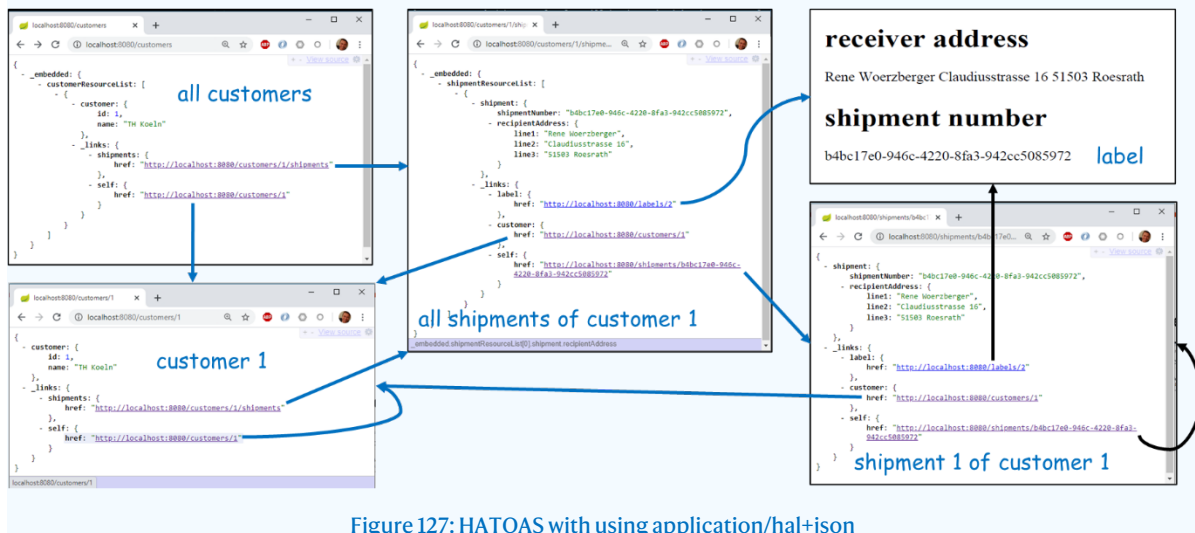


Figure 127: HATOAS with using `application/hal+json`

28.4 Cacheability

Caching is a means to reduce response times (cf. Section 27.1.5). Cache directives are optional HTTP headers. REST promotes extensive use of them.

28.5 Layered System

In a layered system, a client should not care whether it communicates with an origin server or some intermediate like a share cache or load balancer in order to keep clients and servers loosely coupled.

28.6 Richardson Maturity Model

As REST is just a set of principles, which can either be followed to or not, there is no binary choice of either having a RESTful API or not. The so-called Richardson Maturity Model [141] presents four levels of maturity w. r. t. REST principles. Martin Fowler provides a good discussion of this model in [142] which is summarized in Table 46.

Level	Description
0 - Plain old XML	Devoid of REST principles. HTTP is used as a mere tunnel. There is no notion of resources, just a single endpoint URL that accepts only POST requests. The Simple Object Access Protocol (SOAP) or XML Remote Procedure Calls (XML-RPC) work that way.
1 - Resources	Adds the notion of resources with different URLs
2 - HTTP request methods	Uses appropriate requests methods for each request. In particular, it uses GET requests if applicable in order to profit from HTTP caching.
3 - HATEOAS	The topmost level requires that APIs are self-descriptive to the extent of HATEOAS.

Table 46: Levels of the Richardson Maturity Model

29 API Access Standards

The HTTP protocol and its extensions do not prescribe, how the payloads of HTTP requests and responses, the “representations”, look like and what they mean. In other words: They are agnostic about the syntax and semantics of HTTP requests and responses. The following standards fill this gap.

29.1 OpenAPI

The [OpenAPI Specification](#) [143] specifies a language for machine-readable interface files for RESTful APIs. These interface files are written in either YAML or JSON. They define the resource URLs, request, and response messages along with their respective headers and bodies.

Interface files are machine-readable; there is tooling available for creation of API mocks like Swagger UI as well as stubs and proxies for numerous languages.

EXAMPLE 136: OPENAPI SPECIFICATION FOR A PET STORE

The following specification (abridged version of [144]) in YAML-OpenAPI basically states that one could send HTTP GET requests to a URL like

```
http://petstore.swagger.io/v1/pets/23
```

where 23 is the identifier of a certain pet. If that resource exists, the response has a status code of 200 (OK) and a representation of media type `application/json` of that pet is put in the response body. The JSON in the response body has to key-value pairs, `id`, `name`, and `tag` like

```
{
  "id" : 23,
  "name" : "Beethoven",
  "tag" : "dog"
}
```

with `id` and `name` being mandatory, `id` being an integer and `name` and `tag` being strings.

```
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Swagger Petstore
  license:
    name: MIT
servers:
  - url: http://petstore.swagger.io/v1
paths:
  /pets/{petId}:
    get:
      summary: Info for a specific pet
      operationId: showPetById
      tags:
        - pets
```

```
parameters:
  - name: petId
    in: path
    required: true
    description: The id of the pet to retrieve
    schema:
      type: string
responses:
  '200':
    description: Expected response to a valid request
    content:
      application/json:
        schema:
          $ref: "#/components/schemas/Pet"
  default:
    description: unexpected error
    content:
      application/json:
        schema:
          $ref: "#/components/schemas/Error"
components:
  schemas:
    Pet:
      type: object
      required:
        - id
        - name
      properties:
        id:
          type: integer
          format: int64
        name:
          type: string
        tag:
          type: string
    Error:
      type: object
      required:
        - code
        - message
      properties:
        code:
          type: integer
          format: int32
        message:
          type: string
```


29.2 GraphQL

OpenAPI specification predefine the schemas of resources and therefore HTTP responses. This is typical for most RESTful API regardless of whether they are documented in an OpenAPI specification. Although HTTP makes no restrictions about how sophisticated a query can be which defines what to put in the response, typical RESTful API use the URL path segment to select hierarchically organized resources. The URL query segment is not used at all or just for pagination or filtering⁶⁴.

However, API designers often do not exactly know about the needs of the API's clients, which is the normal case in the World Wide Web. So, clients have an **over-fetching** issue, i.e., just use some data from the response and thus suffer from unnecessary additional server-side processing times and transmission delays for the unused data. Conversely, clients also have to deal with **under-fetching** data, i.e., have to do multiple requests and wait multiple response times in order to gather the required data. This is known as the **n+1 problem**, where a client initially requests one resource that contains n links to other resources requiring additional requests.

GraphQL [145] is a data query and manipulation language for HTTP-based APIs. It particularly tackles the problem from above. Bodies of request messages contain the exact needs of a client. They contain keys to those values a client is interested in.

EXAMPLE 137: STAR WARS DATABASE WITH A GRAPHQL API

This example is taken from the GraphQL playground at <https://graphqlbin.com/v2/6RQ6TM>. It demonstrates an API that serves facts from the Star Wars universe.

A request message body could look like this

```
query {  
  Person(name: "Darth Vader") {  
    id  
    name  
    birthyear  
    films {  
      title  
    }  
  }  
}
```

which yields the response

⁶⁴ RESTful APIs generated by PostgREST are exception here. The URL query segment may contain an expression of a complex relational query language which is converted to SQL.

```
{
  "data": {
    "Person": {
      "id": "cj0nv9pa9ewco0130v4ocnhed",
      "name": "Darth Vader",
      "birthYear": "41.9BBY",
      "films": [
        {
          "title": "A New Hope"
        },
        {
          "title": "Revenge of the Sith"
        },
        {
          "title": "Return of the Jedi"
        },
        {
          "title": "The Empire Strikes Back"
        }
      ]
    }
  }
}
```

Although REST is an architectural style and GraphQL a query language, both can be seen as competitors. GraphQL APIs are level-zero-APIs w. r. t. the Richardson Maturity Model (cf. Section 28.6) as there is typically just one URL endpoint and just POST requests. Because of that and because response messages also depend on the request message body and not just the request URL, HTTP caching is ineffective for GraphQL.

29.3 Model Context Protocol (MCP)

The standards described so far have the following in common:

- Specific clients are programmed with respect to a certain API specification, e.g. a developer implements how requests are constructed and how responses are processed.
- Generic clients like browsers leave it mostly to the user to make the right requests and to make sense the rendered response.

The advent of Generative Artificial Intelligence gave rise to a new type of clients, i.e., [AI agents](#) (a.k.a. [AI application](#)). An AI agent has access to a Large Language Model (LLM) and can therefore reason about API specifications as well as request and responses by itself. In particular, an AI agent can decide on its own which API to call and how to call it.

For example, an AI agent might query each calendar of team members and subsequently propose a meeting in a free slot. Claude Code, Visual Studio Code (with certain plugins) or ChatGPT are well known AI agents.

29.3.1 The $m \cdot n$ Problem

With m AI agents and n MCP servers, a classic approach would demand for $m \cdot n$ implementations; one for each pair. This so-called $m \cdot n$ -[problem](#) is known from other contexts as well:

- m different code editors that require support like syntax highlighting, reactive compilation etc. for n different programming languages
- m different programming languages that need to be compiled to n different CPU architectures.

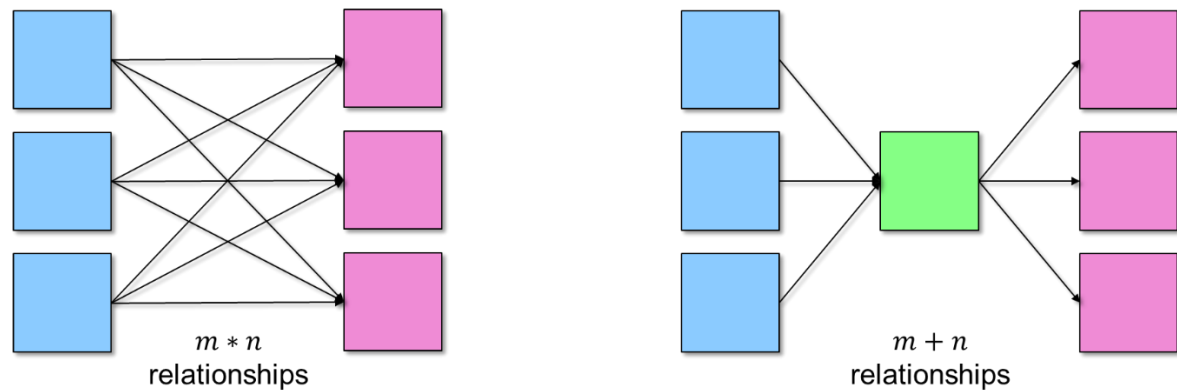


Figure 128: $m \cdot n$ problem

In either case, one can turn the $m \cdot n$ times the effort into a $m + n$ times the effort by standardizing a component in the middle that decouples both sides.

- Code editors use the [language server protocol](#) that abstract from a concrete programming language. The support of a programming language has to be implemented just once.
- Compiler frontends compile source code to [intermediate representation](#). Compiler backends transform intermediate representation to actual CPU-specific machine code.

In case of AI agents vs. servers, the [Model Context Protocol \(MCP\)](#) is that component in the middle.

29.3.2 Model Context Protocol Concepts

In the terminology of MCP, an AI agent or AI application is called [MCP host](#) that is a program that controls one or more [MCP clients](#). Each MCP client is an instance of the very same universal code implemented in the MCP host, but each MCP client instance is connected to a different [MCP server](#) in a stateful way and holds session information for that specific connection.

According to MCP specification MCP hosts with their MCP client instances on the one side and MCP servers on the other side are distinct programs running in distinct processes and communicate with each other with messages whose payloads conform to [JSON-RPC 2.0](#) [146]. Besides that, MCP does not prescribe any part transport protocol.

29.3.3 Transport Layer

If a MCP client and server run [locally](#) to each other, [standard input/output \(stdio\)](#) is common for transport. Local MCP server usually wrap local resources like the file system or a local database or may act as a proxy to external web APIs.

Conversely, the MCP server may be [remote](#) and wrap a services that resides on the same remote machine. Then the JSON-RPC messages can be transported via [WebSockets](#) (cf. Section 27.5) or [Server-sent Events](#) (cf. Section 27.3).

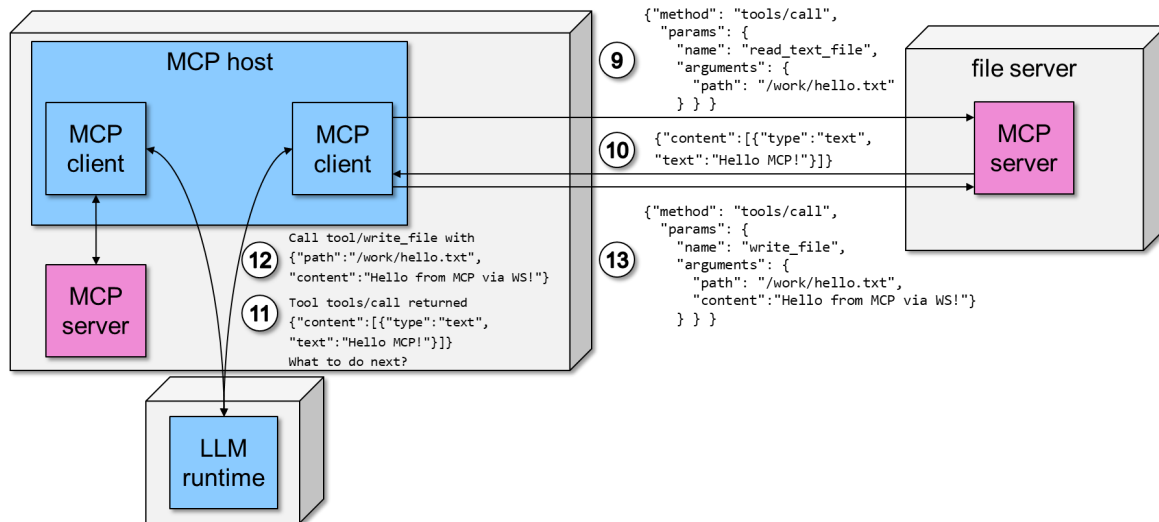


Figure 129: Model Context Protocol

Figure 129 exemplifies how a simple file server could be connected as an MCP server to an MCP host. The steps are roughly as follows (where some steps have been omitted in Figure 129 for brevity). Steps 1 to 6 are repeated for all MCP servers that are configured in the `mcp.json` that is a local file in the host which contains a `url` plus HTTP headers (for authentication etc.) for each remote MCP servers or command lines with their respective args for local MCP servers.

a) Initialization

1. **User → invoke → MCP host:** The user starts the MCP host, i.e., runs a command like `codex`, `claude`, or `gemini-cli` on the command line or invokes a certain plugin in his or her development environment (like Visual Studio Code). In particular:
2. **MCP host → create → MCP client:** The MCP host creates an MCP client for each MCP server declared in the `mcp.json`. The `mcp.json` is a local file in the host which contains an `url` plus HTTP headers (for authentication etc.) for each remote MCP servers or command lines with their respective args for local MCP servers.
3. **MCP client → capabilities → MCP server:** The MCP client sends a JSON-RPC to the MCP server with `method="initialize"`. The params contain the latest supported MCP protocol version and the supported client capabilities, i.e., if and how roots, samplings, and elicitation are supported and how (cf. Section 29.3.6).
4. **MCP client ← capabilities ← MCP server:** The MCP server sends a JSON-RPC to the MCP client with `method="initialize"` thereby sending its own capabilities, i.e., if tools, resources and prompts are supported and how.
5. **MCP client → tools/list → MCP server:** The MCP client iterates over the capabilities from the server using a JSON-RPC like

```
{ "jsonrpc": "2.0", "id": 1, "method": "tools/list", "params": {} }
```
6. **MCP client ← list ← MCP server:** The MCP server response with a JSON containing all tool/resource/prompt names along with a JSON schema [147] that describes what parameters a certain tool need as input.
7. **MCP client → list prompt → LLM:** The MCP client constructs a prompt from the lists to inform the LLM which tools/resources/prompts are available at which MCP server. This prompt particularly includes the name, the description and the inputSchema of each tool/resource/prompt for the LLM to reason about.

b) Interaction

8. **User → MCP host → LLM:** The user continuously interacts with the MCP host by formulating prompts that are forwarded to the LLM. The LLM particularly may signal in its response that the respective MCP client should make a certain call to its MCP server.
9. **MCP client → tools/call → MCP server:** The MCP client then calls the `read_text_file` tool with the path as a parameter.
10. **MCP client ← content ← MCP server:** The MCP server response with the content.
11. **MCP client → tools/call → LLM:** The MCP client puts the response into a prompt.
12. **MCP client ← LLM response ← LLM:** The LLM runtime response contains the JSON that is next to be send to the MCP server, namely calling the `write_file` tool.
13. **MCP client → tool/call → MCP server:** The MCP client sends this new JSON to the MCP server
14. **MCP client ← content ← MCP server:** The MCP server response with a JSON that reports a successful write

The key to turn the $m \cdot n$ problem into a $m + n$ problem is that the code of an MCP client is completely agnostic of the MCP server and the meaning of exchanged JSONs. The main responsibilities of a MCP client are

- to paste the MCP server response JSONs into prompts
- parse responses from LLM responses, thereby
 - recognize and extract a JSON-RPC for a follow-up MCP server request,
 - validate the embedded params JSON if it conforms to the `inputSchema` of the respective tool,
- cache the conversation history in order to decide what to feed into the next prompt in addition to the last MCP server response for the otherwise stateless LLM. For example, an MCP client should cache the response for the initial `tools/list` request and put the response into prompts in order to keep the LLM informed which tools are available at all.

Besides that, it is completely up to the LLM to reason about the incoming JSONs with their `result` values and to construct the outgoing JSONs. In comparison to classical architectures like the hexagonal architecture, a LLM performs high-level orchestration and decision logic, while the MCP client provides protocol mediation and the MCP server performs execution logic.

EXAMPLE 138: MODEL CONTEXT PROTOCOL FILE SERVER

In the following, we simulate the example scenario from above where some steps are omitted.

First create a directory with a file `~/mcpdemo/hello.txt` and some content in it for a start

```
mkdir -p ~/mcpdemo
cd ~/mcpdemo
echo "old text" > hello.txt
```

The run a MCP file server container (first line is only necessary in GitBash)

```
MSYS_NO_PATHCONV=1 MSYS2_ARG_CONV_EXCL="*" \
docker run -it --rm -p 8000:8000 \
-v "$(pwd):/work" \
supercorp/supergateway:3.4.3 \
--stdio "npx -y @modelcontextprotocol/server-filesystem /work" \
--outputTransport ws \
--port 8000 \
--messagePath /message
```

Start wscat in a Docker container as a client

```
docker run -it --rm monotykamary/wscat -c ws://host.docker.internal:8000/message
```

③ Simulate the initialization by pasting the (empty) client capabilities:

```
{"jsonrpc":"2.0","id":1,"method":"initialize","params":{"protocolVersion":"2024-11-05","capabilities":{},"clientInfo":{"name":"ExampleClient","version":"1.0.0"}}}
```

④ The MCP server responds with its capabilities namely tools without any sub-capabilities:

```
{
  "result":{
    "protocolVersion":"2024-11-05",
    "capabilities":{
      "tools":{}
    },
    "serverInfo":{"name":"secure-file-system-server", "version":"0.2.0" }
  }, "jsonrpc":"2.0", "id":1
}
```

⑤ Now paste this message into the terminal

```
{"jsonrpc":"2.0","id":1,"method":"tools/list","params":{}}
```

⑥ This responds a lengthy JSON that describes all available tools

```
{
  "jsonrpc":"2.0",
  "id":1,
  "result": {
    "tools": [
      {
        "name": "read_text_file",
        "description": "Read the complete contents of a file from ...",
        "inputSchema": {
          "type": "object",
          "properties": {
            "path": { "type": "string" },
            "tail": { "type": "number", "description": "If provided, ..." },
            "head": { "type": "number", "description": "If provided, ..." }
          },
          "required": ["path"],
          "additionalProperties": false,
          "$schema": "http://json-schema.org/draft-07/schema#"
        }
      },
      ...
    ]
  },
  ...
}
```

⑨ This JSON describes how JSON-RPC may look like such that the MCP server understands them, for example (copy that in the wscat session):

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "tools/call",
  "params": {
    "name": "read_text_file",
    "arguments": {
      "path": "/work/hello.txt"
    }
  }
}
```

⑩ This responds with a websocket message

```
{"jsonrpc": "2.0", "id": 2; "result": {"content": [{"type": "text", "text": "old text"}]}}
```

⑪ The MCP client now could construct a prompt like

Tool tools/call returned

```
{"content": [{"type": "text", "text": "old text"}]}
```

What to do next?

⑫ and, after some reasoning, the LLM could respond with

Call tool/write_file with

```
{"path": "/work/hello.txt", "content": "new text"}
```

⑬ From this LLM response, an MCP client forms the following request and sends to the MCP server (i.e. you must insert it into the running wscat session)

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "tools/call",
  "params": {
    "name": "write_file",
    "arguments": {
      "path": "/work/hello.txt",
      "content": "new text"
    }
  }
}
```

⑭ returning a message

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "result": {
    "content": [
      {
        "type": "text",
        "text": "Successfully wrote to /work/hello.txt"
      }
    ]
  }
}
```

29.3.4 MCP Server Capabilities

MCP servers provide three different capabilities each with different use case: tools, resources, and prompts. Every method the MCP server implements belongs to one of these capabilities.

a) Tools

MCP tools are meant to implement functions that are executed on the server and can mutate state there, like writing to files or databases, calling external APIs etc. The tools of an MCP server can be queried like so⁶⁵

```
{ "method": "tools/list", "params": {} }
```

and the response looks like

```
{
  "result": {
    "tools": [
      {
        "name": "read_text_file",
        "description": "Read the complete contents of a file from the file ...",
        "inputSchema": {
          "type": "object",
          "properties": {
            "path": { "type": "string" },
            "tail": { "type": "number", "description": "If provided, returns only ..." },
            "head": { "type": "number", "description": "If provided, returns only ..." }
          },
          "required": ["path"],
          "additionalProperties": false,
          "$schema": "http://json-schema.org/draft-07/schema#"
        }
      }, ...
    ]
  }
}
```

As the list of available tools may change, the MCP server can support the sub-capability `listChange` meaning that it will send a

```
{ "jsonrpc": "2.0", "method": "notifications/tools/list_changed" }
```

⁶⁵ The jsonrpc version and message id are left out for brevity in the following

to the client in that case. The Example 138: provides further details on how to use MCP tools.

b) Resources

MCP resources represent read-only data that can be retrieved from various sources. All resources of an MCP server can be queried by

```
{ "method": "resources/list", "params":{} }
```

and the response looks like

```
{
  "resources": [
    {
      "id": "filesystem:/work/hello.txt",
      "name": "hello.txt",
      "mimeType": "text/plain"
    },...
  ]
}
```

A subsequent query for that resource would be

```
{ "method": "resources/get", "params": { "uri": "filesystem:/work/hello.txt" } }
```

With a response like

```
{ "content": { "text": "old text", "mimeType": "text/plain" } }
```

Resources also support sub-capability `listChange` since the list of resources themselves might change. If the contents of a resource change, a MCP client can use "method":

"resources/subscribe" together with a `uri` to the resource if the MCP server supports this sub-capability. The MCP client then gets update notifications in case of a change in a particular resource.

c) Prompts

With tools and resources alone, it is completely up to a MCP client to transform or embed the incoming JSON-RPC responses into prompts for the LLM. However, an MCP server can assist MCP clients in formulating prompts for an LLM by means of **MCP prompts**.

A MCP client can list all available prompts with

```
{ "method": "prompts/list", "params":{} }
```

and get a response like

```
{ "result": {
  "prompts": [ {
    "name": "summarize_text",
    "description": "Summarize the given text into a short or detailed ...",
    "inputSchema": {
      "type": "object",
      "properties": {
        "text": { "type": "string" },
        "style": { "type": "string", "enum": ["short", "detailed"] } },
      "required": ["text"] } } ] } }
```

like that of `tools/list`. Consequently, a prompt behaves like a tool if called by the MCP client:

```
{
  "method": "prompts/get",
  "params": {
    "name": "summarize_text",
    "arguments": {
      "text": "MCP (Model Context Protocol) is a standard way...",
      "style": "short" } } }
```

Responses to prompt/get requests always contain messages each of which contains a fully formulated prompt text in content and a role to distinguish system and user prompts.

```
{
  "result": {
    "name": "summarize_text",
    "description": "Summarize the given text into a short or detailed summary.",
    "messages": [
      {
        "role": "system",
        "content": "You are an expert writer. You summarize text for a human."
      },
      {
        "role": "user",
        "content": "Summarize the following text in a short way.\n\n--- BEGIN  
TEXT ---\nMCP (Model Context Protocol) is a standard way...\n..." } ] } }
```

It remains for the MCP client to put the message contents together with additional text from the conversion history into a single prompt for the LLM.

29.3.5 Comparison with HATEOAS

MCP and HATEOAS (cf. Section 28.3.4) have in common that they stress the self-descriptiveness of APIs. During development, clients can be kept agnostic of the actual API specification to some degree, as the server provides methods or affordances dynamically and it is up to what- or whoever controls the client to explore these.

However, MCP was designed the intention of being interpreted by an LLM. The self-descriptiveness lies in the names of parameters and methods but particularly in the descriptions. Particularly lacking these descriptions is what makes HATEOAS not suitable for being a replacement for MCP aside it tight bounds the transport protocol HTTP.

29.3.6 MCP Client Capabilities

If a MCP server method is called, the MCP server can itself call the MCP client for further information that is necessary to fulfill the methods. The MCP client capabilities are roots, sampling, and elicitation and they can be queried by the server by, e.g., roots/list like the client can query the methods from the server.

a) Roots

A local MCP server might require read and/or write request to the local file system. By default, no file system access is permitted. This can be loosened by the client by allowing read-write access to some directory trees, which typically are those of a local git working tree.

b) Sampling

In some cases, a MCP server reasonably may use an LLM on its own. For example, if the MCP server has collected relevant data for its task, a LLM is ideal to summarize that data. The obvious design would be to let the MCP server prompt a LLM by itself. However, this has two drawbacks:

- The MCP server cannot be sure if the MCP client (and its users) would approve the data to be forwarded to another LLM.
- Using LLMs incurs costs that providers of MCP server like to avoid, particularly if they offer their MCP servers for free.

With MCP sampling, the MCP server can get access to the LLM of the MCP client together with its context. If the server sends a sampling request to the client, the client can present it to the user for approval, then feed it into the LLM and let the user also approve the LLM's response before it is responded to the MCP server.

c) Elicitation

With elicitation, the MCP server can request additional information from the user to fulfill its request. This additional information can be user properties (name, birth date etc.) or credential that are required by the MCP server to make certain API calls on behalf of the user.

30 Web Security Protocols

In this chapter, we elaborate on common web security protocols that deal with certain security threats.

30.1 Web Security Threats

Table 47 provides an overview of which web security protocol of this chapter covers which threat type.

STRIDE Model threat types	Transport Layer Security (TLS)	OAuth2	OpenID Connect
Authenticity (Anti-Spoofing)	<i>Server/client authenticity:</i> - "The server side of the channel is always authenticated; the client side is optionally authenticated." [148, p. 5]	-	<i>User authenticity:</i> Users can prove their authenticity with arbitrarily many systems (so-called relying parties) using single credentials on a single identity server.
Integrity (Anti-Tampering)	"Data sent over the channel after establishment cannot be modified by attackers without detection." [148, p. 5]	-	-
Non-repudiation (Proof of data integrity and origin)	-	If a user cannot repudiate his identity as authenticated by an authorization server, he or she cannot repudiate granted consents.	Identity providers can pass a user's non-repudiation of his or her identity to relying parties.
Confidentiality (Anti-Information-Disclosure)	"Data sent over the channel after establishment is only visible to the endpoints." [148, p. 5]	-	-
Availability (Anti-Denial-of-Service)	-	-	-
Authorization (Anti-Elevation-of-Privilege)	-	End-users can authorize a client A to access an arbitrary resource (data record) stored in server B.	Clients can be authorized to access user-related information owned by the user and stored at the authentication server.

Table 47: Web Security Protocols vs. STRIDE threat model types

These security protocols basically implement

- **encryption** and **decryption**, i.e., ensuring confidentiality of data (in transit),
- **signing** and **verifying**, i.e., ensuring integrity of data (in transit),
- **identification**, i.e., deriving an identity of a subject from certain data,

- **authentication**, i.e., proving that a subject's identity is authentic, and
- **authorization**, i.e., granting certain rights to certain web resources for identified and authenticated subjects.

By **subject**, sometimes called **principal**, we mean entities that potentially require identification, authentication, and authorization like **end-users** or **nodes** (clients and servers) in a network.

Please note that TLS has no concept of end-users but just clients and servers. OAuth2 just authorizes clients to access resources that are owned by a user and located at some other server. With OpenID Connect, a user can authenticate with a certain system (relying party) and authorize access to user related data using a centralized identity server, which stores the user's credentials and user related data.

30.2 Cryptographic Basics

The web security protocols rely on certain cryptographic algorithms, a.k.a. **ciphers**, which we discuss in this section.

30.2.1 Hashing Algorithms

Hashing algorithms implement **hash functions**. A hash function takes data of arbitrary size as input and maps it to a fix-sized output. Hash functions can be used to

- compute a storage address from a fixed range of possible storage addresses for an arbitrary large value. Hash map and dictionary data structures, which are usually supported by most programming languages, or their standard libraries are examples.
- compute a fix-sized key for an arbitrarily large value, e.g., the version control system Git stores a file's current contents in a file, whose filename is—except the first two letters—a hash value of the file's content,
- compute checksums for data in order to detect transmission or storage errors, or
- compute an equality check for two data records without actually storing and comparing these records, e.g., passwords.

Hash functions therefore cannot be injective: There are arbitrarily many inputs that **collide**, i.e., map to the same output. A **cryptographic hash function** has the following properties:

- It is **surjective**, i.e., every output value is result of some input value.
- It is **uniform**, i.e., each output value has roughly the same number of input values.
- It can be **efficiently** computed.
- It is **deterministic**, i.e., applying it to some input value always yields the same output value. So, the seemingly random output of a cryptographic hash function is not really random.
- It is **one-way**, i.e.,
 - it is practically impossible to find an input value for a given output value and
 - a collision for two arbitrary given input value is unlikely by chance, and it is practically impossible to produce a second input value that produces the same output value
- It has an **avalanche effect**, i.e., small changes like a single bit-flip in the input yields a completely different output.

The most common cryptographic hash algorithm is SHA-256 [149] which produces outputs of 256 bits.

EXAMPLE 139: OPENSSL

With the open-source tool `openssl`, you can compute SHA-256 output values in particular.

```
echo "Hello World" | openssl dgst -sha256
```

outputs

```
(stdin)= d2a84f4b8b650937ec8f73cd8be2c74add5a911ba64df27458ed8229da804a26
```

A large file also produces a SHA-256 output value of the same size.

```
dd if=/dev/random of=./largef iflag=count_bytes count=10M
```

```
openssl dgst -sha256 largef
```

outputs (for example)

```
SHA256(largef)= 52b4ddd1af82057703f746a488bf17cf3d7ccf364cf00301aeee16a2b19b03f6
```

30.2.2 Secrets and Keys

Cryptographic literature often uses terms like "key" or "secret", yet both metaphorical meanings require some clarification: A key/secret is nothing but a chunk of (seemingly) random data like 110111110100101000101001000110111010101000011110101101111001111 (in binary notation) and does not represent any information on its own. As every chunk of data can be interpreted as a (huge) integer like 16089717817529513935, keys/secrets are often considered to be (huge) integers. Keys/secrets are used as input parameters for cryptographic algorithms along with those data that is supposed to be cryptographically treated, i.e., encrypted, decrypted, signed, or verified.

If you take "unlocking / locking something" as "decrypting / encrypting something", the metaphor "key" might be misleading since common keys usually can lock and unlock the same lock. In the cryptographic world this is just true for symmetric encryption algorithms.

The metaphor "secret" might suggest that the secret itself contains some valuable and confidential information. However, if a cryptographer talks about a "shared secret" she or he normally means equal copies of again just some random data where each copy resides at a bearer of that secret.

As the name suggest, a secret must be kept **private**, e.g., hidden from the public and adversaries in particular. Conversely, a key might either be private or **public**.

Different secrets / keys differ in their lifespan. For data in rest, i.e., stored data, a key must be **static (non-transient) key** since a lost key would make later decryption or verification of encrypted or signed data impossible. Conversely, for data in transit, i.e., data that is send through a network, a key preferably is an **ephemeral (transient) key** since decryption or verification takes place immediately when the data arrives at the receiver. If an adversary eavesdrops encrypted data and gets hold of the key later, he or she can just decrypt the data which was encrypted with the key, which is in case of an ephemeral key usually just the data of a certain communication session.

30.2.3 Symmetric Ciphers

Symmetric ciphers need a single key as input along the cryptographically treated data. Symmetric ciphers are normally used for symmetric bulk en-/decryption but can in theory also used for signing and verifying.

The most commonly use symmetric cipher is "**Advanced Encryption Standard**" (AES).

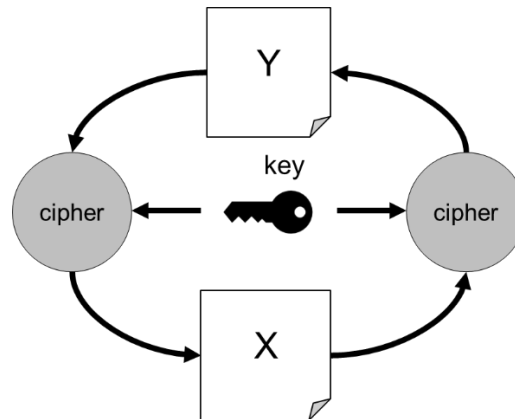


Figure 130: Symmetric ciphers

a) Encryption and Decryption

A sender feeds unencrypted data and a shared secret into a symmetric cipher in order to obtain encrypted data. On the receiver's side, the opposite happens.

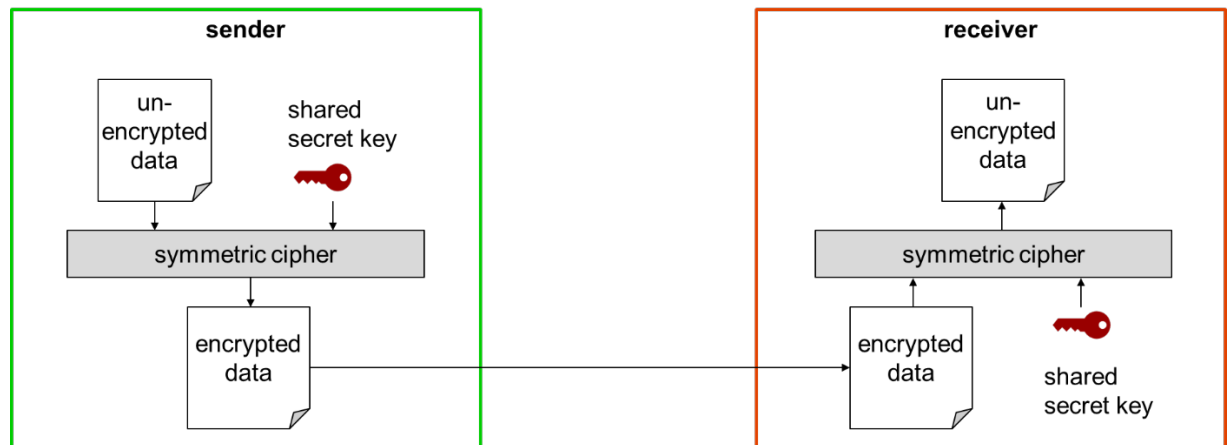


Figure 131: Symmetric encryption and decryption

EXAMPLE 140: SYMMETRIC EN- AND DECRYPTION WITH OPENSSL

You can use the cryptographic "Swiss army knife" openssl again to try out symmetric en-/decryption:

```
echo "Transfer 300$" | openssl enc -aes-256-cbc -k yourSecretKey -a -pbkdf2
```

encrypts the text `Transfer 300$` to the ciphertext

`U2FsdGVkX19dBvY0ossnK805gn4FHMk9oJaGpK4coVM=` using the symmetric cipher `aes-256-cbc` (Advanced Encryption Standard with 256 bits key length and chained block cipher) and the key `yourSecretKey` (parameter `-k`). The ciphertext is printed in Base64 encoding (due to parameter `-a`). The cipher does not use the key `yourSecretKey` as-is but derives a key of suitable length (256 bits) using the "password-based key derivation function 2" (parameter `-pbkdf2`).

Decryption (parameter `-d`) can be done like this

```
echo "U2FsdGVkX19dBvY0ossnK805gn4FHMk9oJaGpK4coVM=" | \
openssl enc -d -aes-256-cbc -k yourSecretKey -a -pbkdf2
```

b) Signing and Verifying

Although, signing and verifying is rarely used based on symmetric ciphers, it works analogously to asymmetric signing and verifying (cf. Section 30.2.4b) just with a shared secret.

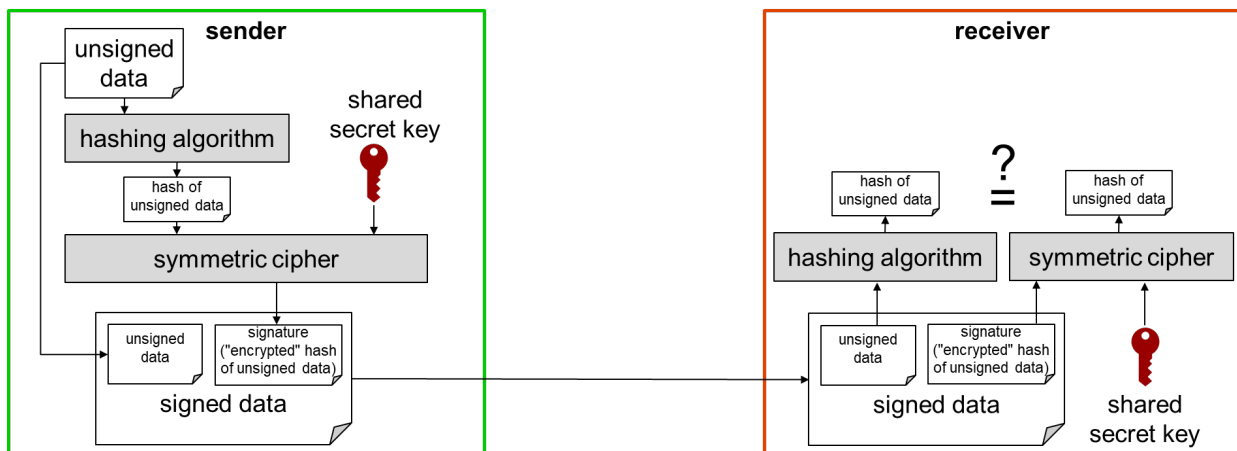


Figure 132: Symmetric signing and verification

EXAMPLE 141: SYMMETRIC SIGNING AND VERIFICATION WITH OPENSSL

With openssl you first have to hash the unsigned data

```
echo "Transfer 300$" | openssl dgst -sha256
```

will produce the hash

```
78c388063282bf66c118cf9a40e92c8f10ae258cc81befdf368c73d6d496e2a1.
```

This hash value is encrypted

```
echo "78c388063282bf66c118cf9a40e92c8f10ae258cc81befdf368c73d6d496e2a1" | \
openssl enc -aes-256-cbc -k yourSecretKey -a -pbkdf2
```

which yields an encrypted hash value

```
"U2FsdGVkX196uyzhGy1zdrhUH2a6k80UeVPU4YCN4/5PrJUjb4QTz7bwZyLm1Ng1XwmhHMD+Amk8WZD8
qWo3iqr0XCmcEAOQN5MJxoYqr3m8ClvqVe2P26h69pHbD0Sj"
```

This encrypted hash value can be decrypted like so:

```
echo
"U2FsdGVkX196uyzhGy1zdrhUH2a6k80UeVPU4YCN4/5PrJUjb4QTz7bwZyLm1Ng1XwmhHMD+Amk8WZD8
qWo3iqr0XCmcEAOQN5MJxoYqr3m8ClvqVe2P26h69pHbD0Sj " | \
openssl enc -d -aes-256-cbc -k yourSecretKey -a -pbkdf2
```

and equals the original hash value.

30.2.4 Asymmetric Ciphers

Asymmetric encryption algorithms work with key pairs. A data record X can be converted by a cipher and a **public key** to a data record Y and converted back with a cipher and a **private key**. As the name suggests, a public key is not confidential and can be blissfully transported over insecure network connections. A private key, however, must be kept hidden from the outside world, i.e., (copies of it) must never leave the filesystem of a communicating node and it should itself be symmetrically encrypted using a passphrase as the symmetric key.

The most widely used asymmetric ciphers are [Rivest-Shamir-Adleman \(RSA\)](#) and [Diffie-Hellman \(DH\)](#) and their respective variants.

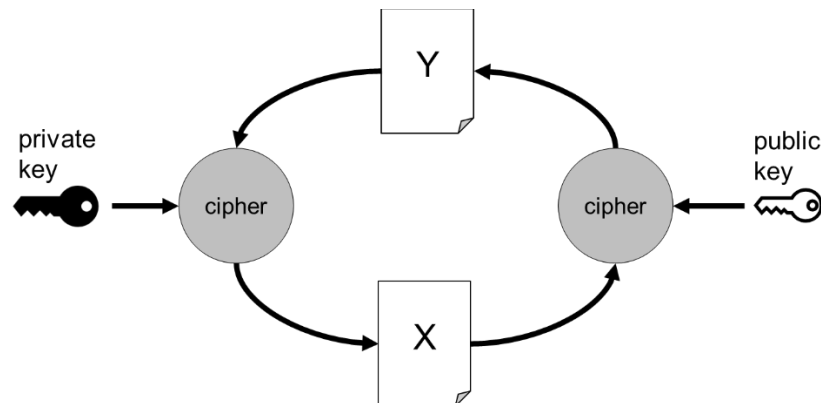


Figure 133: Asymmetric ciphers

EXAMPLE 142: GENERATING KEY PAIRS WITH OPENSSL

With `openssl`, we can create a private key `private.key` and a public key `public.key`. We use the asymmetric cipher RSA.

```
openssl genrsa -aes256 -out private.key \  
-passout pass:privateKeyPassphrase 4096  
  
openssl rsa -in private.key -passin pass:privateKeyPassphrase \  
-pubout -out public.key
```

a) Encryption and Decryption

Key pairs can be used to en- and decrypt data.

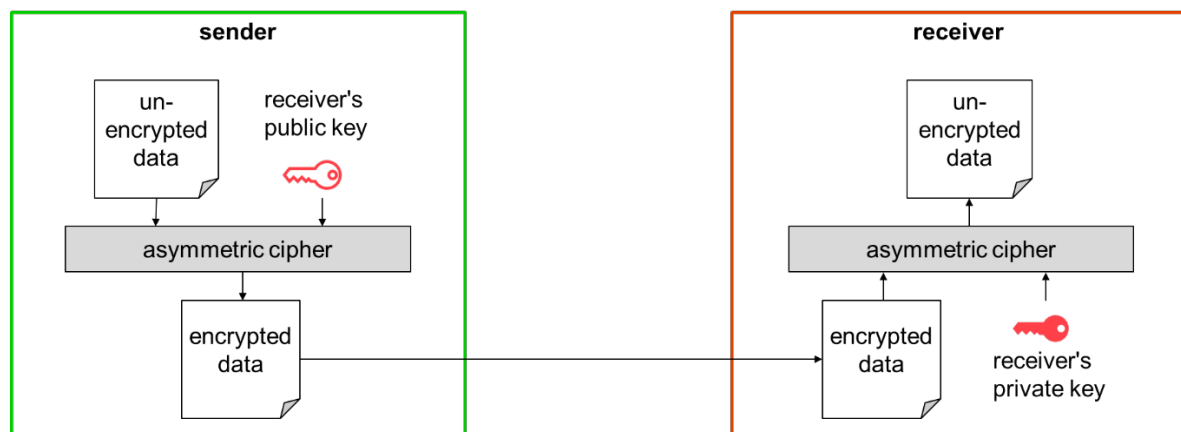


Figure 134: Asymmetric encryption and decryption

Encryption

A sender uses the receiver's [public key](#) and the unencrypted data as inputs for an asymmetric cipher in order to obtain encrypted data (a.k.a. [ciphertext](#)) as output.

EXAMPLE 143: ASYMMETRIC ENCRYPTION WITH OPENSSL

We can use the asymmetric cipher RSA to encrypt a file `unencrypted_data` with content

```
echo "transfer 300$" > unencrypted_data
```

like so:

```
openssl pkeyutl -encrypt -pubin -inkey public.key \
-in unencrypted_data -out encrypted_data
```

Decryption

The receiver uses its **private key** and the encrypted data as inputs for the asymmetric cipher to restore the unencrypted data.

EXAMPLE 144: ASYMMETRIC DECRYPTION WITH OPENSSL

Conversely, we can decrypt the encrypted_data with the private.key.

```
openssl pkeyutl -decrypt -inkey private.key \
-passin pass:privateKeyPassphrase \
-in encrypted_data -out unencrypted_data
```

Please note that if a sender encrypts data with a receiver's public key and delete the unencrypted original data, it cannot decrypt the encrypted data without the corresponding private key, which is only in the possession of the receiver.

b) Signing and Verifying

Key pairs can also be used to sign and verify data.

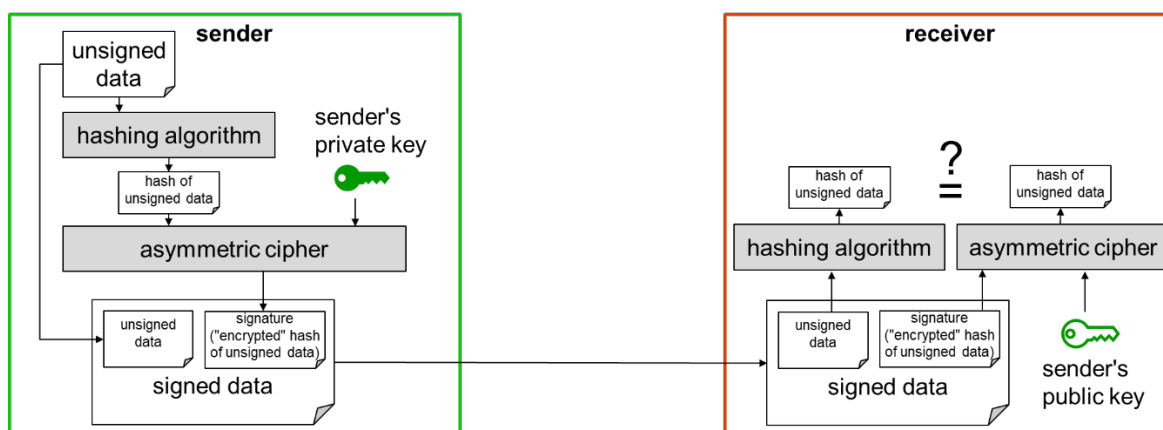


Figure 135: Asymmetric signing and verifying

Signing

On the sender's side, the original unsigned data is hashed. That hash value is now encrypted with the sender's private key which yields a **signature**. The signature and the unsigned data are then transferred to the receiver

EXAMPLE 145: ASYMMETRIC SIGNING WITH OPENSSL

Again, with openssl: Say, we want to sign a file unsigned_data with content transfer 300\$

```
echo "transfer 300$" > unsigned_data
```

Then we use the private.key to sign the unsigned_data, i.e., create a signature. Note that this does the hashing and encryption in a single step.

```
openssl dgst -sha256 -sign private.key \
-passin pass:privateKeyPassphrase -out signature unsigned_data
```

Verifying

The receiver tries to derive the very same hash value by (a) hashing the unsigned data again and (b) decrypting the signature with the sender's public key. If both hash values are equal, the signature is considered valid.

EXAMPLE 146: ASYMMETRIC VERIFICATION WITH OPENSSL

Continuing the singing example, verification with openssl works like this:

```
openssl dgst -sha256 -verify public.key \  
-signature signature unsigned_data
```

This does the hashing, decryption, and comparison in one call, and outputs

Verified OK

Encryption and signing can be combined. In such a combination, either

- the original data is first signed and then the signed data (with the signature) is encrypted or
- the original data is first signed and then the encrypted data is signed.

c) Key Exchange

Another use case for asymmetric encryption is to derive a shared secret key which is private to the sender and receiver without ever transmitting it (even in encrypted form). Asymmetric key exchange algorithms rely on the fact that some mathematical functions can be implemented, such that they work efficiently, whereas their inverse function cannot.

"Color" Key-Exchange

A simple analogy that uses color mixtures instead of mathematical operations is shown in Figure 136.

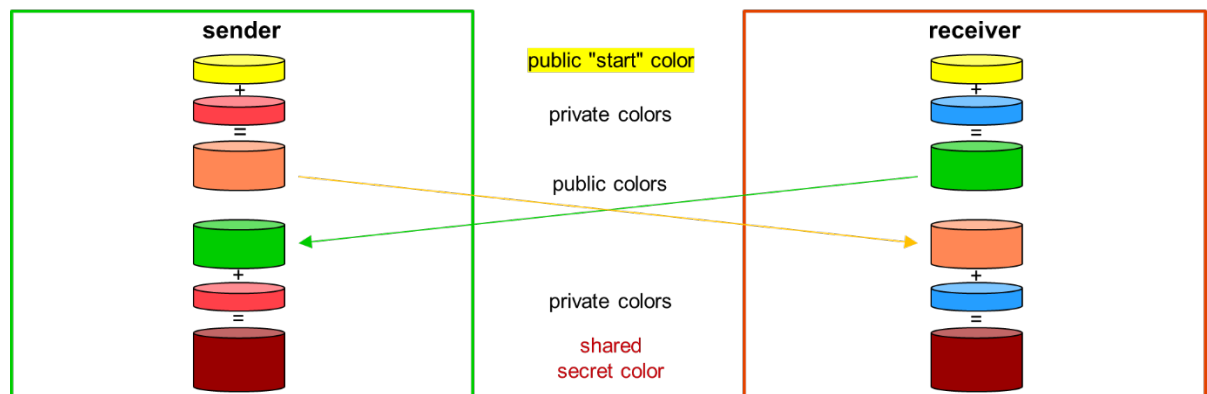


Figure 136: Derivation of a shared symmetric key with "color exchange"

We assume, that it is hard to "subtract" colors, i.e., that—like on the sender's side—red was added to yellow in order to get an orange or—like on the receiver's side—that blue was added to yellow in order to get green. The yellow is a start parameter which is publicly known. Red and blue are the private colors which are only known by the sender and receiver, respectively. The mixtures orange and green are public again and exchanged between sender and receiver. If the sender and receiver now add their respective private color to the other's public color, then both arrive at the same shared secret color.

Diffie-Hellman Key Exchange

Diffie-Hellman (DH) is a key exchange algorithm that is based on cyclic groups and exponentiation. Figure 137 shows the math along with an example.

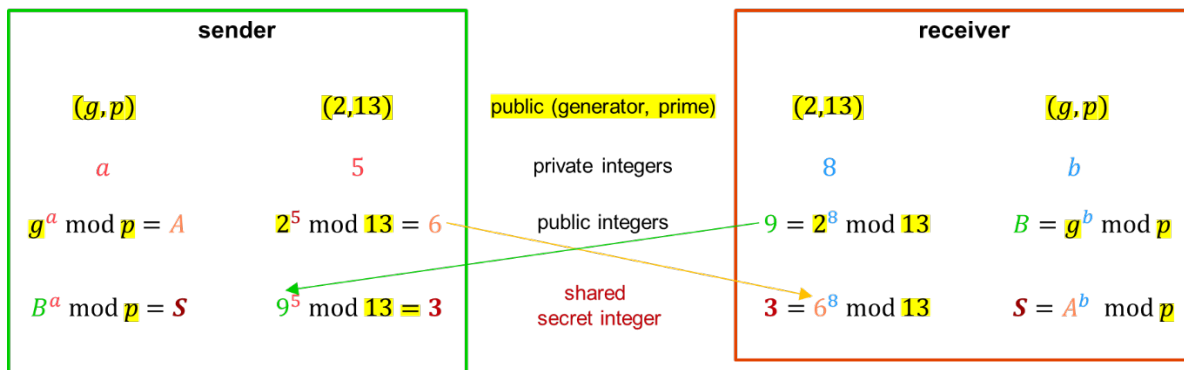


Figure 137: Derivation of a shared symmetric key with Diffie-Hellman key exchange

Please note that in real settings the value of the so-called generator g does not matter much and is therefore often chosen to be 2 or 5. Conversely, the prime p is a very large prime number of typically 2048 bits length. The private integers a and b are randomly chosen from the set $\{1, \dots, p-1\}$.

EXAMPLE 147: DIFFIE-HELLMAN PRIME NUMBER GENERATION WITH OPENSSL

One can use

```
openssl dhparam -2 -text
```

to generate such a prime number p with generator $g := 2$, for example (and after conversion to a decimal number):

```
208552955314896958588775557314503473085446963503506024263427231538367907487658656
772857476124778179949993783306514895485710479647418081394311257546876107534778148
990003449339352413769483532231507390127377462876289171209418154180532822060051644
820302580914090332871399656448078105672616791535719528512582355872496462881748678
312437182544795092980972718896493370091906768008920581396561252426859495255671487
957009591089357050404572105777908876980217481770957226063911488821568514428889682
192016136634618283114166436689138956071529885285251409001668574458032258995971380
67143134645532638555020167507010393130754055334243
```

Diffie-Hellman is based on the fact, that it is computationally easy to compute A from g, a, p , but there is no known efficient algorithm that can compute a from given g, A, p . In other words, discrete exponentiation in cyclic groups is easy, but computing the discrete logarithm is not. The best-known algorithms that compute the discrete logarithm have a runtime that grows linear with value of p and therefore exponential with the number of digits in p .

Elliptic Curve Diffie-Hellman Key Exchange (ECDH) works analogously to Diffie-Hellman. However, it is based on mathematical objects called elliptic curves instead of cyclic groups and therefore uses point addition instead of multiplication, and scalar multiplication instead of exponentiation.

30.2.5 Key Derivation

The shared secret integer which results from a key exchange normally does not serve as a symmetric key as-is. Instead, it is a **pre-master secret** and used as input for a **Hash Key Derivation**

Function (HKDF) which yields the actual symmetric key. As opposed to the pre-master key, the actual symmetric key has some desirable properties like a certain length (128, 192 or 256 bits in case of AES). The secrecy of the pre-master secret suffices to guarantee secrecy for the actual symmetric key secret.

30.2.6 Public Key Certificates

Asymmetric ciphers prove nothing about the identity of communicating parties. Senders and receivers remain anonymous to each other. Authentication can be achieved, however, with **public key certificates**.

A public key certificate is a data record that proves that a **subject** (human or server) owns a certain public key. By "owns" we mean that the subject has access to the private key that corresponds to the public key. Of course, the subject itself cannot be included in a certificate. The certificate instead contains a set of attributes (name, organization etc.) that in combination only applies to that single subject. If the subject is a server that has the private key somewhere in its mounted file system, the server can prove that some domain name is really assigned to it. If the subject is a human, he or she can prove that a certain full name or email address apply to him or her.

X.509 [150] is the most common data format for public key certificates and specifies a hierarchical name-value schema which is defined in Abstract Syntax Notation One (ASN.1) [151]. The full explanation all names in a X.509 certificate is not in scope of this document.

EXAMPLE 148: X.509 SERVER CERTIFICATE WITH OPENSSL

In a bash shell however, one can print a server certificate from an arbitrary website in X.509 format and textual encoding using

```
openssl s_client -connect www.example.com:443 </dev/null 2>/dev/null \  
| openssl x509 -text -noout
```

in order to get an impression of what X.509 looks like:

Certificate:

Data:

Version: 3 (0x2)

Serial Number:

0f:be:08:b0:85:4d:05:73:8a:b0:cc:e1:c9:af:ee:c9

Signature Algorithm: sha256WithRSAEncryption

Issuer: C = US, O = DigiCert Inc, CN = DigiCert TLS RSA SHA256

(...)

Subject: C = US, ST = California, L = Los Angeles,

O = Internet Corporation for Assigned Names and Numbers,

CN = www.example.org

Subject Public Key Info:

Public Key Algorithm: rsaEncryption

RSA Public-Key: (2048 bit)

Modulus:

00:ba:fc:ee:cc:ca:0a:08:ff:0e:93:1d:b3:be:0b:

9c:03:96:22:9e:b1:4f:10:ae:51:40:fd:53:5f:b3: (...)

Exponent: 65537 (0x10001)

X509v3 extensions:

X509v3 Subject Alternative Name:

DNS:www.example.org, DNS:example.com, (...)

(...)

Signature Algorithm: sha256WithRSAEncryption

a7:2a:10:30:5c:b8:6b:7a:1b:f8:66:38:f6:e9:a0:0a:d5:13:

82:82:f8:65:89:57:a5:b8:eb:13:29:1d:84:6c:ec:fb:e3:05: (...)

The actual certificate data can be found under a complex key called Data.

One of the names in the certificate data is subject. Its value is a list of name-values which form the so-called **distinguished name** of the subject. These name-values provide identification properties of the subject: If the subject is a server (which is the case in the context of TLS), one of its sub-name-values looks like CN = www.example.com, which basically claims that the domain name—or **common name (CN)** in X.509 speak—of some server really is www.example.com. If a single server has multiple domain names, these are listed with DNS beneath the X509v3 Subject Alternative Name. Additionally, the static public key of the server is included via Subject Public Key Info.

Of course, it would be easy for an adversary just to forge the Data section of an X.509 certificate and let his or her malicious server answer with that forged certificate including a fraudulent common name. Therefore, X.509 certificates include a cryptographic signature (beneath Signature Algorithm), which signs the Data section. The static private key used for the signature is that of so-called **certificate authority (CA)**. Such CAs are trustworthy companies that provide signatures only to those subjects they trust. How one can convince a CA to trust you and that you rightfully possess a certain domain name is out of scope of this document. The distinguished name of the CA that provided the signature can be found in the Data section beneath the key Issuer.

By signing a certificate, all parts in the Data section are in a way "bound together". In particular, the Subject (including the CN) and the Subject Public Key Info are bound together. This proves the ownership of the public key by the subject (server). Thus, a server just needs to prove that it possesses the static private key that corresponds to the public static key in Subject Public Key

Info. By doing so, the server also proves that everything else in the Data section applies to the server, the CN in particular.

30.2.7 Certificate Chains

The CA's public key can be found in another certificate, i.e., that of the CA. However, an adversary now just could act as a CA and forge such a certificate. A client knows that can it trust the CA's certificate if it is installed in a local database which is called **truststore** (a.k.a. **keystore**) and can be found in the local filesystem⁶⁶. Please note that a truststore needs to be protected by a symmetric cipher and appropriate Linux file permissions: If an adversary manages to install his or her certificate in a client's truststore, he or she can indeed undermine security.

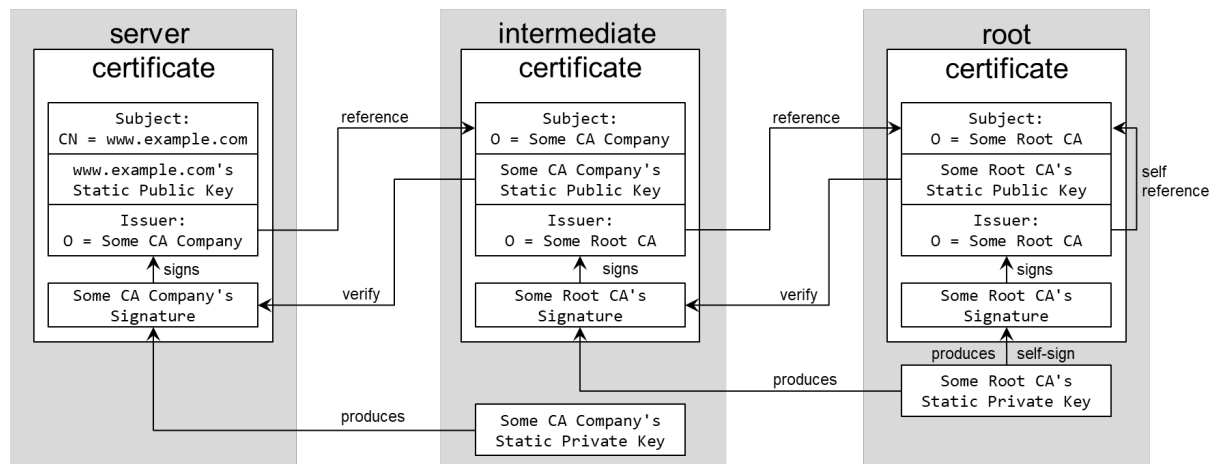


Figure 138: Certificate chain

In practice, CAs arise and cease business quite frequently. Constantly updating truststores of each client in the world with CA certificates is however unfeasible. This is why a **server certificate** is signed by a CA, whose certificate is just an **intermediate certificate**. Such intermediate certificates are not included in the default trust stores that come along with common operating systems or common clients (browsers). An intermediate certificate can be signed by a CA which also possesses an intermediate certificate or (finally) by a so-called **root certificate**. A root certificates signs itself and is typically already included client's trust store.

A server certificate (a.k.a. end-entity certificates), a (or potentially several) intermediate certificates, and a root certificate form a **certificate chain**. So, the client does not need to find the server certificate itself in the client's trust store. Instead, the client just needs to find a root certificate in its trust store that signs a certificate on the certificate chain, which is typically the (topmost) intermediate certificate.

30.2.8 Certificate Formats

X.509 merely defines the data structure for certificates using ASN.1. Certificate usually do not come along in plaintext like the example in Section 30.2.6. Instead, there are encoded in Base64, which makes them embeddable in other text files (like configuration files). Alternatively, they are encoded in a proprietary binary format. Unfortunately, there is an abundance of standards about how to do the encoding and associated file extensions.

⁶⁶ Clients that run on the same host share the filesystem and can thus share a common trust store. However, there is no real standard how (in which format) and where (in the filesystem) trust stores have to be stored and located, respectively. For Linux, common locations are /etc/pki, /etc/ssl, /etc/openssl

Encoding	Format	Contents / Usage	File Extensions
Base64 encoded ASCII	PEM	Certificates (Chain) + Private Key	.pem, .crt, .cer, .key
	PKCS#7	Certificates (Chain)	.p7b, .p7c
Binary	DER	Certificate (Chains)	.der, .cer, .crt
	PKCS#12	Certificates (Chain) + Private Key Used on Windows	.pfx, .p12

Table 48: X.509 encodings and formats

30.3 Transport Layer Security

Transport Layer Security (TLS) is an internet protocol that allows for confidential and integrous transmission of Transmission Control Protocol (TCP) frames between nodes in a network. It also provides means for unilateral and mutual authentication of communicating nodes. However, neither end-user authentication nor authorization is in scope of TLS. As by 2020, the most recent revision of TLS is 1.3, which we will investigate in the following.

Since TLS resides on directly top of TCP in the ISO/OSI protocol stack, several application layer protocols have variants that make use of this underlying TLS layer like HTTP over TLS (HTTPS) [85, Sec. 4.2.2] or POP3S [152]. In March 2025, over 87% of all web sites use HTTPS as their default protocol [153].

Even if violations of authenticity, confidentiality and integrity poses no actual risk for a certain web application, securing its network traffic with HTTPS is best practice with regard to Search Engine Optimization (SEO) [154].

30.3.1 Cipher Suites

TLS is a **hybrid cryptosystem** as it includes several ciphers which are used while a TLS session is established (TLS handshake protocol) and during a TLS session (TLS record protocol). A **cipher suite** a combination of such algorithms each serving a different purpose. Since TLS 1.3, the term cipher suite merely refers to a combination of the latter two cipher types of Table 49.

Cipher Type	Explanation and Use in TLS	Common Algorithms (Categories) (outdated)
Key Exchange Groups	Parameterized asymmetric cipher used in the beginning of a TLS handshake in order to derive a pre-main key for symmetric bulk encryption algorithms over a so far insecure TCP network connection.	— Rivest-Shamir-Adleman (RSA) - Diffie-Hellman (Ephemeral) (DH(E)) - Elliptic-curve Diffie-Hellman Ephemeral (ECDH(E)) - Pre-Shared Keys (PSK)
Digital Signature Algorithm	Parameterized asymmetric cipher by which the server signs so-far exchanged handshake message and thereby authenticates against the client and optionally vice versa.	- Rivest-Shamir-Adleman (RSA) - Edwards Curve Digital Signature Algorithm (EdDSA) - Elliptic Curve Digital Signature Algorithm (ECDSA)
Symmetric Bulk	Symmetric cipher for en-/decryption of	- Advanced Encryption Standard (AES)

Cipher Type	Explanation and Use in TLS	Common Algorithms (Categories) (outdated)
Encryption Algorithms	application data during the record protocol in order to ensure confidentiality of that data. ⁶⁷	- AES 256 with Galois/Counter Mode (AES 256 GCM) - ChaCha20-Poly1305
Secure Hashing Algorithms	Cipher that ensures integrity of exchanged application data.	- Secure Hash Algorithm 256 (SHA256) - Secure Hash Algorithm 384 (SHA384)

Table 49: Cipher types

30.3.2 TLS Handshake Protocol

In order to securely exchange data over an insecure channel, TLS requires to establish a TSL session. This is done in a so-called **handshake protocol**. Figure 139 depicts a very simplified version of this handshake protocol.

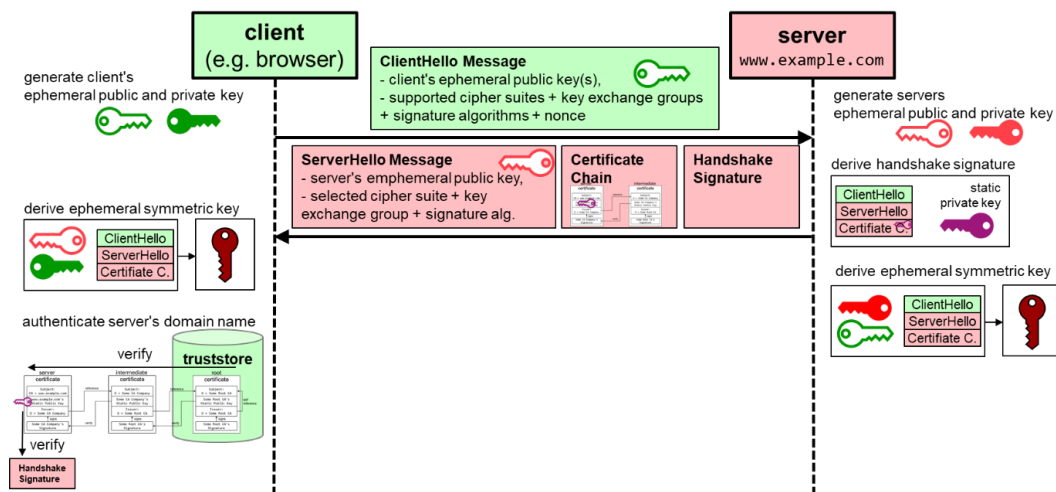


Figure 139: TLS 1.3 single round-trip handshake (simplified)

A new TLS 1.3 session is established in a single client-server roundtrip:

a) ClientHello Message

In the **ClientHello message**, includes the following:

- **Supported TLS version** (either 1.3 or 1.2, older versions have been deprecated since 2021).
- A list of client-supported **symmetric cipher suites** which must be a subset of TLS-1.3-supported cipher suites, which are the following five: TLS_AES_128_GCM_SHA256, TLS_AES_256_GCM_SHA384, TLS_CHACHA20_POLY1305_SHA256, TLS_AES_128_CCM_SHA256, TLS_AES_128_CCM_8_SHA256. Such cipher suites specify a pair consisting of:
 - a **symmetric bulk encryption algorithm**, which are in case of TLS 1.3 **Authenticated Encryption with Associated Data (AEAD)** algorithms like AES_123_GCM and
 - the **Secure Hashing Algorithm** (like SHA256) which is used in a **Hash Key Derivation Function (HKDF)** to derive a symmetric encryption key out of so far exchanged data during the

⁶⁷ Symmetric ciphers are about 10,000 times more efficient than asymmetric ciphers.

handshake process. This so-far exchanged data particularly includes the pre-main key that results on both sides from that the key exchange group.

- A list of client-supported **key exchange groups** (i.e., parameterized algorithms), one of which is used to derive a pre-main key on both sides without actually exchanging it. An example for such a key exchange group is x25519 [155, p. 4], which is an **Elliptic Curve Diffie Hellmann Ephemeral (ECDHE)** algorithm.
- A generated **ephemeral public key**, for each of the client-supported key exchange groups, which is valid just for the duration of the TLS session.
- A list of client-supported digital **signature algorithms**. For example, `rsa_pkcs1_sha384`, which means that data is hashed with a Secure Hashing Algorithm that produces 384-bit hashes (SHA384) which are then signed using a Rives-Shamir-Adleman (RSA) algorithm.

b) **ServerHello Message**

The server selects one of the client-supported cipher suites, key exchange groups and signature algorithms and generates an ephemeral server key pair. The server's ephemeral public key is included in the **ServerHello message** as well as the agreed-on algorithms (like `TLS_AES_128_GCM_SHA256`, `x25519`, `rsa_pkcs1_sha384`). A **certificate chain** containing the server certificate and potentially intermediate certificates are included in a parallel server response. Moreover, the server signs previous handshake messages with its static private key yielding a **handshake signature**.

c) **Ephemeral Symmetric Key (Shared Secret) Computation**

Both the client and the server now can derive the same ephemeral **symmetric key**⁶⁸ as a **shared secret** for bulk symmetric encryption of subsequent (application) data from their ephemeral private the other's ephemeral public key. This computation particularly includes the exchanged ephemeral public keys in order to compute a pre-main key on both sides. Both sides convert to pre-main key to an ephemeral symmetric key using the secure hashing algorithm defined in the cipher suite. Please note that this derived ephemeral symmetric key is not only used for encrypting the application data, which would provide confidentiality, but also for signing it, which adds proof for integrity of that data. Without signing, a man-in-the-middle could otherwise flip some bits in the encrypted data hoping that it would change the unencrypted data in his favor.

d) **Server Authentication**

The client is now also able to **authenticate** the server:

- The client uses the server's static public key found in the received certificate chain in order to verify the handshake messages.
- The client checks if the server's certificate can be derived from the client's trust store.

If both steps succeed, the client can trust everything else in the server's certificate and particularly the domain name(s) as specified by `Subject` or `X509v3 Subject Alternative Name`

Please note that this is just a very abridged version of the handshake protocol. If the client fails to guess and include a cipher suite in the ClientHello message that is supported by the server, an additional roundtrip is necessary. If the client and server already had a TLS session, they can re-

⁶⁸ Actually, a set of keys where each different key is used depending on the communication direction. However, the crucial point here is that client and server can derive the very same output data from different input data, which is own ephemeral private key, other's ephemeral public key, and exchanged messages.

establish a TLS session in a 0-RTT, i.e., the client can directly send application data. However, this bears a security downgrade with respect to forward secrecy.

30.3.3 TLS Record Protocol

After the handshake, the ephemeral symmetric key is used in the TLS record protocol in order to encrypt and sign application data. Note that signing is mandatory in addition to encryption in order to rule out tampering with the encrypted data.

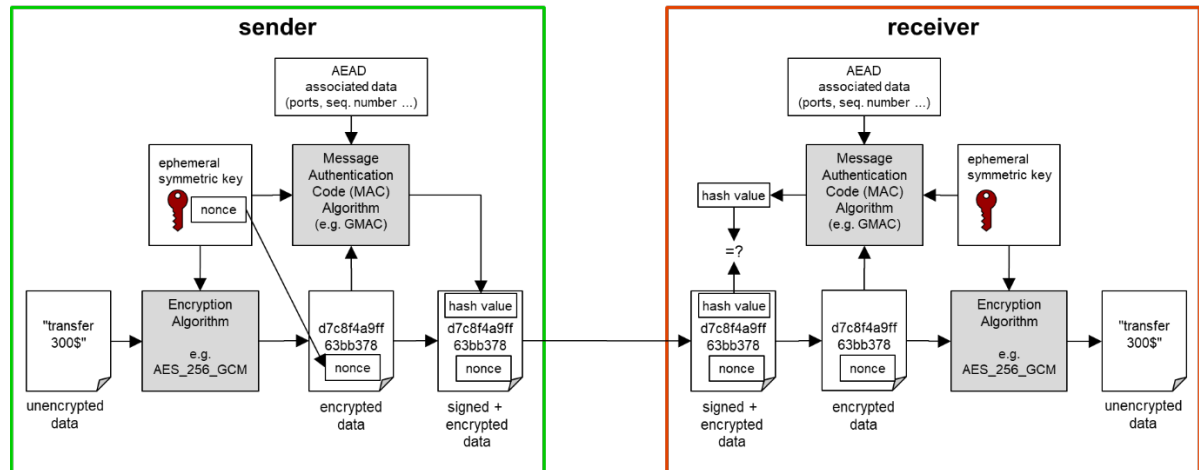


Figure 140: TLS 1.3 record protocol

As Figure 140 shows that an encryption algorithm like AES_256_GCM takes the ephemeral symmetric key, the nonce from the ClientHello message and the unencrypted application data as input in order to produce encrypted data. The nonce is padded to that encrypted data.

A **Message Authentication Code Algorithm (MAC)** like GMAC uses this encrypted data, the shared secret and the nonce plus **Authenticated Encryption with Associated Data (AEAD)** data, which only the sender and receiver know of and produce a hash value, which symmetrically signs the encrypted data.

On the receiver side, the encrypted data together with the ephemeral symmetric key and AEAD is fed again into the MAC algorithm. The receiver knows that the encrypted data is authentic if the produced hash value equals the one which came as the signature because just the sender knows about the shared secret, too. The receiver also knows that the encrypted data is integrous because it is practically impossible to tamper with the encrypted data in a way that the MAC still produces the same hash value. The encrypted data is then fed into the encryption algorithm along with the shared secret. This yields the unencrypted data again.

A more detailed explanation of TLS 1.3 can be found in [156], [157].

30.4 OAuth2

For contemporary web applications, it is common that different systems access each other's web resources (cf. Section 27.1.1). Yet, such resources might be confidential, e.g., they must not be shared without proper authorization on behalf of the user who owns the web resource.

OAuth2, which stands for "Open Authorization Version 2", as specified by RFC 6749 [158], provides HTTPS-based protocols that define the interaction and data exchange between

- a user, which is a **resource owner**,
- a so-called **client**, which requests—hence client—a certain remote protected web resource,

- which is stored on a **resource server** that demands access tokens in order to authorize access to the protected resource,
- an **authorization server**, which is a kind of identity provider (cf. Section 23.12) and able to authenticate a resource owner by means of, e.g., username-password credentials and which issues security tokens, and
- a **user agent** by means of which the resource owner interacts with the authorization server and the client.

With OAuth2, the resource owner can authorize the client to access (some of) the resource owner's resources which are stored on the resource server.

OAuth2 alone does not force the resource owner to reveal much of his or her identity. In principle, it suffices to pass around a pseudonymous but constant identifier for each resource owner such that clients and resource servers can (re-)identify a resource owner and existing data associated with him or her.

30.4.1 Credentials and Security Tokens

OAuth2 mainly revolves around the exchange of credentials. Credentials can be presented by end-users, like username-password combinations. These kinds of credentials have to trade off security against usability and are thus rather static, i.e., expire rarely, and cryptographically insecure, i.e., short and not random. Conversely, **security tokens** are credentials that are stored and exchanged by communicating nodes. Security tokens are ephemeral and cryptographically secure by design and can therefore evade the downsides named above.

30.4.2 Clients

OAuth2 distinguishes between public and confidential clients.

a) Confidential Clients

The client could itself be a server in classic web application. In this case a **user agent** acts as this client's client and is most likely a browser. The user agent mediates the interactions between the resource owner, the client, and the authorization server. Confidential clients can keep security tokens physically out of the reach from their users (resource owners) and other potentially malicious processes which run on the same machine as the user agent.

b) Public Clients

A public client is a program that runs locally on a resource owner's device and provide a user interface on its own. Smartphone native apps and desktop applications are of this kind. If a web application is implemented in a way that it runs completely in the user agent, like certain single-page applications, it is also considered a public client. Such public clients may not be capable of hiding credentials from the end-user (resource owner) or other (malicious) local processes.

30.4.3 Endpoints

There are different endpoints (URLs) that are crucial for OAuth2: An authorization server exposes an

- **Authorization endpoint**, which is used to start an authentication + authorization web flow between the user agent and the authorization server, and which finally yields an authorization code,
- **Token endpoint**, which is used by the client to convert an authorization code to a security token called access token, which can be used by the client to authorize with the resource server,

- **Introspection endpoint** if RFC 7662 [159] is supported, by which the resource server can convert an opaque access token into something meaningful,
- **JWK endpoint** if RFC 7517 [160] is supported, by which the resource server can obtain a public key from the authorization server in order to verify the access token,
- **Discovery endpoint**, if [161] is supported, which responds with a JSON that particularly includes the URLs for the endpoints explained above.

The client exposes a **redirection endpoint**, which is used by the user agent to forward the authorization code to the client.

30.4.4 Client Registration and Authentication

OAuth2 requires that the client and authorization server trust each other. A client must therefore register with an authorization server. During this registration, the authorization server and the client have to agree on a

- unique **client identifier** and
- a **client secret**,
- a list of or pattern for valid **redirection URLs**, which define the client's redirection endpoint and are required during the later authorization process.

Figure 141 exemplifies the web flow.

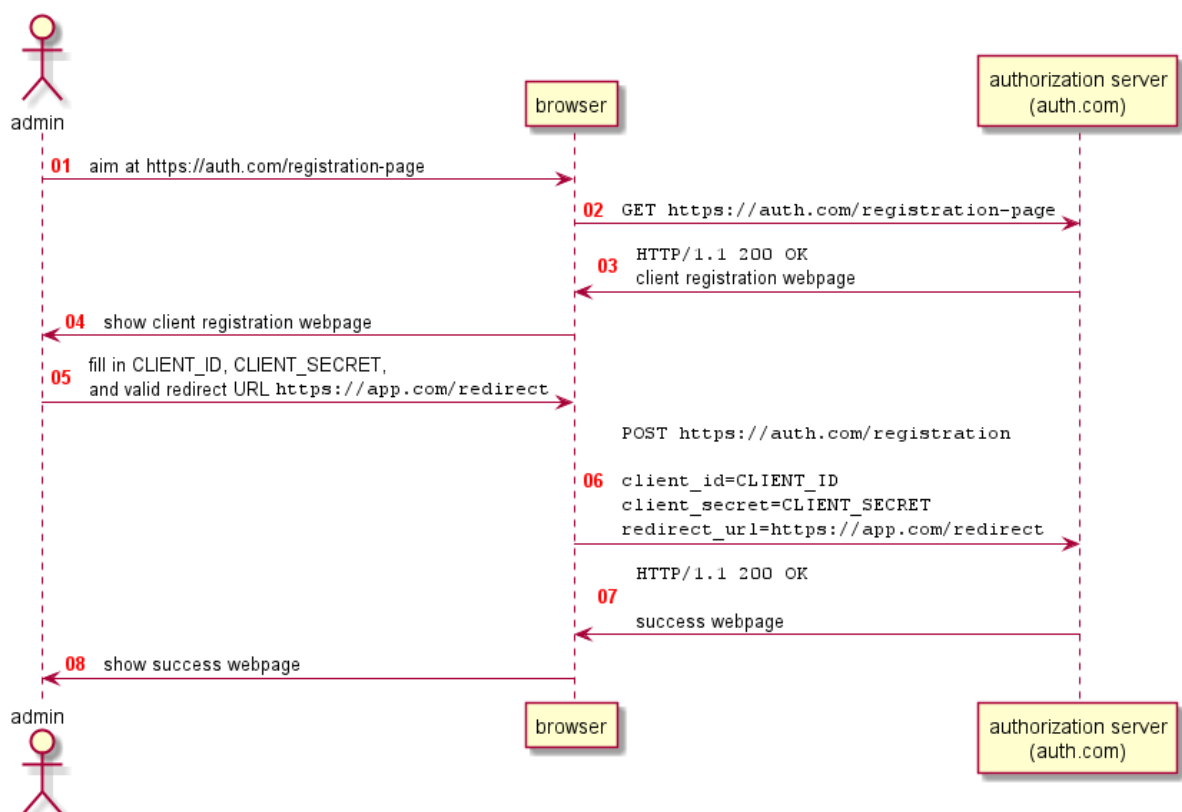


Figure 141: OAuth2 client registration

After registration, the client can put a Basic Authentication [162] header in each HTTP request for the authentication server, which contains the Base64 encoded `CLIENT_ID:CLIENT_SECRET`, in order to identify and authenticate with the authorization server.

30.4.5 Authorization Code Grant Flow

OAuth2 specifies different flows and usages of the endpoints in Section 30.4.3. The purpose of all these flows is to securely convey an access token to the client by means of which the client can access a protected resource on a resource server. For web applications with browsers as user agents and confidential clients in the sense of Section 30.4.2a), the most suitable flow is called **Authorization Code Grant**.

The authorization server is normally unable to send HTTP requests directly to the client and is also not aware of the session between the browser and the client. The browser must therefore be used as a relay which is not trusted to be safe from accesses by malicious programs on the same machine or from the resource owner him- or herself. An additional intermediate code (authorization code) and additional HTTP messages are therefore required. Only by presenting a combination of the **CLIENT_SECRET**, which is only the client possesses but not the browser, and the **AUTHORIZATION_CODE**, the authorization server issues an access token. This access token suffices to authorize requests to the resource server and is therefore just exchanged between the client and resource server, but not with the browser.

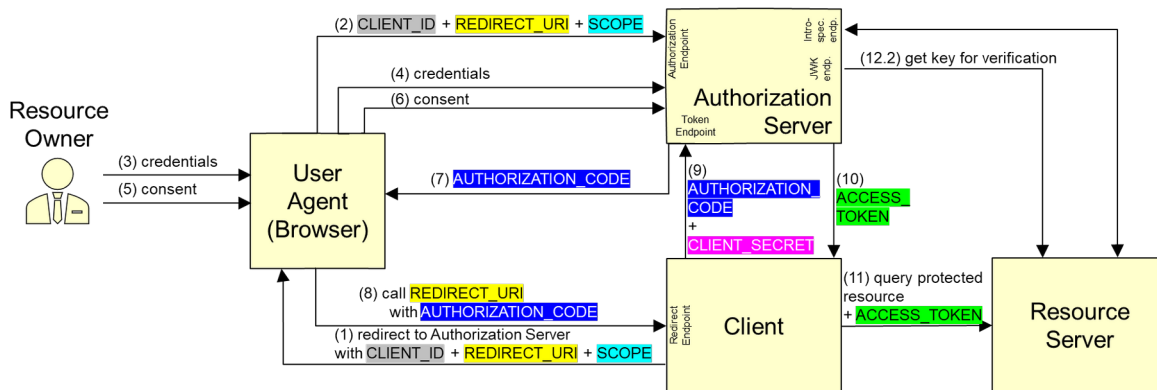


Figure 142: OAuth2 ACG flow - overview

Figure 142 and provides a cut-out overview example of the Authorization Code Grant flow with just the crucial codes/tokens and messages.

1. It starts with the client realizing that it needs authorization for accessing a protected resource on the resource server. Therefore, it redirects the browser to the authorization endpoint of the authorization server.
2. The browser includes the **CLIENT_ID**, the **REDIRECT_URI** and (optionally) the requested **SCOPE** in the request to the authorization endpoint
3. The authorization server responds with a login page, in which the resource owner can present (type in) his or her credentials (typically username and password).
4. These credentials are forwarded (POSTed) from the browser to the authorization server.
5. The authorization server responds with a consent page with a list of requested grants. The resource owner acknowledges, e.g., by clicking "yes".
6. The consent is forwarded to the authorization server.
7. The authorization server responds with an **AUTHORIZATION_CODE**. Please note that this **AUTHORIZATION_CODE** alone will not grant access to the protected resource on the resource server.
8. Instead, the browser calls the client's **REDIRECT_URI** including the **AUTHORIZATION_CODE**.
9. The client combines the **AUTHORIZATION_CODE** with the **CLIENT_SECRET** (from Section 30.4.4) in a call to the token endpoint of the authorization server.

10. The authorization server responds with an **ACCESS_TOKEN**.
11. The client includes the **ACCESS_TOKEN** in a request to the protected resource on the resource server.
12. The resource server verifies the **ACCESS_TOKEN**.
 1. Either the **ACCESS_TOKEN** is exchanged for a plain text claims set in a call to the introspection endpoint (and thereby verified) or
 2. The resource server obtains an ephemeral public key from the JWK endpoint in order to verify a claims set which is encoded in **ACCESS_TOKEN** itself.

Figure 143 to Figure 148 complete the example with all messages and code/tokens that are exchanged.

Some facts are noteworthy about Figure 143 to Figure 148:

- The example is end-to-end. It starts with a resource owner navigating to a web page that renders a protected resource. It ends with that web page being rendered. Actually, the Authorization Code Grant flow just covers steps in of Sections 30.4.5a) to d). The subsequent steps are included in order to provide a complete understanding.
- It is easy to confuse "authentication" with "authorization". Readers should take extra care not to do so.
- SNAKE_CASE_UPPERCASE strings actually denote variable values. This includes **CLIENT_ID**, **CLIENT_SECRET**, **SESSION_STATE**, **USER**, **PASSWORD**, **AUTHORIZATION_CODE**, **ACCESS_TOKEN** and **REFRESH_TOKEN**. One can imagine these values as (apparently) random strings of some length
- Other values have been exemplified by "speaking" concrete values. This particularly includes the endpoints
 - Client Redirect URI: **https://app.com/redirect**
 - Authorization Endpoint: **https://auth.com/authorization**
 - Token Endpoint: **https://auth.com/token**
 - JWK Endpoint: **https://auth.com/jwk**
 - Introspection Endpoint: **https://auth.com/introspection**
- The interaction with the resource user, i.e., requesting and responding web pages, is also included in Figure 143 to Figure 148 in order to present a realistic overall web flow. The OAuth2 specification actually does not cover these as the web flow could also be implemented in alternative ways. These web pages are served by the following endpoints
 - **https://app.com/protected-resource-page** which is supposed to, say, visualization of the protected resource from **https://res.com/protected-resource**
 - **https://auth.com/authentication-page** which presents a login page to the resource owner
 - **https://auth.com/consent-page** which presents the authorization consent inquiry to the resource owner

The flow goes as follows:

a) Resource owner accesses protected resource

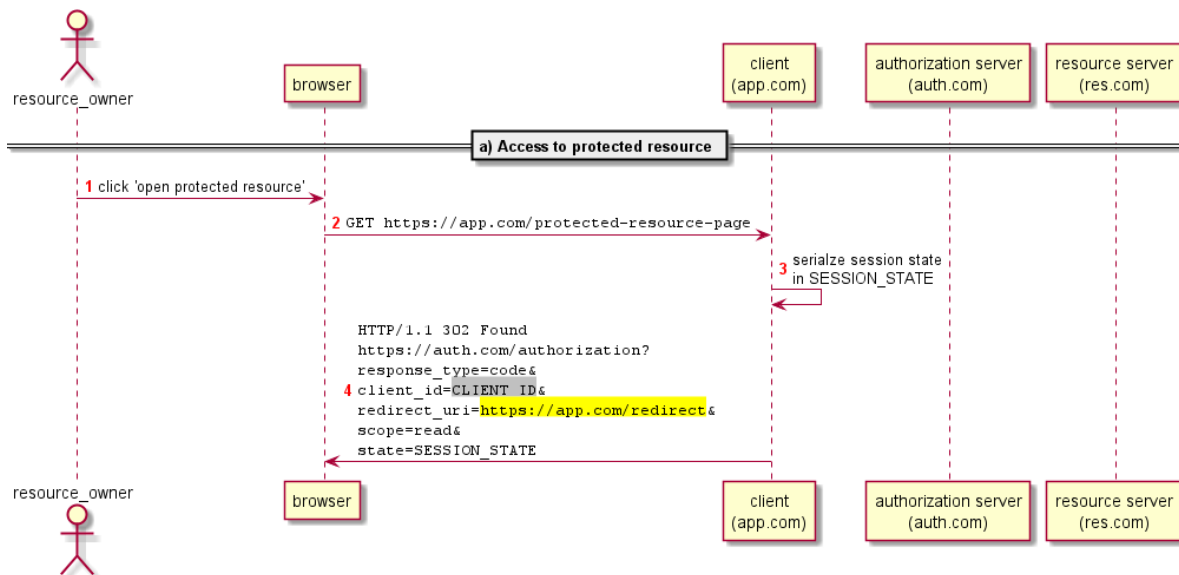


Figure 143: OAuth2 ACG flow - a) Resource owner access protected resource

- Steps 1 and 2: A resource owner aims the browser at `https://app.com/protected_resource-page`. We assume that there is some kind of programming or configuration such that the client knows it must now start the OAuth2 ACG flow before it is able to access the protected resource at `http://res.com/protected_resource`.
- Step 3: The client serializes its current session state, or at least its current navigational state, in order to remember after authorization to redirect to `https://app.com/protected_resource-page`
- Step 4: The client redirects the browser to the authorization endpoint and appends the following URL query parameters:
 - `response_type = code` indicates an authorization code is requested. Other flows directly request the access token at this point using `response_type = token`.
 - `client_id = CLIENT_ID` sets the client id which was set during the client registration (cf. Section 30.4.3)
 - `redirect_uri = https://app.com/redirect` must match a redirect URI pattern that has been set during client registration. After the authorization server obtained authorization from the resource owner, the browser is redirected from the authorization server (`auth.com`) back to the client (`app.com`). Please note that this redirect URL is different from `https://app.com/protected_resource-page`, which is stored in `SESSION_STATE`
 - `scope = read` is optional and indicates restrictions to the requested scope. The value (or space separated values) must be understood at least by the resource server. They must be also understood by the authorization server, if they should be presented in a human-readable form in the consent page presented in Step 15. Otherwise, the authorization server can just present these scope values in their original, literal form.
 - `state = SESSION_STATE` contains the serialized client session state of Step 3.

b) Resource owner authenticates with authorization server

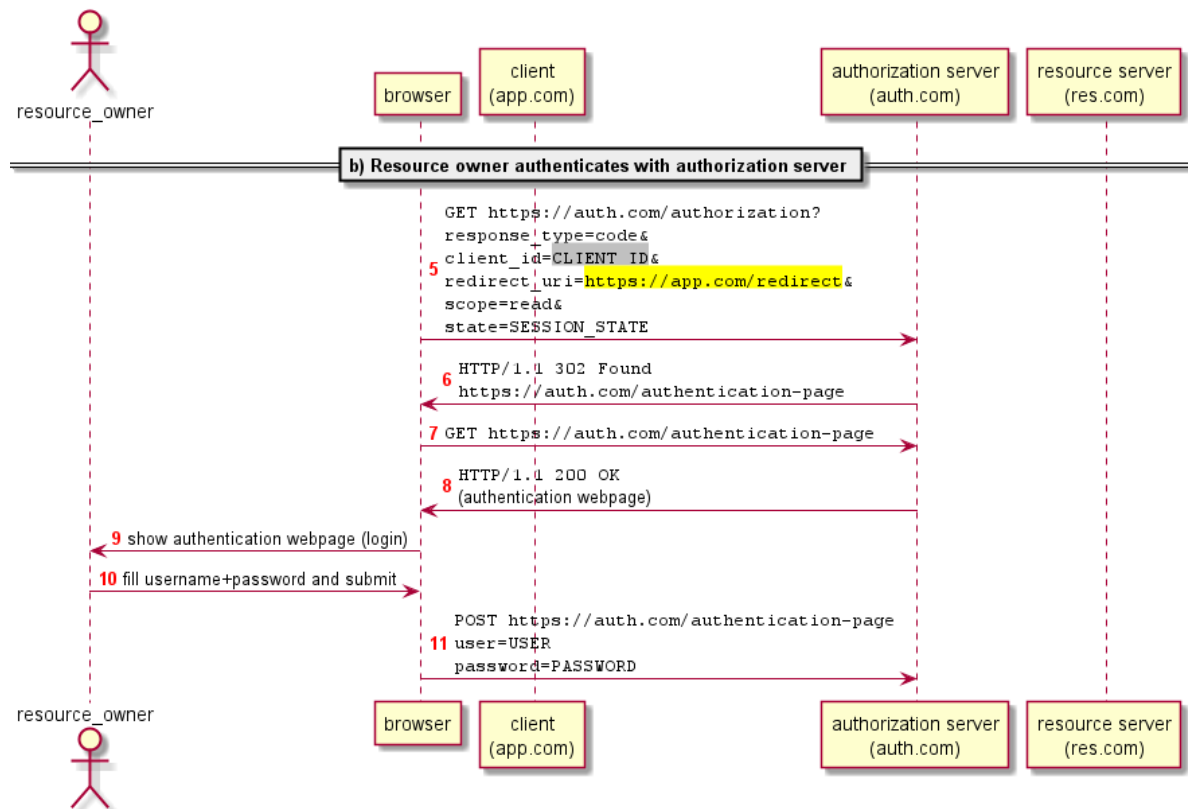


Figure 144: OAuth2 ACG flow – b) Resource owner authenticates with authorization server

- Step 5: The browser follows the redirect and thereby initiates the resource owner authentication and authorization consent flow.
- Steps 6 to 11: These steps implement the authentication of the resource owner with the authorization server and the consent for authorization. As these steps are not covered by the OAuth2 specification they might be implemented differently as well.

c) Authorization server asks resource owner for authorization consent

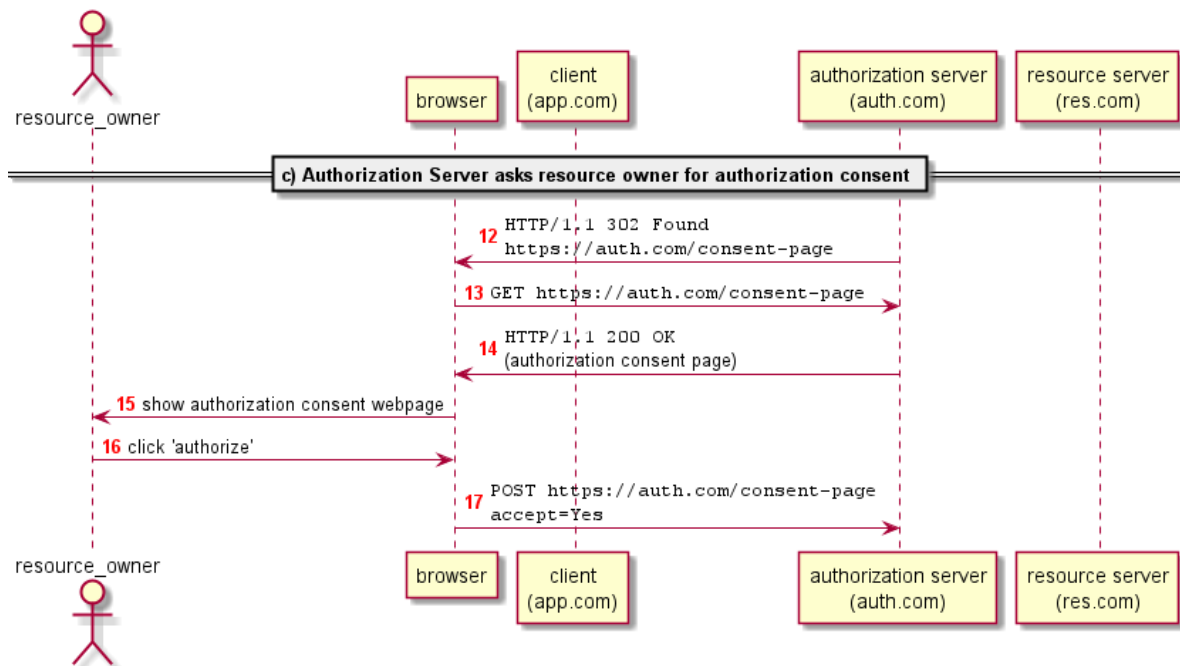


Figure 145: OAuth2 ACG flow – c) Authorization server asks resource owner for authorization consent

- Steps 12 to 17: These steps implement how the authorization server obtains the actual authorization consent from the resource owner. Typically, this is done via a dedicated webpage or pop-up, which presents the requested scopes and the client that asks for it in a human-readable form.

d) Client obtains access token in exchange for authorization code

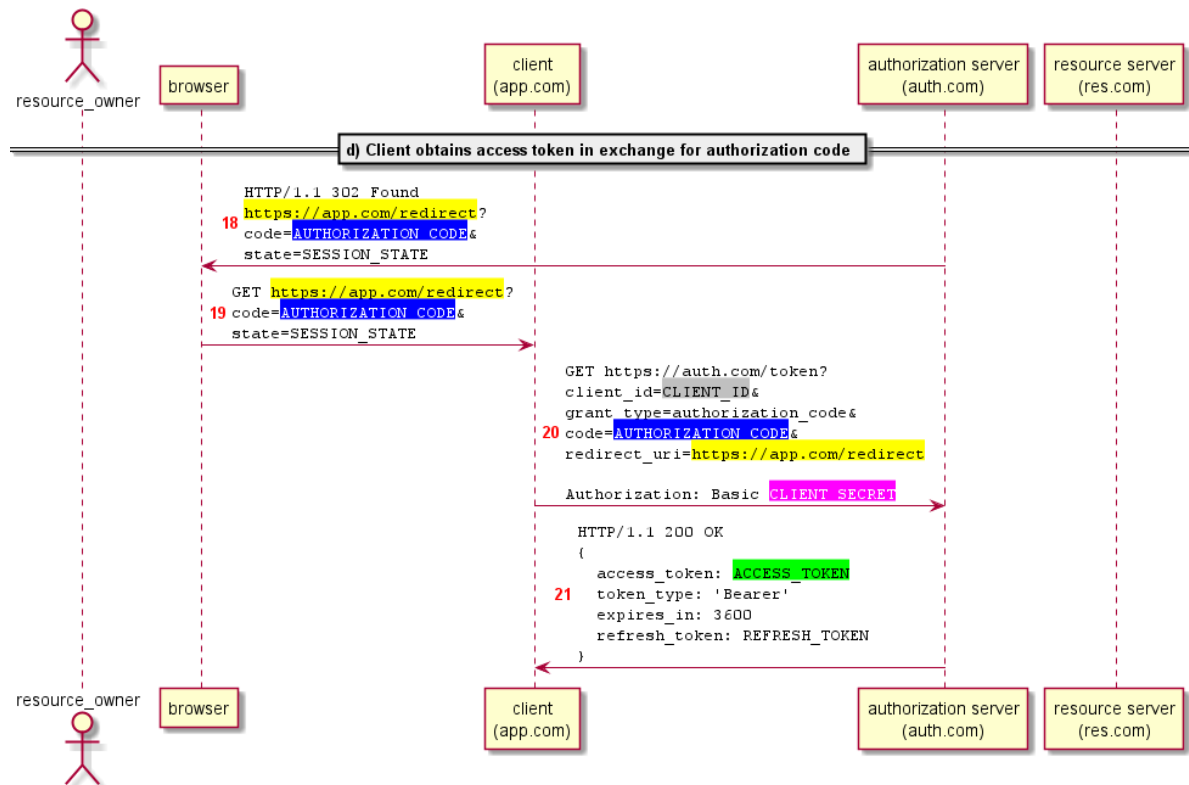


Figure 146: OAuth ACG flow - d) Client obtains access token in exchange for authorization code

- Step 18 and 19: Here, the authorization server actually uses the client's redirection URL and appends and code = **AUTHORIZATION_CODE** as well. The authorization server cannot deliver an access token here, because of the detour via the browser.
- Step 20: The client uses the token endpoint `https://auth.com/token` to convert the authorization code to an access token. The client sends the following parameters:
 - client_id = **CLIENT_ID** identifies the client
 - client_secret = **CLIENT_SECRET** authenticates the client and was agreed upon during the client registration (cf. Section 30.4.3).
 - code = **AUTHORIZATION_CODE** is the authorization code
 - grant_type = `authorization_code` tells the authorization server how to interpret parameter code.
 - redirect_uri = `https://app.com/redirect` again tells the authorization server about the redirect URL. This redirect URL must be identical to that of Step 4.
- Step 21: The authorization server shares a secret **ACCESS_TOKEN** and optionally a secret **REFRESH_TOKEN** with the client together with an expiration period (`expires_in`) in seconds and a `token_type`, which tells the client how to include the access token in requests to the resource server.

EXAMPLE 149: ACCESS TOKEN

The actual response of Step 21 might look like this:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache
```

```
{
  "access_token": "2YotnFZFEjr1zCsicMWpAA",
  "token_type": "Bearer",
  "expires_in": 3600,
  "refresh_token": "tGzv3J0kF0XG5Qx2TlKWIA",
}
```

- Requests from the client to the resource server must include an HTTP header

EXAMPLE 150: ACCESS TOKEN IN REQUESTS

Authorization: Bearer 2YotnFZFEjr1zCsicMWpAA

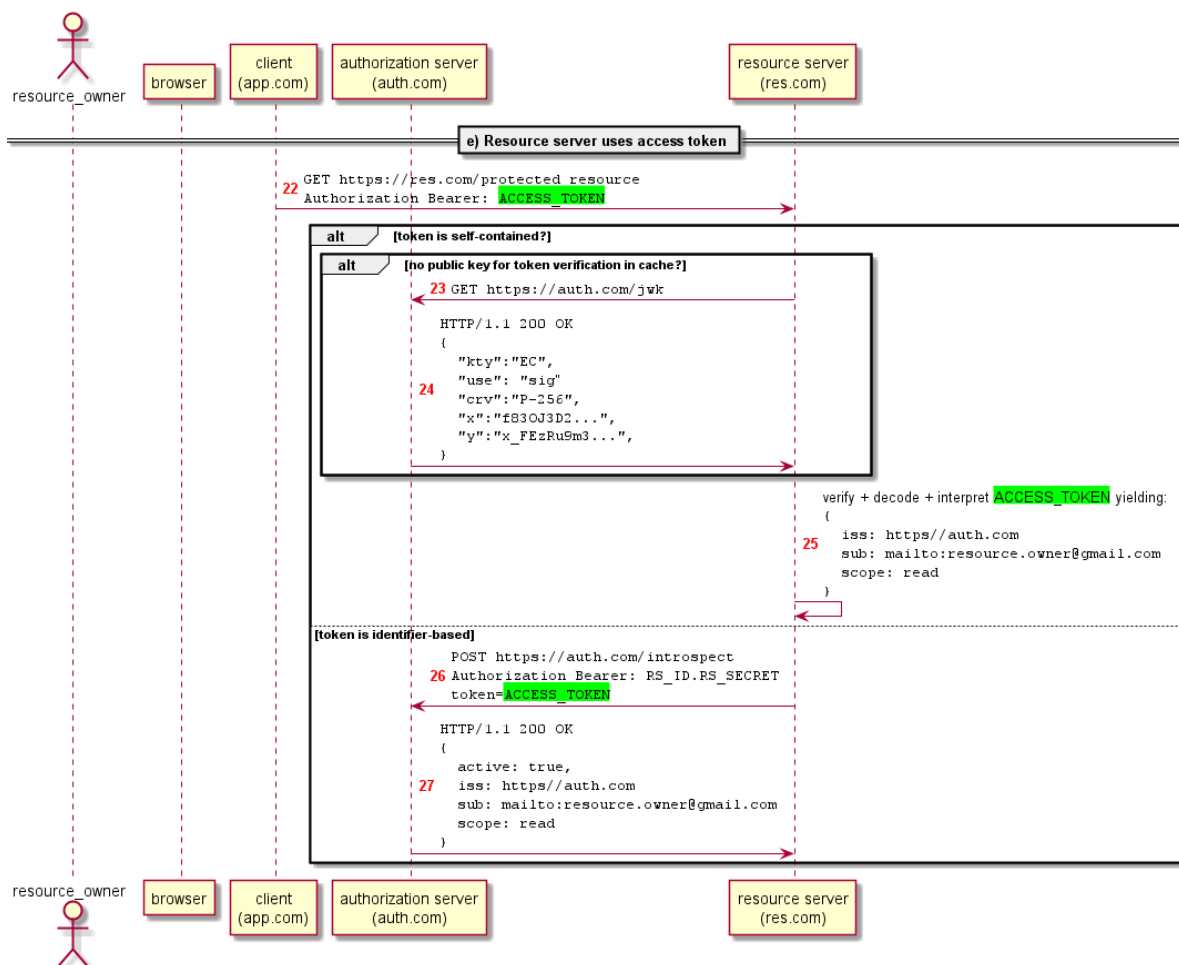
e) Resource server uses access token

Figure 147: OAuth2 ACP flow - e) Resource server uses access token

- Step 22: Now the client can use the **ACCESS_TOKEN** as the value in an Authorization Bearer HTTP header to authorize the access to `https://res.com/protected-resource`.
- Steps 23 to 27: These steps are again out of scope of the OAuth2 specification (RFC 6749 [158]). The question remains, however, how the resource server makes sense of the access token and particularly how it gets the granted scopes. There are two ways:
 - Step 23 to 25: **Self-contained access token**: An access token might be a self-contained **JSON Web Token** (JWT, cf. Section 30.5) thus carry all necessary information in itself. Particularly, the scope can be part of the JWT Claim Set (cf. Section 30.5.2) within such an access token. Such a JWT access token must be verified with an ephemeral public key from the authorization server. The resource server can obtain (and cache for a while) this ephemeral public key for the access token verification from the authorization server's **JWK endpoint**, `https://auth.com/jwk` in this case. This endpoint responds a public key (or even a set of public keys) in the **JSON Web Key (JWK)** format, as defined by RFC 7517 [163].
 - Step 26 and 27: **Opaque access token**: An opaque access token can merely serve as an unguessable identifier. The resource server uses this access token in a call to the introspection endpoint of the authorization server according to RFC 7662 [159]. The resource server identifies and authenticates itself with another client id (`RS_ID`) and secret (`RS_SECRET`). The response would deliver some kind of JSON Web Token claim set (cf. Section 30.5.2).

In both cases, the resource server could derive that client request with the particular **ACCESS_TOKEN** is authorized to read-access (scope: read) all resources that are owned by a subject (sub), i.e. user, with the email address `resource.owner@gmail.com`. The value of sub might also be a random but constant identifier for the resource owner that is initially assigned by the authorization server. Then, the client and resource server are still able to (re-)identify the resource owner and his or her owned resources but do not learn anything about the resource owner's identity (like the email address).

The downside of using the introspection endpoint is the additional roundtrip between the resource server and the authorization server for every new access token. Moreover, this approach requires that the resource server may do outbound HTTP request to the authorization server. The upside is that an authorization server can revoke an access token before it expires just by setting `active: false` in the response.

Please note that in the case above, the resource server not just authorizes a client's request but effectively also identifies and authenticates the resource owner during authorization. This is not uncommon because a scope: read alone tells nothing about what resources are supposed to be read-accessible. If the resource server associates all resources of the resource owner with the subject (sub: `mailto:resource.owner@gmail.com`), then it would just make these resources readable.

f) Client renders protected resources

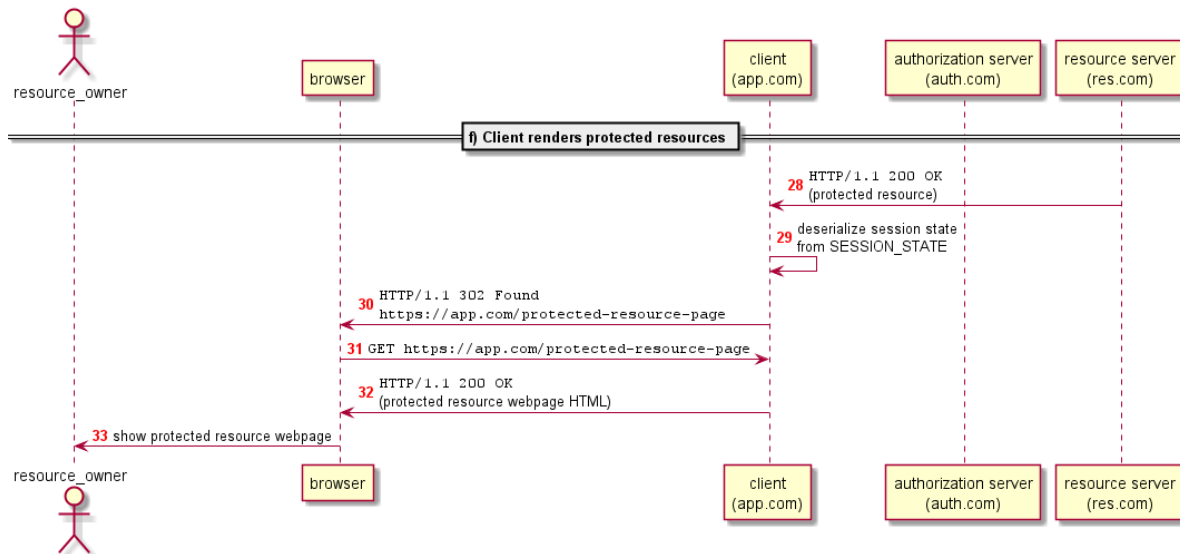


Figure 148: OAuth2 ACG flow – f) Client renders protected resources

- Step 28ff: The following steps are out of any OAuth-related RFC. Typically, a client would deserialize SESSION_STATE which contains the former client's navigational state `https://app.com/protected-resource-page` and redirect the browser to it. Please note that the last inbound HTTP request to the client was at Step 19. So, even if the client actually is a scaled tier of nodes, Steps 20 to 30 are supposed to be executed by the very same node such that Step 29 can access the SESSION_STATE relayed in Step 19.

30.4.6 Refreshing an Access Token

As described in Section 30.4.5d), an authorization server can issue a refresh token along with the access token. If a client's access token expires, the client can use the refresh token in order to get a new access token without replaying the whole Authorization Code Grant flow and bothering the resource owner.

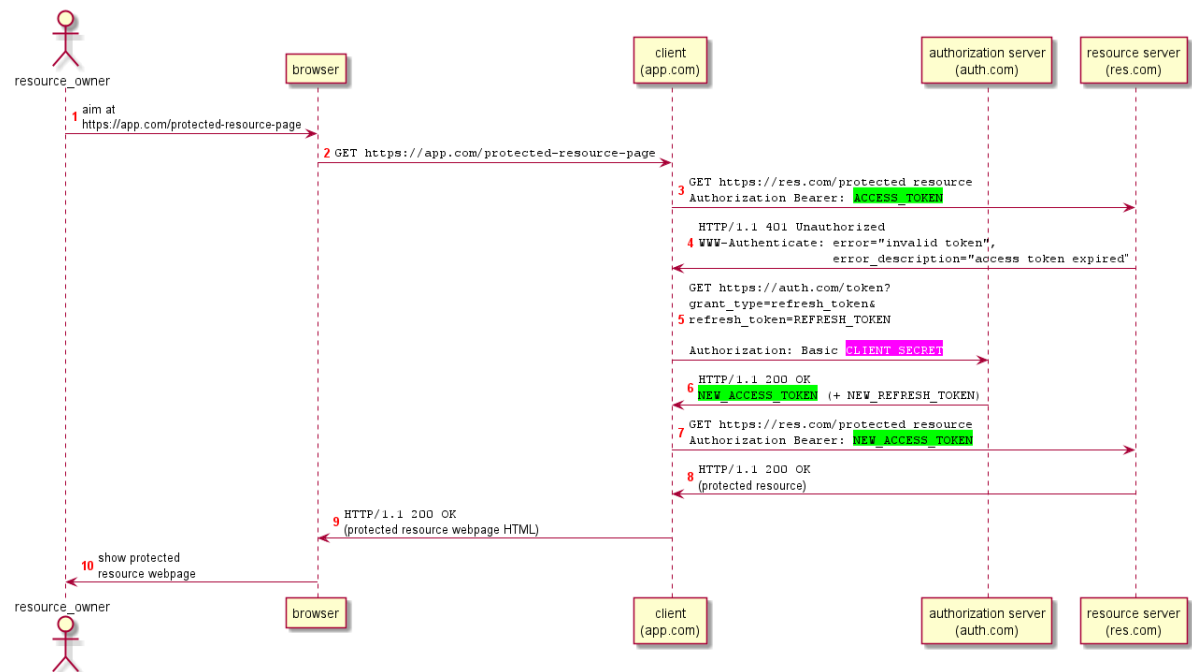


Figure 149: OAuth2 Refreshing an access token

Figure 149 exemplifies a refresh flow which could happen, e.g., any time after the Authorization Code Grant flow. Please note that a new refresh token in the response of Step 6 is issued optionally.

30.5 JSON Web Token (JWT)

Security tokens can represent just unguessable but meaningless values (strings), which are just used for equality checks or as keys in a lookup for some meaningful data. Such tokens are called **opaque tokens**. Alternatively, as discussed in Section 30.4.5e), security tokens can be self-contained, i.e., carry meaningful data in themselves.

JSON Web Token (JWT) as defined in RFC 7519 [153] is a standard about how to format, encode and sign such self-contained security token. Such tokens contain a set of claims. A **claim** is a property that is asserted about a subject. Claims are technically represented as key-value pairs, where **claim name** is the key **claim value** the value.

A JWT token is special kind of a **JSON Web Signature (JWS)** [160] and as such composed of

1. a **JOSE Header** [160, p. 9], which describes cryptographic algorithms and their parameters that are applied to a **JWT Payload**, which is in case of JWT
2. a **JWT Claims Set** in JavaScript Object Notation (JSON), i.e., a set of claims, and
3. a **JWS Signature**, which is basically a signature (cf. Section 30.2.4b) derived from other two plus the private key of the issue of the respective JWT

30.5.1 JOSE Header

In decoded form, a JOSE Header typically looks like this:

EXAMPLE 151: DECODED JOSE HEADER

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Here, `alg` is key for a hashing algorithm and `typ` denotes the media type (cf. Section 27.1.1) of the complete JWS, which is `JWT`. (As value length is of the essence, the value lacks the default prefix `application/`, so actually the media type is `application/JWT`.)

Since JWTs are embedded in HTTP headers, a plain JSON representation (with its special characters and line breaks that would confuse a HTTP parser) is not suitable. Instead, the JOSE Header is transformed into a URL-compatible Base64 encoding, which looks like this for the example above.

EXAMPLE 152: ENCODED JOSE HEADER

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

30.5.2 JWT Claim Set

A JWT Claim Set looks with the three claims like in Section 30.4.5e) looks like

EXAMPLE 153: DECODED JWT CLAIM SET

```
{
  "iss": "https://auth.com",
  "sub": "mailto:resource.owner@gmail.com",
  "scope": "read"
}
```

and its Base64 encoding would be

EXAMPLE 154: ENCODED JWT CLAIM SET

```
eyJpc3MiOiJodHRwczovL2F1dGguY29tIiwic3ViIjoibWFpbHRvOnJlc291cmNlLn93bmVyQGdtYWlsLmNvbSIsInNjb3BlIjoicmVhZCJ9
```

30.5.3 JWS Signature

Even in their Base64 encoding, JWTs would be easy to tamper. Therefore, their integrity can be checked by an JWS Signature. In the example from above, the data to sign are the Base64 encodings from above. They are delimited by a dot, i.e.

EXAMPLE 155: ENCODED JWT JOSE HEADER AND CLAIM SET

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJodHRwczovL2F1dGguY29tIiwic3ViIjoibWFpbHRvOnJlc291cmNlLn93bmVyQGdtYWlsLmNvbSIsInNjb3BlIjoicmVhZCJ9
```

This encoded, concatenated data together with a private key of value, say, `somePrivateKey` is taken as input a HS256 signing algorithm.

EXAMPLE 156: JWT SIGNATURE COMPUTATION

```
HS256(eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJodHRwczovL2F1dGguY29tIiwic3ViIjoibWFpbHRvOnJlc291cmNlLn93bmVyQGdtYWlsLmNvbSIsInNjb3BlIjoicmVhZCJ9,
somePrivateKey) == Qbo5qTxTxR518I_5dgB1E9tRMfKEoboP6iipUhSYccw
```

whose result is also concatenated after the other two

EXAMPLE 157: COMPLETE ENCODED JSON WEB TOKEN (JWT)

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJodHRwczovL2F1dGguY29tIiwic3ViIjoibWFpbHRvOnJlc291cmNlLn93bmVyQGdtYWlsLmNvbSIsInNjb3BlIjoicmVhZCJ9.Qbo5qTxTxR518I_5dgB1E9tRMfKEoboP6iipUhSYccw
```

which constitutes a complete JWT.

The website <https://jwt.io> provides a handy debugger that encodes and decodes JWTs.

30.6 OpenID Connect

The OAuth2 specification leaves open not just how a resource owner authenticates with an authorization server but also how an end-user (resource owner) in general authenticates with a client in the first place. Authentication of an end-user for a client is at best a by-product of OAuth2 and just works if either the access token is a JWT, or the client is authorized to access the introspection endpoint. If not, the client is left completely unaware of the end-user's identity. Moreover, the JWT Claims Set in the JWT or the introspection endpoint response might just include minimal end-user-related properties, in the worst case just an opaque end-user ID in the sub claim.

OpenID Connect (OIDC) [164] specifies a way how OAuth2 can be used to identify and authenticate an end-user for a client. As with OAuth2, the end-user only has to authenticate with an authorization server but not with potentially numerous clients. The end-user's credentials are just transferred to that single authorization server and verified there, which makes them less likely to be leaked and compromised. The end-user's authentication with the authorization server normally stays valid for the lifetime of the session between the end-user's browser and the authorization server. During that period, the end-user can be authenticated with arbitrarily many clients without having to present his or her credentials again. In this regard, OIDC provides a means for implementing **single sign-on (SSO)**.

In OIDC, the authorization server also acts as a resource server for protected resources of the kind **user-info**. User information are data like name, email address, phone number, an identifier of the end-user, which is unique within the authorization server, and the like.

30.6.1 User-info Endpoint

In OIDC, an authorization server therefore provides another endpoint called **user-info endpoint**, exemplified as `https://auth.com/user-info` in the following. Regarding this endpoint, the authorization server acts like a resource server.

A client can obtain an access token using the Authorization Code Grant flow in order to get an access token by means of which the client can access the user-info endpoint. With the user info at hand, a client then is able to identify and authenticate an end-user.

30.6.2 ID Token

Alternatively, or additionally, an authorization server can include user info a self-contained **ID token**. The authorization server includes such ID tokens in the response from the token endpoint (along with the access token and refresh token).

Please note that both methods are not redundant to each other. Their up- and downsides resemble the alternative ways in which a resource server may treat an access token (cf. Section 30.4.5e): The ID token spares the client another request-response cycle. However, the user info in the ID token might get outdated while responses from the user info endpoint are always up to date. Even more, obtaining an ID token requires the presence, i.e., login, of the end-user. In contrast, if the client is already in possession of an access token (and a refresh token), the client can also access the user info endpoint "in the background", i.e., without the login of an end-user.

ID tokens are JWT tokens (cf. Sections 30.5). The JWT Claim Set of an ID token might look like this:

EXAMPLE 158: JWT CLAIM SET

```

{
  "iss": "https://auth.com",
  "sub": "mailto:resource.owner@gmail.com",
  "aud": "aKdf6s3idf",
  "nonce": "n-0S6_WzA2Mj",
  "auth_time": 1311280969,
  "iat":      1311280970,
  "exp":      1311281970,
  "email": "resource.owner@gmail.com"
  "name": "Resource W. Owner III"
}

```

Here,

- iss denotes the issuer, which is the authorization server,
- sub is the subject which is unique and immutable within the issuer and may be used by the client in order to identify the end-user. (Please note that it is not necessarily smart to use an email address here.)
- aud is the audience, which must be exactly the **CLIENT_ID** that has been set during client registration (cf. 30.4.3)
- nonce a random value associated with the client session for replay attack mitigation,
- auth_time is the Unix time when the authorization server successfully authenticated the end-user,
- iat is the "issued at"-time, i.e., the time the ID token was put in the response to the client, and
- exp is the expiration time for the ID token.

Furthermore, email and name are additional and optional claims that provide extra information about the end-user. [164] standardizes the names and semantics of such predefined claims.

30.6.3 OpenID Connect Authorization Code Grant Flow

Since OpenID connect is based on OAuth2, it can be used with the different flows that OAuth2 defines. In Figure 150 provides an overview how OpenID Connect is used via an OAuth2 Authorization Code Grant Flow. Again, just the crucial messages are depicted. Please note the absence of a dedicated resource server. For OpenID Connect, the authorization server plays the role of a resource server with regard to the user-info endpoint (Step 11.1).

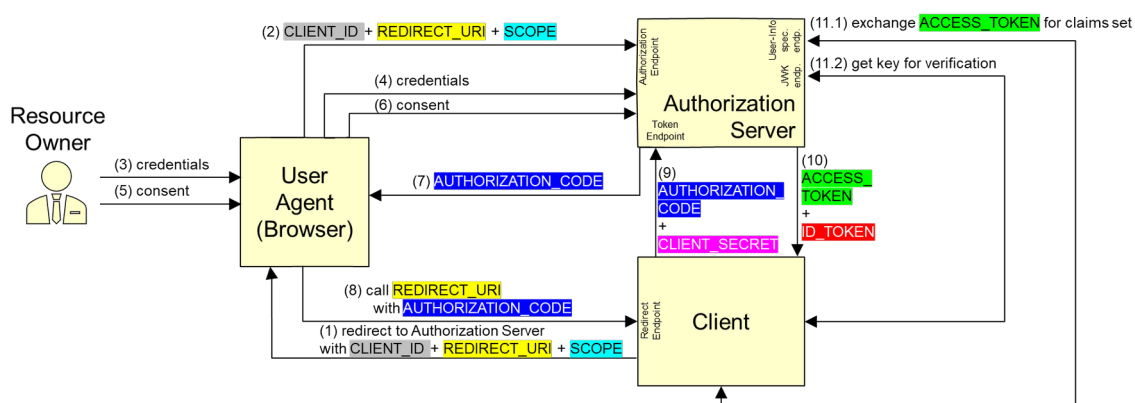


Figure 150: OIDC ACG flow - overview

Figure 151 again gives some more details. Some facts are noteworthy: For brevity, the figure lacks the initial steps 1 to 17. These are the same like those in Figure 143 to Figure 145 with one exception: The **SCOPE** has to mandatorily include `openid` and those claims that should be additionally included in the ID token (`email` and `name`) or should be accessible by the client if it calls the user-info endpoint.

The client authenticates the end-user in step 22 to 26. It can rely on the authenticity of the ID token (Steps 22 to 24) and on the uniqueness and immutability of the `sub` claim. Alternatively, the client can use the access token in order to get the same claims from the user-info endpoint plus an additional active indicator (Steps 25 and 26).

In practice—as state before—pure OAuth2 can yield end-user (resource owner) authentication as a by-product. It is up to the authorization server what to additionally include in the JWT Claims Set in a JWT access token or an introspection endpoint response

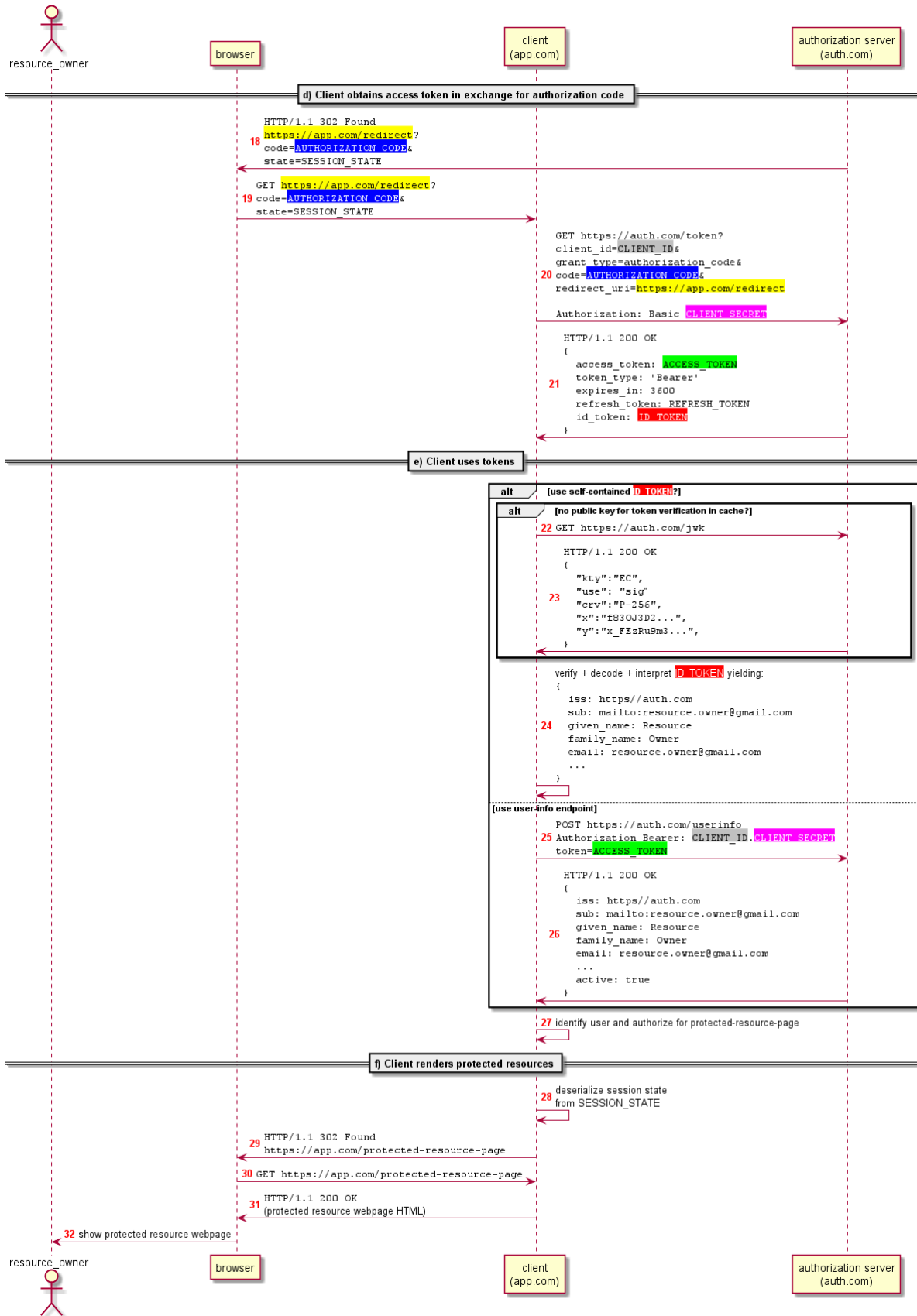


Figure 151: OIDC ACG flow – cut-out with details.

V. Databases

31 Persistence in Databases

The most precious parts of a large and cloud-based software systems are data that are **persisted** in some kind of **storage**. Storages keep the overall application state and its history. This is why some kind of storage belongs to all but trivial web applications.

Unfortunately, storages are particularly hard to get to be consistent and performant and available at the same time for arbitrary kind of data. This is why storage comes in numerous forms all of which serve different use cases and can be classified according to many attributes.

31.1 Databases and File Systems

The main dichotomy in the area of persistence is that between of databases and file systems. **File systems** are an abstraction provided by operating system kernels: An operating system kernel translates system calls (open, read, write etc.) from user-space processes to a local filesystem into raw block accesses to a disk, which can be built-in or mounted via, e.g., a storage area network (SAN) (cf. Section 31.3).

Databases are built on top of file systems. Technically, a database is a set of large files with a complex internal structure managed by a set of user-space processes called **database server**. Such database servers logically expose the data to clients following a **data model** like a relational data model.

Both kinds of persistence are accessible via networks. In the case of databases, database server processes typically expose one or more network ports that receive requests from remote clients. In the case of file system, additional user-space processes that implement a **distributed file system protocol** like the Network File System (NFS) [165] can expose a local file system in a network.

Cloud storage, or **object storage**, is a new kind of storage that is exclusively offered by cloud providers, whereas disks and databases are commodity hard- and software that need not to be operated in a cloud at all. Cloud storage is a compromise of a file system on a disk and a database. It offers a simple **data model** that resembles that of a file system. Data are organized in **objects** similar to files that have unique names similar to paths + filenames in a filesystem and live in **buckets** similar to a single file system.

Buckets cannot be directly mounted into the filesystem of a machine. Instead, cloud storages expose web APIs by which objects can be created, read, updated and deleted (CRUD operations) in a bucket.

Each bucket gets a storage class on creation. The **cloud storage class** mainly determines

- the guaranteed availability of the bucket (ranging from 99,9% to more than 99,99%),
- if data is replicated multi-regionally,
- the cost for storing data for a certain time, which is in the order of magnitude of 0.01\$ per gigabyte and month, and
- the cost for transferring the data from and to the bucket.

Storage classes that provide high availability, multi-regional replication and low transfer costs are suitable for data, which are frequently accessed. Buckets of that storage class might back a CDN. A storage class with low storing costs is suitable for uses cases like backups or archives, since these data are not accessed frequently.

31.2 Database Server

A **database server** cares for the proper access to databases. It is a set of programs running in a set of user-space processes with different responsibilities that could be conceptually assigned to layers as depicted in Figure 152. The **transport layer** cares for communication between database servers (nodes) within the same database server tier and for communicating with database clients like application servers. Such clients send queries in, e.g., **Structured Query Language (SQL)** to a database server. These queries are parsed by some **query parser** yielding one or more possible data access sequences (**execution plans**). It is up to the **query optimizer** to find the optimal execution plan. An execution plan is executed by an **execution engine** either on a **storage engine** in the same host (**local**) or another host (**remote**). The storage engine's **transaction manager**, **lock manager** and **recovery manager** ensure ACID properties to some degree (cf. Section 9). **Access methods** organizes data that is stored on disks while a **buffer manager** manages memory pages that contain cached database data.

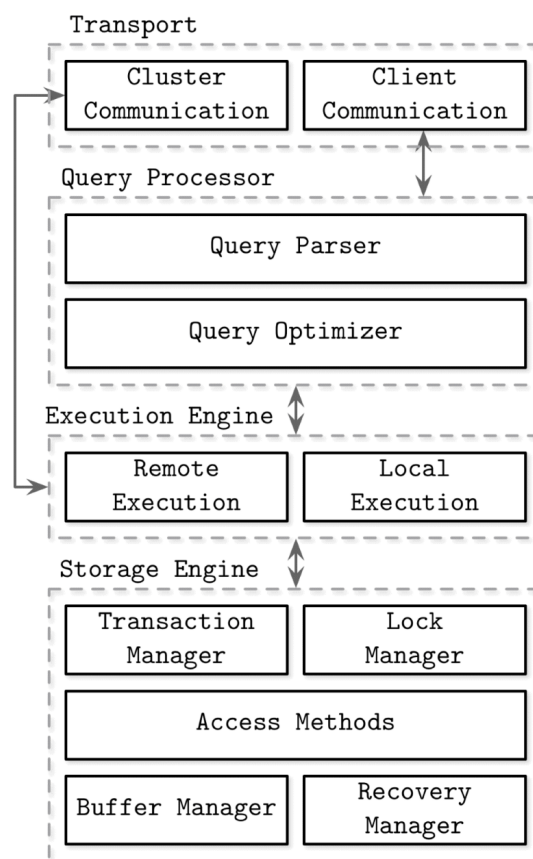


Figure 152: Architecture of database servers

Like application servers, database servers can be scaled horizontally. Instead of talking about n database servers in a database server tier, we speak about a **database management system (DBMS)** in the following. (The term **database instance** is also common and used synonymously to DBMS in the following, outscaled or not. In other contexts, database instance might also mean a single database server in an outscaled tier.)

Sometimes, it makes sense to use different types of databases in a single system. For example, **master data** which is not susceptible to changes but need to be consistent is better accommodated in a relational database with ACID properties. Conversely, highly **transactional data** with non-strict consistency constraints should be stored in a NoSQL database with BASE properties.

EXAMPLE 159: MASTER AND TRANSACTIONAL DATA IN ReWo-RATE

In ReWo-Rate, master data includes user properties like login names, passwords, contact data etc. but also catalogues of products that are rated. Transactional data mainly consists of ratings but also logs in the sense of Section 10.1.1c) which can be stored in a database as well.

31.3 Database Storage

In production environments, the actual database data is not stored in the local file system of a database server. Instead, the **storage**⁶⁹ that contains the actual database data is realized by **hard drive disks (HDD)** or **solid-state disks (SDD)** (or just **disks**) which are attached to a database server using low-latency-high-throughput networks of the following types:

- A **network attached storage (NAS)** is a kind of server on its own and exposes a file system which can be mounted as-is by other servers. A NAS uses common ethernet LAN networks and thus is comparably slow.
- A **storage area network (SAN)** connects a disk to a machine such that it can almost be used like an internal disk, i.e., it can be formatted with a certain file system and provides a high-performance communication between the machine and the disk.

⁶⁹ Storage is a generic term for resources that allow for persisting data. Besides HDDs and SSDs, storage could also be realized by magnetic tapes, which are cheap but have a huge latency in random accesses due to long seek times, SD cards, USB sticks, DVD RW disks etc. However, these devices are not suitable for accommodating database files.

- An **iSCSI NAS** is somewhat in-between: It uses normal ethernet LAN but represents data on disks like SAN, i.e., in a raw and unformatted way.

	Network Attached Storage (NAS)	iSCSI Network Attached Storage (iSCSI NAS)	Storage Area Network (SAN)
Data representation	file system	raw block data	raw block data
Network interface	ethernet LAN	ethernet LAN	fiber channel
Typical machines	workstations	servers	servers

Table 50: Technologies for attaching storage to machines

32 Database Paradigms

A key differentiator between different database products are their respective

- data model (e.g. relational or document-oriented)
- indexing method (e.g. B-Tree or inverted index)
- query access language (e.g. SQL, Cypher, JSON selectors, key-based lookup)
- supported consistency models (e.g. serializable, read committed)
- supported scaling models (sharding, replication etc.)

These criteria are not independent: The data model naturally shapes the query language. One could even say that queries make the data observable in the first place and thus that the data model is manifested in the query language. Moreover, data models correlate with supported consistency and scaling models via their typical use cases.

This chapter focuses on data models and their query languages. A **data model** is similar to a programming paradigm: Eventually, data are stored bitwise on disk like every (compiled) high level language source code ends as op codes. However, a data model or programming paradigm abstract from technical details and provides a human-friendly way to reason about data or programs, respectively. A data model prescribes, how data is organized into hierarchical and interrelated **data records** that carry application data (e.g., “rows”) and other **database objects** that help manage the data (e.g., “sequences” or “users”). Such data model concepts particularly surface in the use of a **query language** used by clients to communicate with a database management system.

32.1 Data Model Criteria

In the following we provide some criteria in order to classify different data models.

32.1.1 Schematic or Non-schematic Data

Data can be rather schematic or non-schematic. If the different data types of a domain together with their respective attributes and relationships with other data types can be completely foreseen in advance, a data model is rather **schematic**. Conversely, if data records differ so much in their attributes and relationships that they cannot be reasonably assigned to data types or the resulting schema would have to incorporate too many variants, a data model is considered to be **rather non-schematic**. In the latter case, one often leaves the data schema **implicit**. All reading and writing clients must tacitly agree on a schema to some degree⁷⁰, i.e., which attributes and relation can be expected in and between which data records. In the former case, a schema can be **explicit** and enforced by the DBMS itself, i.e., schema violating writes **fail fast** since they are already rejected by the DMBS.

32.1.2 Relationality

Data records can have different relationships with other data records. These relations can be classified by cardinalities, like one-to-one, one-to-many, many-to-one, and many-to-many.

EXAMPLE 160: RELATIONS BETWEEN ReWo-RATE ENTITIES

In ReWo-Rate, one would have

⁷⁰ Of course, there are also generic clients for database server like SQLPlus, Toad and the like that can cope with any kind of schema. However, they leave it up to the ultimate client, that is the user, to make sense of the generically presented data.

one-to-one from a transaction data record to a rating data record that gives a rating to that particular transaction,

one-to-many from a product data record to rating data records for that product,

many-to-one from user data records to a country specifics data record, which contains country specific configurations for users of a certain country, and

many-to-many from user data records to rating data records that have been read by a user.

One-to-one and one-to-many relationships induces isolated trees a.k.a. **aggregates** of data records (cf.) whereas many-to-one and many-to-many induce (hyper) graphs.

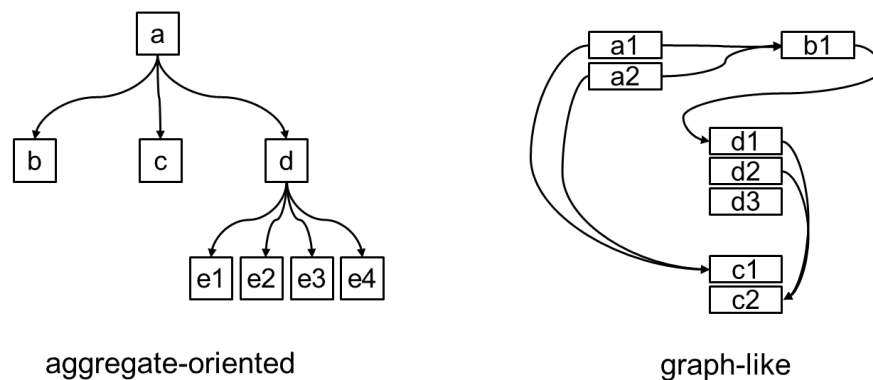


Figure 153: Aggregate-oriented vs. graph-like data model.⁷¹

Graph-like data models directly support each type of cardinality and allow for dereferencing such relationships in a single query. The (oftentimes explicit) data schema in a graph-like data model can be designed without respect to queries. It is up to clients to formulate a certain single query in a way that the DBMS can combined exactly the requested data in the response.

Aggregate-oriented data models directly support just one-to-one and one-to-many relationships. There are basically two ways to deal with that:

1. A client (developer) has to gather and combine data on its own in multiple queries before proceeding computation.
2. The (implicit) data schema is designed particularly with hindsight to the queries. Denormalization, i.e., data duplication along with additional storage consumption must be accepted and potential update anomalies that lead to data inconsistencies must be treated with additional care.

In a way, the trade-offs between an aggregate-oriented and a graph-like data model resemble the trade-offs between different web API styles (cf. Section 27.1.9): A relational database like a GraphQL web API can be queried by clients in a comparably flexible way to avoid problems like over- and under-fetching. Among the downsides is of course that these flexible and build-time-unknown queries might require a longer processing time since the database and the GraphQL server, respectively, cannot be optimized at build time for these queries.

Databases with aggregate-oriented data models are particularly popular in the realm of CRUD-like web applications. The UI of such web applications is often comprised of a number of overview pages where the user can navigate to detail pages of certain types. With an aggregate-oriented

⁷¹ Same letters indicate same sub-schema. In the graph-like data model the nodes with same sub-schema have been intentionally stacked onto each other in order to make them look like table rows.

data model, a developer often needs just to fetch a single document data record (cf. Section 32.2.3) instead of data records from multiple tables when he or she implements the rendering of a certain web page.

32.1.3 Locality

Locality refers to coherent data records, which are likely to be accessed together by a client:

- **Technical locality** contributes to shorter response times between clients and a DBMS and is given if coherent data records are stored “close” to each other, i.e., in the same disk block or adjacent disk blocks or at least on the same disk.
- **Conceptual locality** contributes to simplicity (cf. Section 10.1.2) and is given if coherent data records are contained in a single conceptual element of the respective data model, like a row, table, column, document etc.

In general, both play together: Data in the same, e.g., document is likely to be stored close to each other on the same disk. However, there are exemptions like a Relational DBMS either stores data row- or column-wise or splits tables into different shards placed on different (remote) disks.

32.1.4 Query Languages

In a **query language**, a client developer can formulate read and write accesses to those data the client needs. Different data models come along with different query language that fit to specific concepts of the respective data model. Some query languages are DBMS product independent standards while others only apply to a specific DBMS product.

32.1.5 Impedance Match

Certain data models have a good fit with certain programming paradigms, like document-oriented data models with object-oriented programming languages, while others have a mismatch like object-oriented programming languages and relational data models [24, pp. 29–30]. A mismatch decreases the simplicity of the overall system due to additional subtle mappings.

32.2 Data Models

With the criteria from above one can characterize different data models as shown in Table 51. Note that the listed concepts in the concepts column roughly go from large to small. For example, in a document-oriented database, the “instance” denotes the whole database which may contain several logically separated “database”. The most fine-grained concept in a document-oriented database is a key-value pair.

On the upper conceptual layers with concepts like “instance”, “database”, “schema”, data models and their implementation in different database products resemble each other. These upper layer constructs are rather meant to logically group and separate smaller database objects on the conceptually lower, fine-grained layers, like “table”, “row”, “sequence”, “collection”, “key-value pair”.

Please note, that in architectural discussions, a “separate database” might refer to a technically separate instance, e.g., a whole DBMS on its own. Alternatively, “separate database” sometimes also means a “database” or “schema” that (by coincidence) technically co-exists with other databases and schemas on the same instance. (Likewise, a node that contains a running program may run on the same (virtual) machine like other nodes but is isolated from them (cf. Section 25.1.1). In either way, the crucial point in architectural discussions about “separated databases” is

that data are logically separated from each and, e.g., a single transaction may not access data in two different databases, or one database has foreign keys that refer to the other database.

Data Model	Typical Use Cases	Query Language(s)	Schema	Relation-ality	Concepts	2020 Ranking Leader DBMS
Key-Value	Caches, Session Stores (e.g., shopping carts), User Profiles	mostly DBMS dependent like "Redis Commands"	implicit	aggregates with up to 2 layers. ⁷²	cluster, bucket, key-value	Redis
Wide-Column	OLTP, Event-Logging, Content Management	CQL	implicit or explicit (DBMS dependent)	aggregates with up to 3 layers. ⁷³	cluster, key space, table, row, column	Apache Cassandra
Document	OLTP, Event logging, content management, web analytics, unknown, dynamic domains (e-commerce)	mostly DBMS dependent like "Mongo Shell"	implicit	aggregates of arbitrary depth	Instance, database, collection, document, key-value	Mongo DB
Relational with row-wise storage	OLTP, well understood, static domains	SQL (variants)	explicit	graphs	instance, database, table, column, row.	Oracle DB MySQL SQL Server PostgreSQL
Relational with columnar storage	OLAP	SQL (variants)	explicit	graphs	Instance, database, table, column, row	Snowflake, Google BigQuery, Amazon RedShift
Graph	social networks, routing, recommendation engines, master data (org charts), fraud detection, graph-based search, identity management	Cypher SPARKQL	implicit or explicit (schema optional)	graphs	relationshi p, node, property	Neo4j Cosmos DB
File System	Storing arbitrary data	None. OS provides syscalls with paths as parameters	implicit	aggregates	File system, directory, file, links	-

Table 51: Data models

32.2.1 Key-value oriented

Key-value oriented databases provide the simplest data model. A key-value database is basically just a big persistent rather flat hash map with values of different types (strings, integers, sets, lists, hash maps).

⁷² If values on the first level refer to buckets, which are hash maps on their own.

⁷³ MongoDB knows about super column families, column families and rows.

EXAMPLE 161: REDIS USAGE

In Redis one could use its own query language like this

```
SET users:0001:name "René Würzberger"
HSET users:0001:telephone private "+49 123 321212"
HSET users:0001:telephone business "+49 221-8275-4324"
RPUSH "users:0001:emails" "rene.woerzberger@th-koeln.de"
RPUSH "users:0001:emails" "foo@bar.de"
in order to fill a key-value database that look like this in a JSON representation
```

```
{
  "users:0001:name" : "René Würzberger",
  "users:0001:telephone" : {
    "private": "+49 123 321212",
    "business": "+49 221-8275-4324"
  }
  "users:0001:emails" : ["rene.woerzberger@th-koeln.de", "foo@bar.de"],
}
```

Note that the ":" in the keys make them look like a list of path segments in a tree. However, this is pure naming convention which Redis is agnostic of.

A convenient and cost-free way to try out the Redis queries is: <https://try.redis.io>

The file system can also be seen as a key-value database provided by the operating system itself. Here, the keys are the file paths, and the values are the file contents.

32.2.2 Wide-Column/ Column-Family

Wide-column (a.k.a. **column-family**) data models are somewhat in between the conceptual world of relational and aggregate-oriented data models. On the one hand, they use the concepts relational data models like tables (a.k.a. column families), columns and rows and SQL-like query expressions. On the other hand, creating a plethora of different (sparsely filled) columns is normal usage in wide-column data models like it is common to have many different keys in a document-oriented data model.

In comparison with relational data models, wide-column data models are highly denormalized by design not least for the reason that foreign keys and join operations are not supported. This is what leaves them aggregate-oriented after all: A table contains rows which is said to contain columns in literature although cells would be a better word here.

One could say that wide-column data models sacrifice relationality and queries that include joins of tables that might be non-local to each other for better performance. Even more, column-wide stores promote a "query-based design of tables", i.e., tables are designed to suit queries of clients.

EXAMPLE 162: CQL USAGE

The following CQL statements in Apache Cassandra

```
CREATE KEYSPACE ReWo-Rateks WITH REPLICATION = {'class': 'SimpleStrategy',
'replication_factor' : 1};
USE ReWo-Rateks;
CREATE TABLE user (id TEXT PRIMARY KEY, name TEXT, gender TEXT);
INSERT INTO user (id, name, gender) VALUES ('user0001', 'Woerzberger', 'male');
ALTER TABLE user ADD (email text);
INSERT INTO user (id, name, email) VALUES ('user0002', 'Musterfrau',
'b@musterfrau.de');
INSERT INTO user (id, email) VALUES ('user0003', 'm@mustermann.de');
```

would create and fill a table that would be serialized in JSON like this:

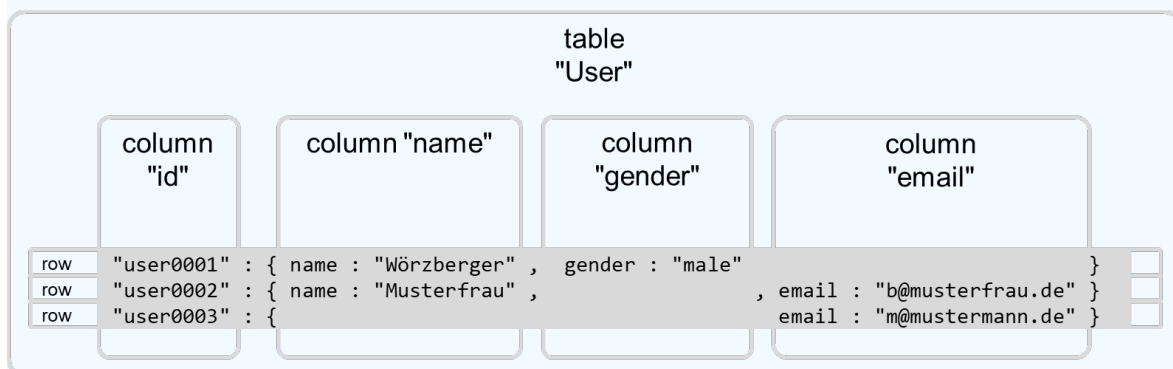


Figure 154: Wide-column data (example)

Note that the email column has been added in-between. SELECTs work like in SQL yet without the chance for JOINS:

```
SELECT email FROM user;
```

yields

```
email
-----
null
m@mustermann.de
b@musterfrau.de
```

A convenient and cost-free way to try out Apache Cassandra and CQL queries is:

<https://astra.datastax.com/>

32.2.3 Document-oriented

A document-oriented data model can be thought of as a set of arbitrary JSON documents. A document is a set of key-values (a.k.a. dictionaries a.k.a. hash maps) where values again might be sets of key-values etc.

Documents themselves are aggregated by collections. Oftentimes, documents in a collection are instances of a certain data type and therefore similar in their schema. These are, however, effects of implicit schemas and are rather convention and best practice but not enforced by document-oriented data models.

EXAMPLE 163: MONGODB USAGE

In MongoDB, one mainly creates, reads, updates, or deletes a single document per query, set aside bulk operations that apply an update on a set of documents that match a certain filter.

Start a MongoDB-Atlas Database

```
docker run --rm --name atlas-local \  
-p 27017:27017 mongodb/mongodb-atlas-local:8.0.14
```

and connect to it via MongoSh

```
docker exec -it atlas-local mongosh
```

The following query statement in Mongo Shell

```
db.credentials.insertOne({"username" : "abc", "pwhash" : "ek3nd37n"});  
db.credentials.insertOne({"username" : "xyz", "pwhash" : "nhedhz5h"});  
db.users.insertOne({  
  "name" : "René Wörzberger",  
  "username" : "abc",  
  "telephone" : {  
    "private": "+49 123 321212",  
    "business": "+49 221-8275-4324"  
  },  
  "email" : [ "rene.woerzberger@th-koeln.de", "foo@bar.de" ]  
});
```

create the following state in a MongoDB database with two collections `credentials` and `user`, where `credentials` contains two documents and `users` one document, which is a little more complex. Each document contains an additional key `_id` by default whose value is unique and generated. In the example, denormalization leads to duplication of `"username" : "abc"` in two documents. This leads to inconsistencies if not all occurrences of this key-value get updated in case of a username change. This is accepted in aggregate-oriented databases in order to keep the aggregates self-contained documents in this case.

collection "credentials"	document	{ "_id" : "5efdb4bfec4389001f345ab3", "username" : "abc", "pwhash" : "ek3nd37n" }
	document	{ "_id" : "5efdb4bfec4389001f345ab4", "username" : "xyz", "pwhash" : "nhedhz5h" }
collection "users"	document	{ "_id" : "5efdb4bfec4389001f345ab5", "name" : "René Wörzberger", "username" : "abc", "telephone" : { "private": "+49 123 321212", "business": "+49 221-8275-4324" }, "email" : ["rene.woerzberger@th-koeln.de", "foo@bar.de"] }

Figure 155: Document data (example)

Then one can query a concrete user document by, e.g., the private telephone number like this

```
db.credentials.find({username : "xyz"});
```

which evaluates to

```
{"_id" : "5efdb4bfec4389001f345ab4", "username" : "xyz", "pwhash" : "nhedhz5h" }
```

i.e., a list of all documents that match the search criterium (just one in the little example).

A convenient and cost-free way to try out the MongoDB queries is:

<https://www.mplay.run/mongodb-online-terminal>

32.2.4 Relational

Relational data models come along with explicit schemas. They support arbitrary cardinalities in relations insofar that a single query can dereference foreign keys that point from one table to another. Therefore, they are particularly good at avoiding data redundancies if they are normalized. This not only avoids unnecessary storage allocations but also avoids denormalization anomalies (cf. Section 33.1). However, locality decreases with increasing normalization: The more dereferencing by a relational DBMS takes place, the more disk I/O is necessary. Moreover, it is harder for a developer to make a conceptual (mental) model out of the data if it is scattered among many tables.

Relational DBMSs are suited particularly for domains that are well understood and allow for modeling decently sized schemas a priori.

Most relational DBMS store the data **row-wise**, cells in a single row are local to each other and likely to be in the same block on disk. Therefore, such relational DBMS can fetch entire rows with lesser disk I/O and are thus suited for online transaction processing (OLTP) access patterns. If they store **column-wise**, fetching an entire row for is comparably cheap. Such columnar relational DBMS are suitable for online analytics processing (OLAP), which typically queries entire columns for certain aggregations (like SUM).

EXAMPLE 164: SQL USAGE

Start a PostgreSQL Docker container via

```
docker run --rm --name postgres \
-e POSTGRES_HOST_AUTH_METHOD=trust -p 5432:5432 postgres:15.15
```

Connect a psql client to it via

```
docker exec -it postgres psql -U postgres
```

You can then define an explicit schema like this

```
CREATE TABLE countries (id int PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY, name
text, code char(2));
CREATE TABLE users (id int PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY, name
text, country int REFERENCES countries (id));
CREATE TABLE products (id int PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY, name
text);
CREATE TABLE users_products (usr int REFERENCES users (id), product int
REFERENCES products (id));
```

and put data into the tables like this

```

INSERT INTO countries (name, code) VALUES ('Germany', 'DE');
INSERT INTO countries (name, code) VALUES ('France', 'FR');

INSERT INTO products (name) VALUES ('iPhone 10');
INSERT INTO products (name) VALUES ('Galaxy S10');
INSERT INTO products (name) VALUES ('Nokia 6210');

INSERT INTO users (name, country) VALUES ('Steinmeier', (SELECT id FROM countries
WHERE code='DE'));
INSERT INTO users (name, country) VALUES ('Merkel', (SELECT id FROM countries
WHERE code='DE'));
INSERT INTO users (name, country) VALUES ('Macron', (SELECT id FROM countries
WHERE code='FR'));

INSERT INTO users_products (usr, product) VALUES (1,1);
INSERT INTO users_products (usr, product) VALUES (1,2);
INSERT INTO users_products (usr, product) VALUES (1,3);
INSERT INTO users_products (usr, product) VALUES (3,2);

```

which would produce a database state like this

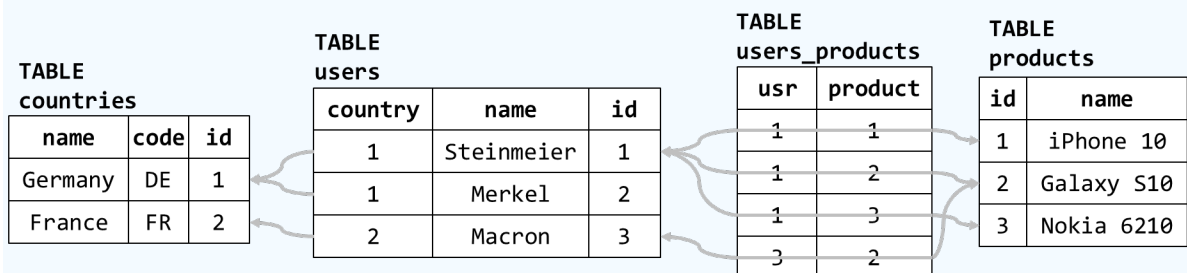


Figure 156: Relational data (example)

Note that the table `users_products` is just for establishing a many-to-many relationship. Even though the data is distributed among several tables, you can figure out which user has rated which product by a single query like

```

SELECT users.name, products.name FROM users
JOIN users_products ON users.id = users_products.usr
JOIN products on products.id = users_products.product;

```

which evaluates to

name	name
Steinmeier	iPhone 10
Steinmeier	Galaxy S10
Steinmeier	Nokia 6210
Macron	Galaxy S10

Please note that SQL has many dialects. The above example just works with PostgreSQL.

32.2.5 Graph

In graph data model, relationships gain even more importance in comparison to relational data models. In a database with a graph data model, data is stored as maps of properties (key-value

pairs) in nodes. Nodes can be connected to other nodes with directed relationships (edges), which can also carry properties.

Graph query languages like Cypher allow for quantifying about relationships. This is particularly useful if a client wants to query a certain context of some data record, i.e., properties of nodes that are connected to a node either directly or indirectly to a certain degree and have certain properties. These kinds of queries are difficult to formulate in SQL as they typically require long cascades of UNION and JOIN statements.

EXAMPLE 165: CYPHER USAGE

Start a Neo4J container like so

```
docker run --rm --name neo4j \
-p 7474:7474 -p 7687:7687 -e NEO4J_AUTH="neo4j/password" neo4j:5.19.0-community
```

and login to <http://localhost:7474/> with neo4j / password

Then execute the following Cypher statements

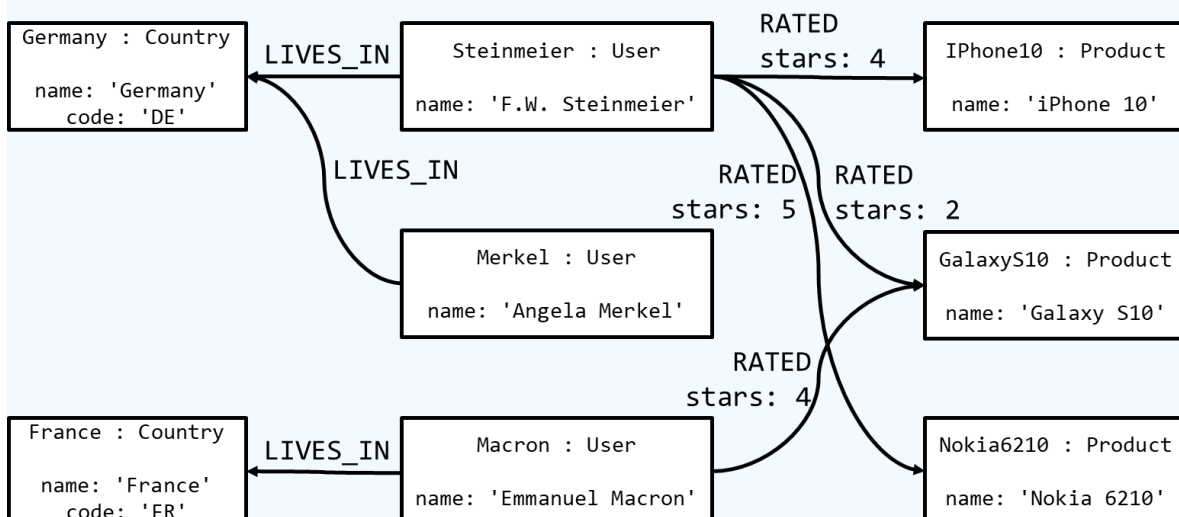
```
CREATE (Steinmeier:User {name: 'F.W. Steinmeier'})
CREATE (Merkel:User {name: 'Angela Merkel'})
CREATE (Macron:User {name: 'Emmanuel Macron'})
CREATE (DE:Country {name: 'Germany', code: 'DE'})
CREATE (FR:Country {name: 'France', code: 'FR'})

CREATE (Steinmeier)-[:LIVES_IN]->(DE)
CREATE (Merkel)-[:LIVES_IN]->(DE)
CREATE (Macron)-[:LIVES_IN]->(FR)

CREATE (Iphone10:Product {name: 'iPhone 10'})
CREATE (GalaxyS10:Product {name: 'GalaxyS10'})
CREATE (Nokia62010:Product {name: 'Nokia 6210'})

CREATE (Steinmeier)-[:RATED {stars: 4}]->(Iphone10)
CREATE (Steinmeier)-[:RATED {stars: 2}]->(GalaxyS10)
CREATE (Steinmeier)-[:RATED {stars: 5}]->(Nokia62010)
CREATE (Macron)-[:RATED {stars: 4}]->(GalaxyS10)
```

create the following database state



In comparison with the example of the relational data model, we have an additional property stars assigned to the RATED relationship.

For example, with a query like this

```
MATCH (country) <-[:LIVES_IN]- (user) -[:RATED {stars: 5}]-> (product)
RETURN country.name, product.name
```

a client can figure out those countries in which there is at least one citizen which gave a 5-star rating.

With

```
MATCH (user) -[r1:RATED]-> (product:Product) <-[r2:RATED]- (otherUser:User)
WHERE (r1.stars - 2 >= r2.stars) AND user <> otherUser
RETURN user.name, product.name, otherUser.name
```

a client can figure out those products that received two ratings that differ by at least 2 stars. These kinds of queries are useful when building recommendation engines.

A convenient and cost-free way to try out the Cypher queries in Neo4J is to start a Neo4J instance as a Docker container like so

```
docker run --name testneo4j -p 7474:7474 -p 7687:7687 -d \
--env NEO4J_AUTH=neo4j/password neo4j:5.19.0-community
```

This container serves a web UI at <http://host:7474>. You can log in with neo4j/password as set in the NEO4J_AUTH environment variable.

32.3 Derived Usages

Not every database class comes along with a unique data model. Often databases use one of the above but in a special way to support specific use cases.

32.3.1 Search & Analytics

It is worth noting that the way documents in a database are being indexed depends very much on the database engine and use case. In a document-oriented NoSQL database like MongoDB, documents are treated to be rather mutable, i.e., written (updated) often. It uses a [document index](#) a.k.a. [forward index](#) that maps the `_id` of the documents and its contents, so roughly speaking `document_id` → `content`. A search & analytics engine like Elasticsearch treats documents to be practically immutable (read-only). It uses an [inverted index](#) that maps single words to lists of documents where they occur, i.e., `content` → `document_id`.

EXAMPLE 166: ELASTICSEARCH

You can start a local Elasticsearch database via

```
docker run --rm -p 9200:9200 -e "discovery.type=single-node" \
-e "xpack.security.enabled=false" elasticsearch:9.2.1
```

```
export ELASTIC_URL=http://localhost:9200
```

In Elasticsearch, you can create an index called `products` with a schema of three attributes like so:

```
curl -X PUT "$ELASTIC_URL/products" -H 'Content-Type: application/json' -d \
'{
  "mappings": {
    "properties": {
      "name": { "type": "text" },
      "price": { "type": "double" },
      "brand": { "type": "keyword" }
    }
  }
}'
```

Note that “index” in Elasticsearch is a little ambiguous. Here, it is rather comparable to a table in relational databases.

Data can be put in bulk into products like so:

```
curl -X POST "$ELASTIC_URL/products/_bulk" \
-H 'Content-Type: application/x-ndjson' -d '
{ "index": {} }
{ "name": "Wireless Headphones", "price": 99.9, "brand": "SonicCo" }
{ "index": {} }
{ "name": "Gaming Mouse", "price": 59.0, "brand": "Clickr" }
'
```

Queries depend on the attribute datatypes. Elasticsearch particularly shines in text searches:

```
curl -X GET "$ELASTIC_URL/products/_search?pretty" \
-H 'Content-Type: application/json' -d '
{
  "query": { "match": { "name": { "query": "headphnoes", "fuzziness":
"AUTO" } } } }'
```

still finds a document despite the typo

```
{ "name": "Wireless Headphones", "price": 99.9, "brand": "SonicCo" }
```

thanks to the inverted index and due to the permitted fuzziness.

32.3.2 Time Series

The main characteristic of time series data is

- that a timestamp attribute is mandatory in each data record,
- that records are immutable
- new records are typically appended instead of inserted, and
- that the data can be effectively queries with respect to time, e.g. by specifying time ranges.
- Besides that, the underlying data model of time series databases is typical one of the others.

EXAMPLE 167: TIME SERIES WITH MONGODB

With a running MongoDB and MongoShell (cf. Example 163:) one can generate a collection like so

```
db.createCollection("sensor_readings", {
  timeseries: {
    timeField: "timestamp",
    metaField: "metadata",
    granularity: "minutes"
  },
  expireAfterSeconds: 60 * 60 * 24 * 30 })
```

and simulate the generation of sensor data like so

```
db.sensor_readings.insertMany(
  Array.from({length:24}, (_,i)=>({
    timestamp:new Date(Date.now()-(24-i)*3600000),
    value:18+Math.random()*10})))
```

For example, the sensor data of the last three hours can be queried like so

```
db.sensor_readings.find({ timestamp: { $gte: new Date(Date.now() - 3*3600000)}})
.sort({ timestamp: 1 })
```

yielding

```
[
  {
    timestamp: ISODate('2025-11-17T09:01:49.054Z'),
    value: 26.299463720127797,
    _id: ObjectId('691b009d0016fa5f469dc2b6')
  },
  {
    timestamp: ISODate('2025-11-17T10:01:49.054Z'),
    value: 22.490908431708455,
    _id: ObjectId('691b009d0016fa5f469dc2b7')
  }
]
```

32.3.3 Vector

A vector database stores records containing numerical vectors (plus identifiers and other metadata). The most common use case for vector databases is the storage of embeddings which are high-dimensional vectors (thousands of components) that are used in Generative Artificial Intelligence. Vector databases particularly allow for efficiently queries of vectors, e.g. vectors that diverge not much from a given vector and thus represent a similar meaning.

Again, the underlying data model may be one of the others. For example, you can PostgreSQL with its relational data model together with the plugin `pg_vector` can be used as a vector database or MongoDB with its document-oriented data model.

EXAMPLE 168: VECTOR DATABASE WITH ATLAS-MONGODB

With a running MongoDB and MongoShell (cf. Example 163:)

Then create some embeddings along with a search index

```
db.docs.insertMany([
  {_id:1,text:"Java Spring microservices",embedding:[0.9,0.1,0.0]},
  {_id:2,text:"Distributed systems intro",embedding:[0.8,0.2,0.1]},
  {_id:3,text:"Baking sourdough bread",embedding:[0.0,0.1,0.9]},
  {_id:4,text:"Event-driven arch with Kafka",embedding:[0.85,0.15,0.0]}
]);
db.docs.createSearchIndex("vector-index","vectorSearch",{
  fields:[{type:"vector",path:"embedding",numDimensions:3,similarity:"cosine"}]
});
```

Given a vector q , one can find cosine-similar vectors and what they embed like so

```
const q=[0.85,0.15,0.05];
db.docs.aggregate([{$vectorSearch:
  {index:"vectorindex",path:"embedding",queryVector:q,numCandidates:100,limit:3}},
{$project:{_id:0,text:1,score:{$meta:"vectorSearchScore"}}});
```

which yields the result

```
[
  { text: 'Event-driven arch with Kafka', score: 0.9991631507873535 },
  { text: 'Java Spring microservices', score: 0.9981407523155212 },
  { text: 'Distributed systems intro', score: 0.997789740562439 }
]
```


33 Consistency in Unscaled Databases

Data in a database is usually related and thus must be modified in an all-or-nothing way. Moreover, a single database server typically has several clients which access the database concurrently. This poses the risk that a client 2 can see intermediate changes by another client 1.

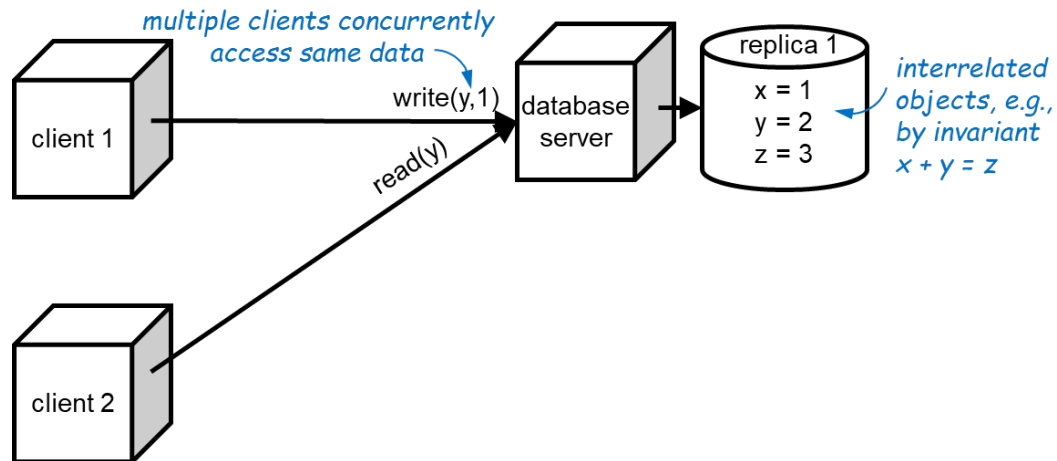


Figure 157: Main consistency challenges in unscaled databases

These challenges are independent of the data model a DBMS supports (cf. Section 32). Furthermore, they are not exclusive to databases but occur in every concurrent system. This includes common file systems, where clients are local user-space processes, objects are files, and the database server is the operating system kernel. The first challenge is also an issue for, e.g., multiple CPU cores that share a second level cache.

Without proper treatment, anomalies arise that cause inconsistencies.

33.1 Denormalization Anomalies

Some anomalies already occur in the simplest distribution variant of Figure 157, where there is just a single client that queries data in a single replica. These anomalies usually refer to relational data models in which table rows are the target of insertion, deletion, and update operations. Rows might have constraints imposed by an explicit schema that forces clients to, e.g., set multiple cells in the row. If data is denormalized, then conceptually identical values occur in multiple rows. A client must therefore change that value at every occurrence in case of an update. In case of a failure to do so, we speak of anomalies. Of course, these kinds of anomalies also apply to, e.g., document-oriented DBMS where we speak of documents and collections instead of rows and tables.

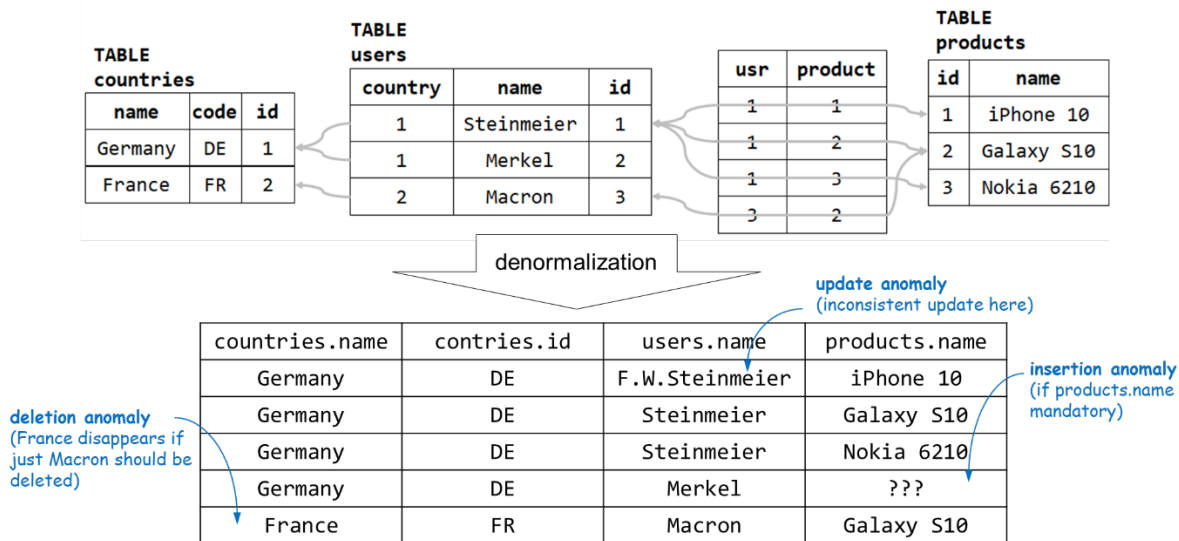


Figure 158: Multi-object anomalies

33.1.1 Insertion Anomaly

The insertion anomaly occurs, if not all mandatory values in a to-be inserted row are available. Conversely, in a normalized schema these could be offloaded into a separate table whose rows refer to the original one.

33.1.2 Deletion Anomaly

The deletion anomaly occurs in denormalized tables and refers to the fact that sometimes deletion of a row deletes more than desired. This is particularly true if a row contains more than one conceptual entity.

33.1.3 Update Anomaly

An update anomaly occurs if the same data (of a single entity) is redundantly stored in multiple rows but not consistently updated.

33.2 Atomicity

DBMS usually provide a basic notion of atomicity: A single write access either succeeds in writing a whole "single atomic data record" or is aborted and thus changes nothing at all. It mostly depends on the data model, what the DBMS considers "single atomic data record", which are typically like this:

- In key-value data model, an atomic data record is a single value.
- In a relational data model, an atom is a single row, and
- in a document-oriented data model, an atom is a document.

Ensuring atomicity for any "atom" that is larger than a single bit, is not trivial to implement in a DBMS. Think of a failure halfway during the write of a larger document. However, we do not go into details how this is guaranteed by DBMSs.

Anyway, atomicity on just single data records does not help with, e.g., the update anomaly (cf. Section 33.1) where the data to update is scattered among multiple data records. Thus, many DBMS support explicit **transactions** that are sequences of read and write accesses. Atomicity w. r. t. transaction means that all writes in a single transaction are executed in an all or nothing manner.

33.3 Concurrent Access Anomalies

In order to avoid performance degrading bottlenecks, transactions from multiple clients are supposed to be executed concurrently. However, this bears the risk of inconsistencies.

In the following, we just talk about simple objects x , y , and z with integer values and optionally with constraints like $x+y=z$ like in Figure 157. We denote by $r1(x)=6$ the event that a transaction 1 reads (hence r) the value 6 from key x . Another example would be the event $w2(y, 2)$ in a transaction 2 that writes (hence w) the value 2 to key y . By $c1$ and $a2$, we denote the commit of transaction 1 and abort (rollback) of transaction 2, respectively. By $r1(x)=6 \quad w2(y, 2)$, we denote a chronology of events, i.e., that the write comes after the read.

33.3.1 Dirty and Skewed Concurrent Access

The following Table 52 summarizes the first four concurrent access anomalies. **Dirtness** is about concurrent access to the very same object that has uncommitted changes. **Skewness** refers to concurrent access to distinct data objects that are related and therefore one object may not be changed freely without regard to the other.

transaction 1	transaction 2	same data	distinct but related data
writes	just reads	dirty read	read skew
	(reads and) writes	dirty write	write skew

Table 52: Dirty and skewed concurrent access

a) Dirty Read

A **dirty read** occurs if a transaction 2 reads uncommitted data. This uncommitted data may even be written in a concurrent transaction 1 that is later aborted instead of committed.

EXAMPLE 169: DIRTY READ

Precondition: $x=1$

History with two concurrent transactions:

transaction 1: $w1(x, 5)$ $a1$
transaction 2: $r2(x)=5$ $c2$

If transaction 1 performs a rollback, $r2(x)=5$ should never have happened.

b) Dirty Write

A **dirty write** occurs, if two transactions update the same object.

EXAMPLE 170: DIRTY WRITE

Precondition: $x=1$

History with two concurrent transactions:

transaction 1: $w1(x, 2)$ $c1$
transaction 2: $w2(x, 3)$ $a2$

Here, it would be unclear whether x has to be reset to 1 as it was before transaction 2 or to 2, because of the prior write of transaction 1.

c) Read Skew

A **read skew** might occur if a transaction 1 reads multiple objects, a concurrent transaction 2 writes to the same objects and the objects are related, e.g., by an invariant constraint.

EXAMPLE 171: READ SKEW

Precondition: $x=3, y=4$, with invariant constraint $x < y$

History with two concurrent transactions:

transaction 1:	$w2(x,1)$	$w2(y,2)$	$c2$
transaction 2:	$r1(x)=3$	$r1(y)=2$	$c1$

Here, transaction 2 sees $x=3$ and $y=2$ that violates $x < y$.

d) Write Skew

A **write skew** might occur if two transactions write to two distinct but related objects.

EXAMPLE 172: WRITE SKEW

Precondition: $x=1, y=3$, with invariant constraint $x < y$

History with two concurrent transactions:

transaction 1:	$r1(x)=1$	$r1(y)=3$	$w1(y,2)$	$c1$
transaction 2:	$r2(x)=1$	$r2(y)=3$	$w2(x,2)$	$c2$

Here, transaction 1 finally violates the constraint by effectively setting $x=y=2$.

33.3.2 Repetitive Concurrent Access

Another class of anomalies happen in transactions that access the very same object  at least twice and assume that x remained unmodified in the meantime.

a) Non-repeatable Read

In **non-repeatable read** (a.k.a. fuzzy read), a transaction 1 tries to reread an object, which has been updated or deleted and committed in the meantime by another transaction 2. So, the transaction 1 cannot reread the object.

EXAMPLE 173: NON-REPEATABLE READ

Precondition: $x=1$

History with two concurrent transactions:

transaction 1:	$r1(x)=1$	$r1(x)=2$	$c1$
transaction 2:	$w2(x,2)$	$c2$	

Transaction 1 would "expect" $r1(x)=1$ at the end.

b) Phantom

A **phantom** is like non-repeatable read but refers to collections of objects. If a transaction 1 queries a certain collection of values, i.e., table rows that match certain criteria, another transaction 2 inserts an object, i.e., a row that matches the criteria, transaction cannot "re-query" the same result again.

c) Lost Update

A **lost update** occurs if a transaction 1 reads an object, a transaction 2 writes that object and then transaction 1 (unknowing of the updated) updates / overwrites that object based on the prior read.

EXAMPLE 174: LOST UPDATE

Precondition: $x=1$

History with two concurrent transactions that want to increment x by 1:

transaction 1: $r1(x)=1$ $w1(x,2)$ c1

transaction 2: $r2(x)=1$ $w2(x,2)$ c2

If these transactions had been executed serially, the outcome would have been $x=3$.

33.4 Isolation of Transactions

The anomalies of Section 33.3 can occur if we have multiple clients that concurrently access a database (regardless of whether the database consists of a single or multiple replicas) and overlap in the data they access. Transactions can therefore be used not just for establishing atomicity but also for **isolation** of concurrent accesses. In perfect isolation, transactions seem to run in an arbitrary sequential order. However, such perfection isolation hampers concurrency and thus throughput of a DBMS.

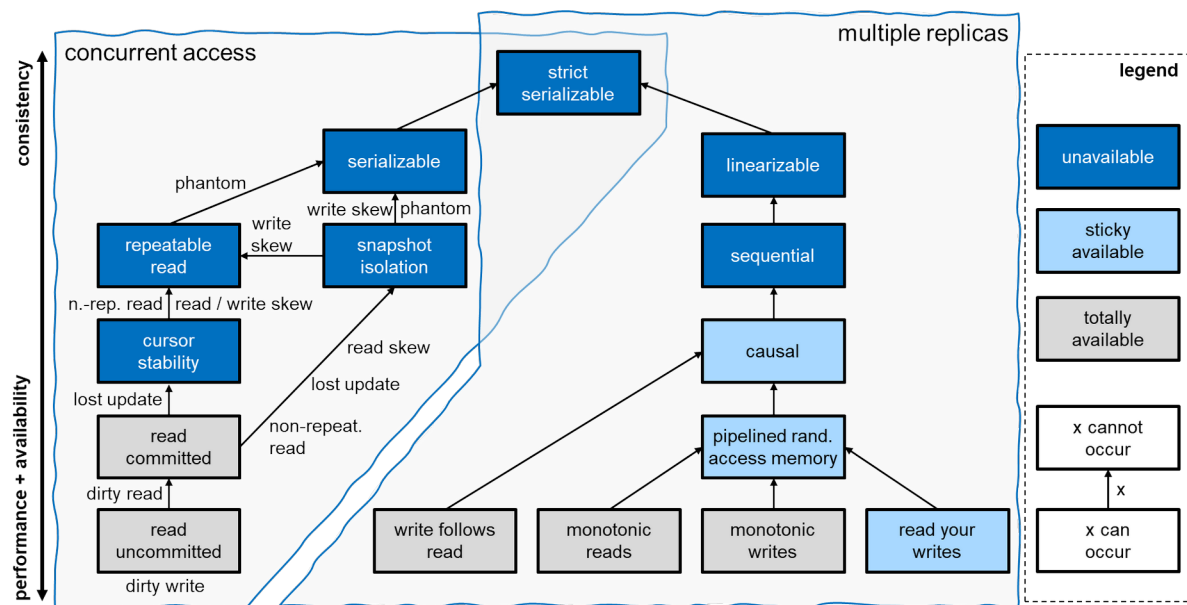


Figure 159: Hierarchy of consistency models (after)

Figure 159 provides a hierarchy of consistency models. On the left-hand side, we have the concurrent access anomalies of Section 33.3 labelling edges. The nodes are labelled with those isolation levels that rule out the anomalies on the incoming edges. (The multi-replica anomalies on the right-hand side will be dealt with in Chapter 35).

Serializability is often achieved with the concurrency control method **two-phase locking (2PL)**. If a certain database object x is written in a transaction t_w , then t_w first has to acquire an exclusive lock on x , which is released if t_w commits or aborts. During t_w , x cannot be even read let alone written by any other transaction, which lowers the performance of the DBMS.

One must take the explanations above about concurrent access anomalies and isolation levels with a grain of salt: Actually, there is no exact standardization about what anomalies and isolation level precisely mean. That is why different DBMS products provide different configurable levels of isolation and what anomalies these isolation levels permit or exclude also differs. Consolidation and standardization of anomaly categories and isolation levels is an active research field .

EXAMPLE 175: POSTGRES 15 ISOLATION LEVELS

Postgres 15, for example, effectively provides the four isolation levels “read uncommitted”, “read committed”, “repeatable read”, and “serializable” . However, “repeatable read” actually is “snapshot isolation” in the sense of Figure 159 since write skews are still possible in PostgreSQL’s “repeatable read” level.

34 Scaling Databases

In Chapter 18, we have seen how to scale out nodes (processes), which are an abstraction of computing. Databases, which abstract from storage, can be scaled out, too. However, scaling databases vertically is intrinsically harder than scaling nodes since one has to deal with a complex, durable, and distributed instead of a volatile job that can basically be forgotten once it has been processed.

This chapter deals with the strategies for out-scaling databases, namely partitioning and replication.

34.1 Partitioning

A database can be split such that its data files are chunked into smaller data files that are managed by different database servers. This kind of non-redundant out-scaling improves performance of a system as client requests can be distributed among the database servers, whose computational resources (mainly CPU and disk I/O) add up.

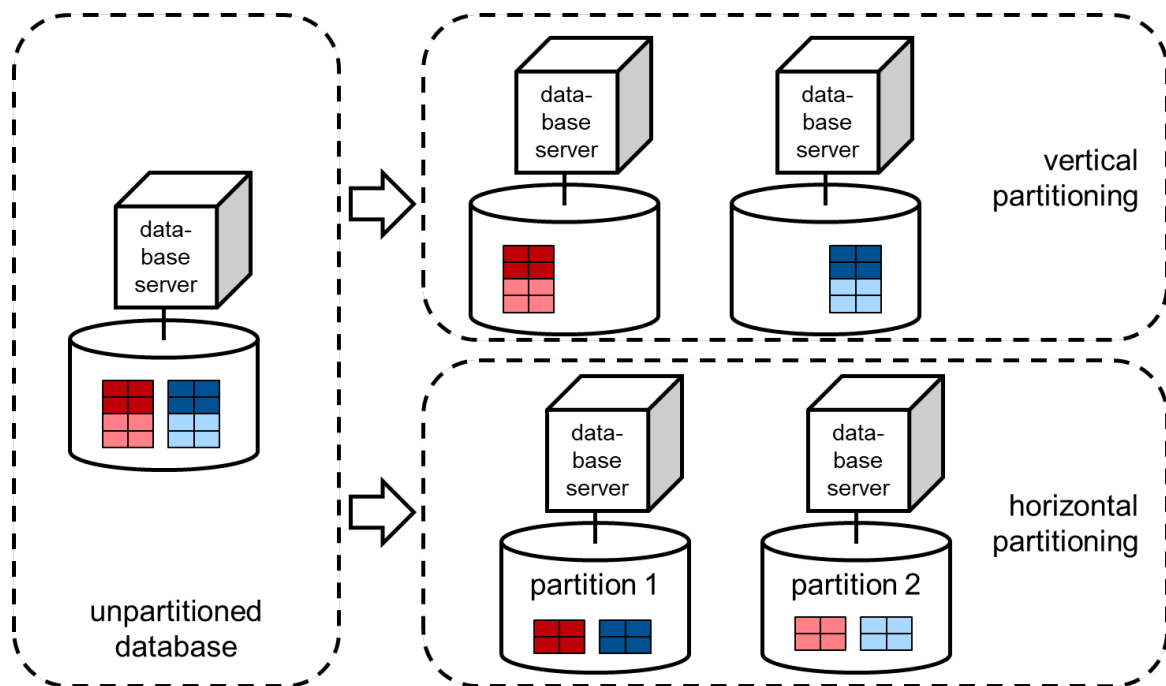


Figure 160: Vertical and horizontal database partitioning

34.1.1 Vertical Partitioning

Partitioning can be **vertical**, i.e., larger but potentially interrelated database objects like a tables or documents, columns, or indexes each are kept as a whole and managed by a single database server.

34.1.2 Horizontal Partitioning

Partitioning can also be **horizontal**: Here, different rows of the same table or different documents of the same collection are stored in different data files called **partitions**. If these partitions are stored on different database nodes, they are called **shards**.

Of course, clients should be kept agnostic of partitioning. A DBMS should expose a partitioned database still as a single one and a client should not bother with locating the relevant partitions for certain query on its own. For partitioning, a certain part of a data record, i.e., certain column in a table or value of a key that recurs in all documents of a collection, is designated to be the **partition key**. We can apply the same algorithms to balance the storage in the partitions that are used for load balancing as described in Section 19.3.1b) to determine a partition for a data record, consistent hashing most notably. Instead of clients with client identifiers, we have data records with keys, instead of destination servers we speak of partitions and the DBMS plays the role of a load balancer. Of course, server affinity is an indisputable property in this context since we want to find the partition to which a certain data record went later on.

34.2 Replication

In order to increase performance as well as availability, one can store the same database or the same database partitions redundantly in different storages each managed by a dedicated database server. We speak of **replicas** in this case.

The cost of replication is, of course, that storage capacity demands increase. Moreover, writes to data cannot be replicated instantly, which impacts consistency – more on that in Chapter 33.

Anyway, there are different models for replication:

34.2.1 Leader-follower Replication

In a leader-follower replication, each set of n identical replicas has a designated and authoritative **leader** and $n - 1$ **followers**. Combined with partitioning, a single database server can serve both leader replicas and follower replicas (cf. Figure 161). The leader replica, a.k.a. the primary replica, is the one in the replica set that accepts writes. Conversely, the followers can only accept read accesses.

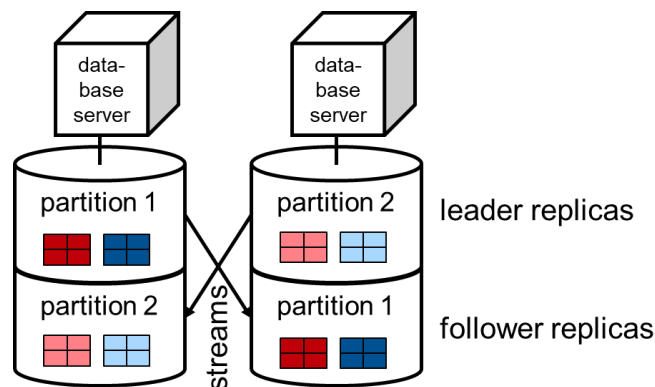


Figure 161: Database replication with a leader-follower model

For sake of brevity, we treat the replica and the database server connected to it as a whole and call just replica in the following.

34.2.2 Multi-Leader Replication

In a **multi-leader replication**, n identical replicas can be distinguished to l authoritative leaders and $n - l$ followers. If just one leader receives write accesses at a time and the others serve as read replicas (followers) and get promoted just in the case of a leader's failure, we speak of **passive-active replication**. If all leaders receive write accesses in normal operation, we speak of **active-active replication** (cf. Section 8.5.2).

EXAMPLE 176: GOOGLE SQL AS MULTI-LEADER PASSIVE-ACTIVE REPLICATION

For example, multi-leader passive-active replication is the model for replication in Google SQL (cf. Section 24.9.10).

34.2.3 Leaderless Replication

In a leaderless replication, there is no distinction between leaders and followers (mere read replicas). Instead, all read and write accesses from clients go finally to all replicas.

Of course, in such a model, replicas might render an inconsistent database state since some replicas might serve stale data.

34.3 Coordination and Load Balancing

Replication would be pointless without a strategy which requests from clients go to which replica. This strategy must take care of the following constraints:

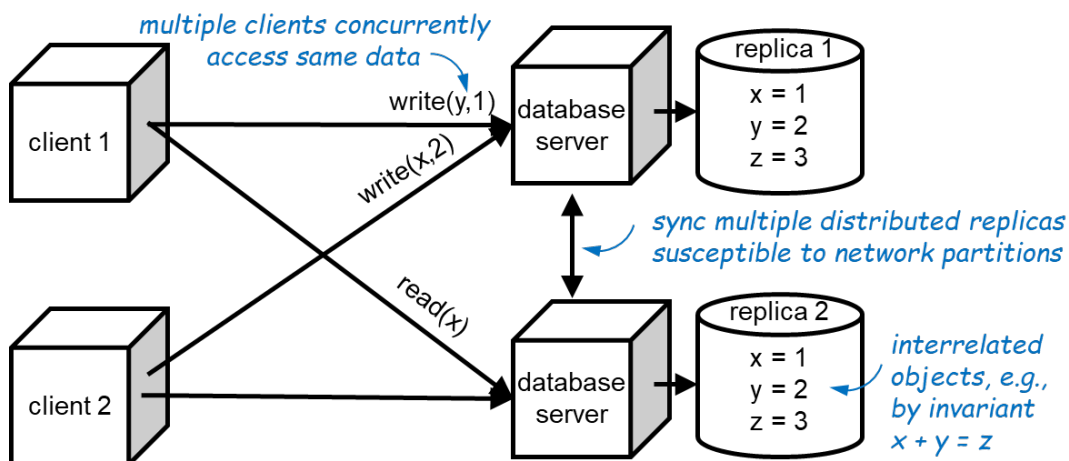
1. In a leader-follower or multi-leader replication, writes may only go to leaders.
2. In a leaderless replication, writes must go to several replicas to reach a quorum (cf. Section 35.2)
3. Depending on the consistency level, clients must be stickily connected to a certain replica or not (cf. Section 33.4).
4. Replicas should be evenly utilized.

Database clients might be aware of replication and choose one or replicas to connect to on their own (cf. Section 19.2.1). Or clients connect to a dedicated proxy that acts as a coordinator and load balancer and decides which replica should receive a query. A mixture of both is also not uncommon, where clients have two connection each with a separate proxy: One for writes that are forwarded to leader replicas and one for reads that are forwarded to follower replicas.

35 Consistency in Scaled Databases

In an out-scaled database tier, replicas need to be constantly synchronized such that queries can be answered consistently with the most recent value. Yet, consistency in scaled databases comes with a whole new set of challenges:

- **propagation delays**, i.e., the non-zero time between the write to an object x in one (leader) replica and the replicating write to x in another (follower) replica,
- **crash faults**
 - **unavailability** of replica nodes, i.e., nodes that have crashed and do not respond to any other nodes and
 - **network partitions** that render a certain replica available for a subset of other replicas and unavailable for another subset,
- **uncooperative** replica nodes that (maliciously) deliberately cause inconsistencies,
- **conflicting writes** may be made to the same object x in different replicas, which may happen in active-active multi-leader or leaderless replication.



35.1 Consistency Levels

Due to propagation delays or crash faults, the following can happen in a scaled database:

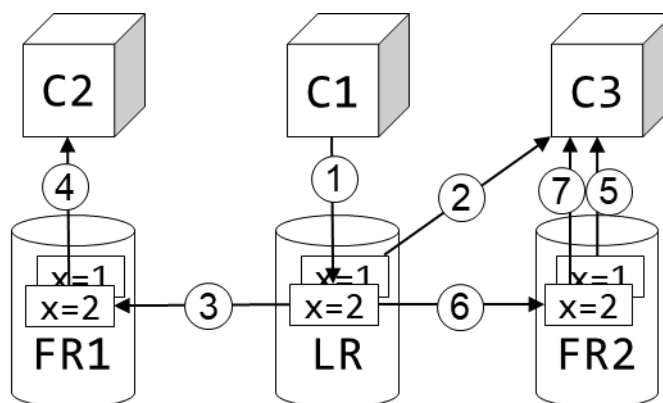


Figure 162: Immediate vs. eventual consistency

In a leader-follower replication model with three clients C1, C2, and C3, a leader replica LR, two follower replicas FR1 and FR2 and just the value $x=1$ in the database the following chronology could happen:

1. Client C1 writes $x=2$ to leader replica LR
2. Client C3 reads $x=2$ from leader replica LR
3. Leader replica LR propagates $x=2$ to follower replica FR1
4. Client C2 reads $x=2$ from follower replica FR1
5. Client C3 reads $x=1$ from follower replica FR2
6. Leader replica LR propagates $x=2$ to follower replica FR2
7. Client C3 eventually reads $x=2$ from follower replica FR2

In this sequence, all clients eventually read the most recent value $x=2$ yet the database responds temporarily inconsistent. Hence, we speak of **eventual consistency**.

The consistency models on the right-hand side of Figure 159 pose recency guarantees for values in a replicated database to various degrees. With **monotonic reads**, steps 2 and 5 cannot occur in that order: If a client C3 has read a value, it cannot read an older one afterwards. **Immediate consistency** a.k.a. **linearizability** [24, p. 324] is an even stronger consistency guarantee which ensures that steps 4 and 5 cannot occur in that sequence: When a read returns the most recent value, all subsequent reads from *any* client must do so as well.

The colors of Figure 159 indicate the quality attribute trade-off between consistency and availability that has to be done is an out-scaled database with multiple replicas and thus refine the CAP theorem of Section 9.1. In the case of a network partition, i.e., a network disconnection (network partition) between two replicas, a client of a DBMS

- can still access all replicas (**totally available**) if the isolation level is, e.g., “read committed”,
- must stick to a certain replica (**sticky available**) if the consistency model is “read your writes”, and
- must be blocked (**unavailable**) otherwise.

Immediate consistency can be achieved with quorums or consensus algorithms.

How consistency is best traded off against performance depends on the usage of the database. The following table discusses some use cases.

Use Case	Typical Readers / Writers	Data Model	Isolation / Consistency & Requirements	Reasoning / Trade-Off
Banking / Finance	few / few	Relational	Serializable – strict ACID; very high consistency, low throughput	Prevents double spending; transactions behave as if executed sequentially.
Airline Booking	few / few	Relational	Serializable – strict isolation; high consistency, low concurrency	Avoids overbooking; correctness prioritized over performance.
E-Commerce Checkout / Inventory	moderate / moderate	Relational (+ Cache)	Repeatable Read → Serializable – strong consistency on stock; medium throughput	Ensures order and inventory correctness; browsing can tolerate staleness.

Use Case	Typical Readers / Writers	Data Model	Isolation / Consistency & Requirements	Reasoning / Trade-Off
Product Catalog Search	many / few	Search & Analytics Engine	Eventual consistency – moderate accuracy, very high throughput	Updates may take ~1s to appear; optimized for fast search queries.
Social Media Feed / Likes	many / few	Document-oriented / Search & Analytics Engine	Eventual consistency – low strictness, very high scalability	Slightly outdated like counts acceptable; prioritizes latency and scale.
Gaming Leaderboard	many / moderate	Relational / Key-Value (e.g., Redis)	Repeatable Read – medium consistency and concurrency	Prevents rank jumps during a session; slight delay acceptable.
IoT Sensor Data	few / many	Time-Series DB	Read Committed – moderate consistency, very high write throughput	Continuous ingestion prioritized; readers tolerate slightly stale data.
Analytics / Reporting	many / few	Search & Analytics Engine / OLAP DB	Snapshot / Eventual – moderate consistency, very high throughput	Queries run on consistent snapshots; slight staleness acceptable.
Log Aggregation	few / many	Search & Analytics Engine	Eventual consistency – low strictness, very high ingestion rate	High write volume prioritized; delayed visibility acceptable.
Recommendation / Autocomplete	many / few (batched)	Search & Analytics Engine	Eventual consistency – moderate accuracy, high throughput	Periodic re-indexing fine; prioritizes fast query responses.
Knowledge Base Search	many / few	Search & Analytics Engine	Eventual consistency – low strictness, high throughput	Updates can lag slightly; optimized for fast lookups.
Messaging / Chat App	many / many	Document / Key-Value	Eventual consistency – medium consistency, high performance	Guarantees per-chat order; global ordering unnecessary.
Stock Trading Engine	moderate / moderate	Relational + Key-Value (Cache)	Serializable – full isolation, medium throughput	Precise trade sequencing required; correctness over speed.
Ad Impression Tracking	many / many	Time-series + Wide-Column	Eventual consistency – low strictness, very high throughput	Approximate counts acceptable; focus on scalability and availability.

Table 53: Isolation Level Tradeoffs for Different Use Cases

35.2 Data Versioning and Quorums

If one of three replicas have not received a write to datum x , e.g., due to a propagation delay or crash-fault of the respective database server, a subsequent read from all replicas might return the

stale old value and two times the new value for x . This is why each data record is assigned with a version field such that recency of read data can be decided.

Even with versioning, there remains a problem: If exactly those replicas get a read for x which still did not get the most recent value for x , a client still would get a stale x . This is why a client—or usually a coordinator (load balancer) within the DBMS on the client's behalf—must get an acknowledgment from a certain number of replicas—a **quorum**—before a write or read is considered successful.

Since several replicas must respond, the longest response time determines the response time from a client's point of view. However, it is not necessary to get a response from all replicas to consider a read or write successful. Instead, if we have

- n replicas and define
- w to be the minimal number of replicas that must acknowledge a write access and
- r to be the minimal number of replicas that have to respond to a read access,

then the most recent value must be in at least one of the read responses, if

$$w + r > n.$$

In this case, writes and reads overlap in at least one replica that holds the most recent value.

35.3 Consensus

In certain cases, quorums are insufficient. Instead, **all** replicas must agree on the same most recent value of a certain data object.

35.3.1 Crash-Fault Tolerant

The following Table 54 provides an overview of consensus algorithms. In this table, we assume the replicas to be cooperative, i.e., each replica either responds correctly or not at all in case of a crash. The consensus algorithms therefore just need to be **crash-fault tolerant (CFT)**. These algorithms require that strictly more than half of all replicas work correctly at every given time.

Algorithm	Fault Tolerance	Scalability	Latency	Energy Efficiency (mW/Node)	Consistency	Throughput (TPS)	Network Overhead	Typical Use Cases Widely Known Products/ Technologies
Paxos	Less than 50% nodes	Up to dozens of nodes	10-100 ms	0.01 - 1	Strong (Serializable)	100-1000	High	Leader election Google Spanner, Chubby, Microsoft Azure Storage, IBM Cloudant
Raft	Less than 50% nodes	Up to dozens of nodes	10-100 ms	0.01 - 1	Strong (Serializable)	100-1000	High	Leader election, state machines replication etcd, Consul, CockroachDB, RethinkDB, InfluxDB

Table 54: Consensus algorithms for cooperative replicas

35.3.2 Byzantine-Fault Tolerant Consensus

A byzantine fault is a fault, where a replica or node holds back or produces erroneous messages, which may be just erratic but can also be malicious. The following algorithms are **byzantine-fault tolerant algorithms (BFT)**. These algorithms require that strictly more than one third of all nodes are working correctly at any given time.

Algorithm	Fault Tolerance	Scalability	Latency	Energy Efficiency (Wh/txn)	Consistency	Throughput (TPS)	Network Overhead	Typical Use Cases Widely Known Products/Technologies
Proof of Work (PoW)	Less than 50% nodes	High (up to thousands of nodes)	Minutes to hours	143 - 3333	Eventual (Read Committed)	3-7	Moderate	Cryptocurrencies (e.g., Bitcoin) Bitcoin, Ethereum (pre-2.0), Litecoin, Monero
Proof of Stake (PoS)	Less than 50% nodes	High (up to thousands of nodes)	Seconds to minutes	1 - 50	Eventual (Read Committed)	20-100	Low	Cryptocurrencies (e.g., Ethereum 2.0) Ethereum 2.0, Cardano, Polkadot, Tezos
Delegated Proof of Stake (DPoS)	Less than 50% nodes	High (up to thousands of nodes)	< 1 second	0.01 - 0.1	Eventual (Read Committed)	1000-3000	Low	Cryptocurrencies (e.g., EOS, Tron) EOS, Tron, Steem, BitShares
Practical Byzantine Fault Tolerance (PBFT)	Less than 33% nodes	Up to dozens of nodes	< 10 ms	0.02 - 0.5	Strong (Serializable)	1000-5000	Very High	High-security applications, blockchain platforms Hyperledger Fabric, Zilliqa, Ripple
Tendermint	Less than 33% nodes	High (up to hundreds of nodes)	1-2 seconds	0.01 - 0.5	Strong (Serializable)	1000-10000	Moderate	Blockchain platforms (e.g., Cosmos) Cosmos, Binance Chain, Terra

Table 55: Consensus algorithms for uncooperative replicas

35.4 Conflict Resolution

A replicated database that is managed by a single DBMS is normally designed to present its clients a consistent state and to keep its clients agnostic of temporary inconsistencies. In some system

architectures however, it is natural for users to temporarily work on leaderless replicated data that is temporarily inconsistent with other replicas.

This is particularly true for so-called **local-first software**, where

- each client has a local replica of some concurrently modified data,
- each replica is managed by a local DBMS, like IndexDB in browsers or even just a tree in a file system and there is no central DBMS that can coordinate a consensus on divergent data,
- network partitions between replicas are common due to unstable internet connection since, e.g., the user is somewhere out in the sticks, so the divergence between replicas can become arbitrarily large,
- there is no central authority like a server that is safe from malicious users and that enforces constraints like fine-grained user-based authorization for editing or viewing data or keeping certain values in balance like in bank accounts.

This makes local-first software suitable for applications like collaborative editing but less suitable for, e.g., online-banking applications. Prominent examples of local-first software are distributed version control systems like Git, where data modifications are first kept in a local repository (i.e., replica) until a deliberate `git pull / push` is executed by the respective user that synchronizes, i.e., converges two repositories again.

Local-first software rely on **optimistic replication** (as opposed to strongly consistent replication) as we accept inconsistencies here in favor of performance and availability but are “optimistic” in the sense that the replicas can be converged again, and inconsistencies can be resolved.

Making the convergence as seamless as possible even if the replicas have diverged massively, is a research field on its own. **Conflict-free Replicated Data Types (CRDT)** as defined in are a helpful tools to do make this possible. Please note that “conflict-free” just means that conflicts are always resolved automatically leaving the merge data in a technically consistent, e.g., well-formed, and syntactically correct state. “Conflict-free” does not mean that deviating user intents are resolved magically. If such deviating intents, like if two users modify the very same atomic original value to different new values, CRDTs are just optimistic in a second way: They assume that they merge data such that the merged result represents the intersubjective intention of its editors (users) or that non-intentional results are not harmful and can easily be manually corrected by one of the editors.

EXAMPLE 177: CRDT FOR LOCAL-FIRST SHOPPING APPLICATION

Consider a local-first shopping list application where a shopping list is replicated between the local databases.

At first, the shopping list just contains a single `[{ description: “apples”, done: false }]`. User A goes shopping and sets `[{ description: “apples”, done: true }]`. Meanwhile, B updates `[{ description: “apples and oranges”, done: false }]`.

A CRDT could merge both modifications into

`[{ description: “apples and oranges”, done: true }]`, which is technically a valid JSON object that can be parsed by the shopping list application. Yet, this merged results clearly is not the intersubjective intend of both users, as A comes home with no oranges. However, the application is free to highlight merged entries, such that A could be notified of the editing of user B.

This makes CRDTs different from version control systems like Git. In Git, conflicts that cannot be resolved automatically like edits in the same line, lead to explicit conflict markers that may leave

documents in and syntactically incorrect state. That way, source code files are becoming deliberately uncompileable.

From a programming theoretic view, CRDTs are abstract data types, e.g., data types whose instances may only be accessed (and mutated in particular) through a prescribed interface and fall into the following two categories.

35.4.1 State-based CRDT: Convergent Replicated Data Type (CvRDT)

Every two unequal, e.g., conflicting instances of a **Convergent Replicated Data Type (CvRDT)** a and b can be passed to a function $merge(a, b)$ to compute a converged variant c that may or may be not equal to either a or b . (In mathematical terms, the instances of a CvRDT form a semi-lattice and the function $merge$ computes the least upper bound of a and b .)

EXAMPLE 178: CONVERGENT REPLICATED DATA TYPE

An example for a CvRDT is the “increment-only Counter” (G-Counter) that encapsulates an integer value x that can only be mutated by an increment function $inc(x, s)$, where s is the positive integer value that is supposed to be added to x .

Now, if two replicas have different values a and b for a G-Counter, then $merge(a, b) := a + b$. This is reasonable as the increments in both replicas are just summed up.

Note that a CvRDT encapsulates a state which may consist of more than just the data, which is edited by users. It may particularly contain additional metadata like identifiers of other CvRDT instances or timestamps. The merge function operates on whole states that comprise the data and the metadata.

EXAMPLE 179: LAST-WRITER-WINS

The following TypeScript program implements a CvRDT with a Last-Writer-Wins strategy and is derived from .

```
#!/deno run
// general definition of a CvRDT
interface CvRDT<ValueT> {
  get value(): ValueT;
  set value(newValue: ValueT);
  merge(other: CvRDT<ValueT>): void;
}

// concrete implementation of the generic CvRDT interface
class LastWriterWinsCvRDT<ValueT> implements CvRDT<ValueT> {
  private readonly ownId: string;
  // these three members constitute the state to synchronize
  private lastWriterId: string;
  private timestamp: number;
  private _value: ValueT;

  constructor(ownId: string, initialTimestamp: number, initialValue: ValueT) {
    this.ownId = ownId;
    this.lastWriterId = ownId; // pseudo initial last writer = self
    this.timestamp = initialTimestamp;
    this._value = initialValue;
  }

  get value(): ValueT {
    return this._value;
  }

  set value(newValue: ValueT) {
    this.timestamp++;
    this.lastWriterId = this.ownId;
    this._value = newValue;
  }
}
```

```

/**
 * merge overwrites this' state by the others state
 * if the other's timestamp is higher or if it is
 * equal and the other's lastWriterId is higher;
 * otherwise there is no effect
 * @param other The other LastWriterWinsCvRDT that
 * should be merged
 */
merge(other: LastWriterWinsCvRDT<ValueT>): void {
  if (
    this.timestamp < other.timestamp ||
    (this.timestamp === other.timestamp &&
      this.lastWriterId <= other.lastWriterId)
  ) {
    this.lastWriterId = other.lastWriterId;
    this.timestamp = other.timestamp;
    this._value = other._value;
  }
}

// the following statements demontstrate how this CvRDT works
const alice = new LastWriterWinsCvRDT<string>("alice", 1, "");
const bob = new LastWriterWinsCvRDT<string>("bob", 1, "");

// network disconnect (merge disabled)
bob.value += "b"; // represents single keystroke
bob.value += "b";
alice.value += "a";
// network reconnect (merge enabled)
alice.value += "a";
bob.merge(alice); // timestamps both 2, but "bob" > "alice" -> merge rejected
console.log(bob.value); // "bb"

alice.value += "a";
bob.merge(alice); // now alice has higher timestamp -> overwrite bob
console.log(bob.value); // "aaa"

bob.value += "b";
alice.merge(bob);
console.log(alice.value); // "aaab"

```

35.4.2 Operation-based Commutative Replicated Data Type (CmRDT)

A **Commutative Replicated Data Type (CmRDT)** instance maintains a log of those interface method invocations that have mutated its state, i.e., the method name plus actual parameter values per log entry. A CmRDT guarantees that the methods are commutative, e.g., the invocations can be replayed in any order but arrive at the very same state. Therefore, two replicas can exchange invocations that already have been applied locally to each other in order to converge at the same state.

EXAMPLE 180: COMMUTATIVE REPLICATED DATA TYPE

The G-Counter is obviously also a CmRDT as the order of invocations of $inc(x, s)$ has no effect on the final state of the counter.

35.4.3 Automerger

Besides rather simplistic CRDTs like G-Counter which are of limited use, there are more complex CRDTs like Automerger, which is a CmRDT and supports replication and merging of JSON-serializable data objects or (collaboratively edited) rich-text documents.

36 Consistency in Distributed Databases

In Chapters 18 and 19 we have learned that horizontal scaling is a rather benign form of distribution: In a scaled system, all nodes run the same program. Likewise, all replicas and partitions and database servers in a horizontally scaled database follow the same data model and schema and run the same DBMS programs, respectively. Moreover, a DBMS can keep its clients agnostic of whether a database is scaled or not to some degree.

With **distributed database**, we mean a set of separate, potentially scaled databases, each of which is managed by a DBMS and which—as opposed to a single (scaled) database—

- might have database overarching consistency constraints,
- might be managed by multiple and potentially different DBMSs, each with a distinct network address,
- with potentially different data models,
- have no leader in any sense,
- communicate potentially over less reliable, higher latency and less secure network connections.

36.1 Distributed Database Transactions

A **distributed database transaction (DDT)** t is a set of sub-transactions $t = \{t_1, \dots, t_n\}$ which are executed by several DBMSs. Distributed transactions can basically be implemented in two different ways: Either they are “DBMS managed”, i.e., participating DBMSs follow a certain protocol, or they are “client managed”, i.e., the respective DBMS client programs jointly execute a distributed transaction.

Both approaches are similar with respect the consistency and durability properties in ACID: Of course, durability of a DDT can be delegated to the participating DBMS. Consistency can be enforced by each participating DBMS as long as the respective constraints refer just to local database objects. Conversely, they cannot enforce consistency if the constraints are DBMS overarching. A typical challenge are pseudo foreign keys, e.g., identifiers in one database that refer to identifiers in another database. Keeping referential integrity here must be delegated to clients.

Still, both approaches have trade-offs which are as follows:

36.1.1 Upsides of DBMS Managed DDTs and Downsides of Client Managed DDTs

On the upside, in DBMS managed DDTs the DBMSs care for keeping ACID properties of the distributed database, atomicity and isolation in particular. No additional complexity is added to the clients.

Conversely, in client managed DDTs, upholding ACID properties is partly delegated to the respective DBMSs’ clients. From the clients’ perspectives, local consistency and durability is not different than in a non-distributed transaction and thus poses no additional complexity. However, atomicity and isolation of the DDT must be ensured by the clients and can be done just to a certain extend.

Atomicity of the distributed transaction breaks if some of the sub-transactions commit and others abort, even if they the local sub-transaction are atomic by themselves.

For a local sub-transaction that is executed within a single DBMS, the supported isolation level is from a whole spectrum as we have seen in Section 33.3. Even if all participating DBMSs are configured to have the highest isolation level “serializable” for local sub-transactions, e.g., “dirty writes”, which are prohibited even by the lowest isolation level “read uncommitted”, are still possible from the viewpoint of a client managed DDT.

36.1.2 Downsides of DBMS Managed DDTs and Upsides of Client Managed DDTs

On the downside, in DBMS managed DDTs every participating DBMS must be reachable from a central coordinator. This tightens coupling between these services since new dependencies are introduced between their respective and then exposed persistence tiers (databases), which is supposed to be hidden. This becomes practically impossible if the services belong to different organizations just due to security considerations. Moreover, locks are being held for a comparable long time thus hampering performance if two-phase commits are being used.

36.2 DBMS Managed Distributed Database Transactions

In the following we elaborate on how DBMS managed DDTs are usually been implemented.

36.2.1 Two-phase Commit (2PC)

In a DBMS managed DDT, all participating DBMSs have to reach a consensus by means of a certain protocol if a DDT has succeeded and should be committed or aborted. Depending on this consensus, each DBMS can commit or abort its local sub-transactions.

Two-phase commit (2PC)—not to be confused with two-phase locking (2PL)—is such a commit consensus protocol. Normal transactions consist of two steps:

1. Local write locks are acquired, and data is written.
2. A commit record is written, and locks are released again.

2PC breaks step 2 into two steps: Before the actual commit in each database server takes place, each has to conduct a prepare phase. The preparation phase checks if the respective server is really ready to commit afterwards. These “ready to commit” notifications are collected by the coordinator. Only if all servers reply positively, the coordinator orders them to actually commit, otherwise, to abort, and release their locks.

36.2.2 XA Interface for Distributed Transaction Processing

In a DBMS managed DDT, we generally have

- a central coordinator, a. k. a. **transaction manager**, which might be a standalone process or be, as a library, part of a client program, and which has global knowledge of the sub-transaction states,
- the client program itself, a.k.a. **application program** which initiates the transaction and
- the diverse DBMS, a.k.a. **resource managers**, which have just local knowledge and influence on their respective sub-transactions.

This model, as depicted in Figure 163, lays the foundation for a standardized interface written in C, called **XA Interface**, which all the resource managers must expose to the transaction manager in order to participate in a distributed transaction with the 2PC protocol. The XA Interface has therefore defines functions like `xa_open`, `xa_start`, `xa_commit`, `xa_prepare`, `xa_rollback`, and `xa_close` for managing connections and transactions.

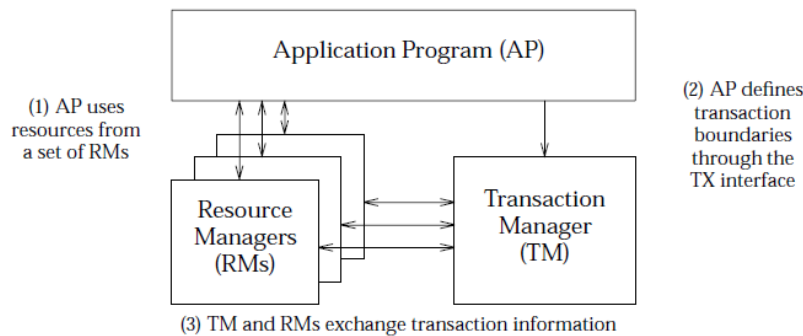


Figure 163: Participants of a distributed transaction according to .

36.2.3 Drawbacks of 2PC and XA Interface

The general downsides of DBMS managed DDTs as described in Section 36.1.2 can be restated regarding 2PC and the XA Interface as follows:

- The additional preparation phase in the 2PC protocol increases the overall time of the distributed transaction to complete.
- The additional preparation phase increases the time that locks on certain database objects are being hold and other transactions have to wait for.
- Some resource managers (DBMSs) might not support a common standard like the XA Interface.

36.3 Client Managed Distributed Database Transactions

In client managed DDTs, the responsibility for atomicity and isolation is left-shifted to the client tier. The toolbox for client managed DDTs is more complicated with more pitfalls than for DBMS managed DDTs. In the following, we show the most common practices and patterns for implementing client managed DDTs mostly as described by .

36.3.1 Commutative Transactions

In a DDT, sub-transaction should be executable in any order, hence be commutative. This is particularly true for sub-transactions that are executed on the same DBMS.

36.3.2 Reread Values

Sub-transactions should only rely on value that have been read in the very same sub-transaction given an appropriate isolation level in the respective DBMS. Values read in other sub-transactions may already be out-of-date because of another transaction on the same DBMS.

36.3.3 Semantic Locks

A DBMS releases locks on database objects when a DDT's sub-transaction commits or aborts. Semantics locks mimic the preparation phase of 2PC as they introduce dedicated attributes to database objects that express the update state of the respective object. A database object that is in the pseudo prepare phase has a state attribute with a value like PENDING or RESERVED. As opposed to real 2PC, this state is visible to clients which must then handle these state as special cases.

36.3.4 Compensation Transactions

A compensation transaction c reverts the effect of a transaction t after t has been committed.

For example, consider a payment service as depicted in Figure 164. Imagine this service as a program that exposes a web API, and which has its own DBMS that manages as single database.

In the upper right corner of this figure, we can see the states of a single payment. The four database snapshots in the lower right corner represent different states of the payment database. This database contains two tables: The upper table has the current balance of a customer and the lower contains a ledger. Of course, we simplify things drastically here as we just assume a single customer.

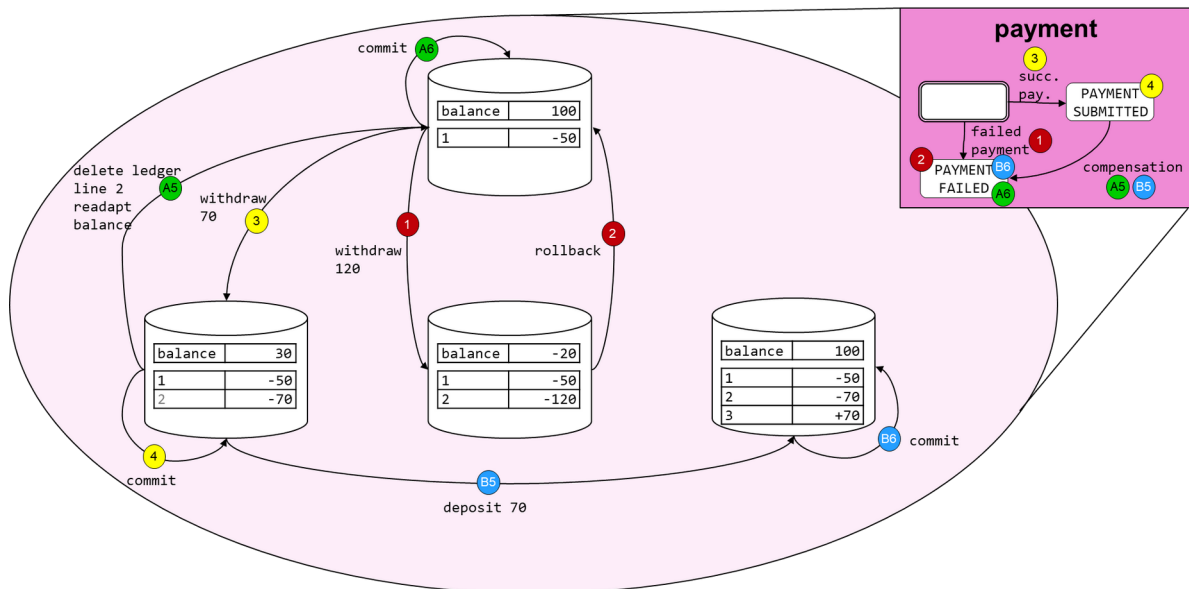


Figure 164: Compensation for a payment service

a) Rollback or Commit

The transitions between the database state tell the following story:

1. We try to withdraw 120 €.
2. This withdrawal violates the internal database constraints, that the balance must not be negative. Therefore, the change is not committed but rolled back
3. Now we try to withdraw 70 €.
4. This leads to a new consistent database state which is then committed.

Such commits and rollbacks (aborts) work well if the payment would be an independent transaction. However, payments normally are part of a distributed transaction.

b) Compensation

Now, if that payment service is part of a distributed system and thus the database part of a distributed database, the committed state 4 sometimes need to be compensated.

Internally with the payment database, this happens in a new transaction, which can be implemented in two ways here:

- a. Undo
 1. We delete the additional ledger line and reset the balance.
 2. The transaction is committed again.
- b. Compensate

1. We can make another database state that has the balance 100 € from before and an additional ledger entry that effectively compensates the previous one.
2. The transaction is committed again.

In itself, a transaction t and its compensation transaction c each may have the property of isolation. Together however they do not form an isolated transaction. After t was committed another transaction t' could see and base on the results of t instead of c . In our example, the payment might have failed because the balance after t was too low but would have been high enough after c .

Therefore, undoing a transaction is often not feasible since a compensation transaction c would not only depend on t but also from unknown transactions t' , t'' , ... which have been committed after t .

36.3.5 Sagas

Of course, a service let alone its database can know by itself when to leave committed database states as-is and when to compensate them. The saga pattern prescribes how services can communicate to also form a sort of distributed transaction.

Sagas avoid the drawbacks of distributed transactions as described in Section 36.2.3. Thereby, they guarantee atomicity of the distributed transaction insofar that all sub-transactions either commit or are aborted or at least compensated. Of course, this is a weaker form of atomicity since compensation does not necessarily revert to the exact former state of the respective database. However, it suffices in most cases. Isolation, however, is not asserted in sagas: If a sub-transaction in a saga commits, other sub-transactions can see its results irrespective of whether other sub-transactions of the saga—potentially targeting the same database—are still in-flight.

In the following we show that there are actually two patterns for implementing sagas.

a) Saga Orchestration

In an orchestrated saga, there is a central “conductor” service like in an orchestra that calls the participating services at the right time to make their respective sub-transactions but also compensation sub-transactions if necessary. Saga orchestration resembles distributed database transaction as described in Section 36.2. However, it does not rely on particular database protocols like 2PC and long-lasting locks on database objects to enforce isolation.

In Figure 165 we can see a checkout process in a microservice architecture, which could be implemented in ReWo-Shop. Again, all the services are very simplified for the sake of brevity. There is a cart service that manages the state of a cart. A cart can either be filled with items or be empty.

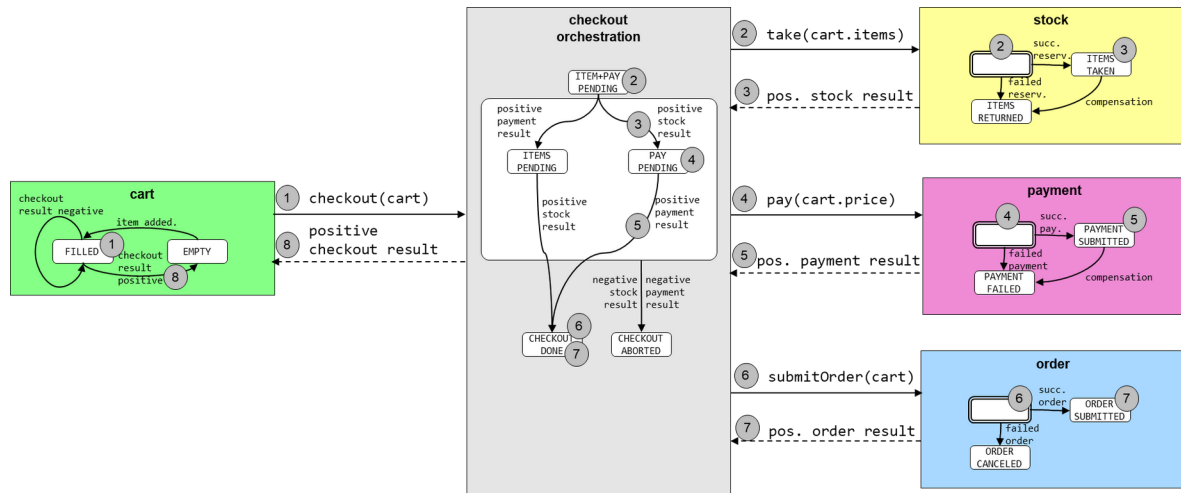


Figure 165: Checkout saga orchestration - normal case

The saga goes as follows:

1. The cart service triggers a checkout by calling a respective web API function on the central conductor service, called “checkout orchestration” here.
2. The checkout orchestration service keeps track of the state of the overall distributed transaction. First, the item removal from stock is still pending and so is the payment. The checkout orchestration now calls the stock service, which manages the inventory of the shop.
3. The stock service makes a sub-transaction, e.g., by lowering the numbers for in-stock items in its database. The success is responded to the checkout orchestration service.
4. Parallel to step 2, the payment service is called.
5. Like described in Section 36.3.4a), the payment service commits a transaction. The success is again responded to the checkout orchestration service.
6. To keep book of orders, an order service is called, which
7. responds positively, too.
8. The overall success is fed back to the cart service which then can purge the cart.

For a comparison, Figure 166 shows the case where one service fails. Here, the payment service cannot withdraw the price. Therefore, the payment sub-transaction is aborted. Since the stock sub-transaction has already been successfully committed, it must be compensated. The order service is not called at all. The orchestration service responds with an error to the cart service, which may therefore leave the items in the cart.

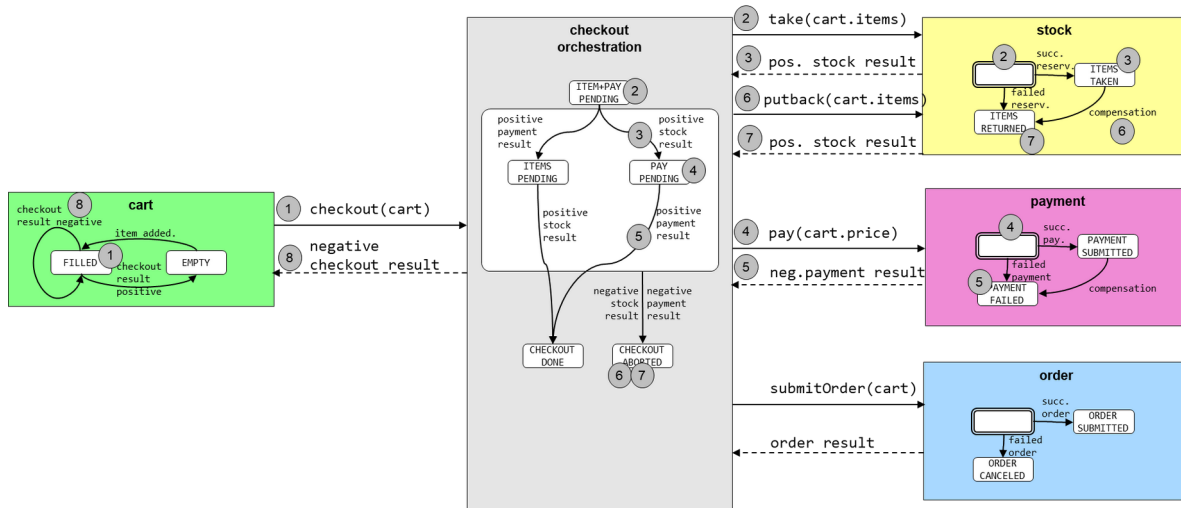


Figure 166: Checkout saga orchestration - payment fails

On the upside, the saga orchestration keeps complicated sagas with many services comparably simple as the overall control flow is centralized in the orchestration service. The participating services also can be agnostic of each other thus leveraging loose coupling. For example, the payment service does not need to know which other services are affected by the failing payment. This knowledge is encapsulated by the checkout orchestration service.

On the downside, the orchestration service poses a single point of failure. If it fails during a saga, atomicity of the overall distributed saga is violated.

b) Saga Choreography

Like in a dance choreography, there is no central conductor in a saga choreography. All participants keep each other informed about their internal events like successful or failed executions of payments, stock item removals and the like.

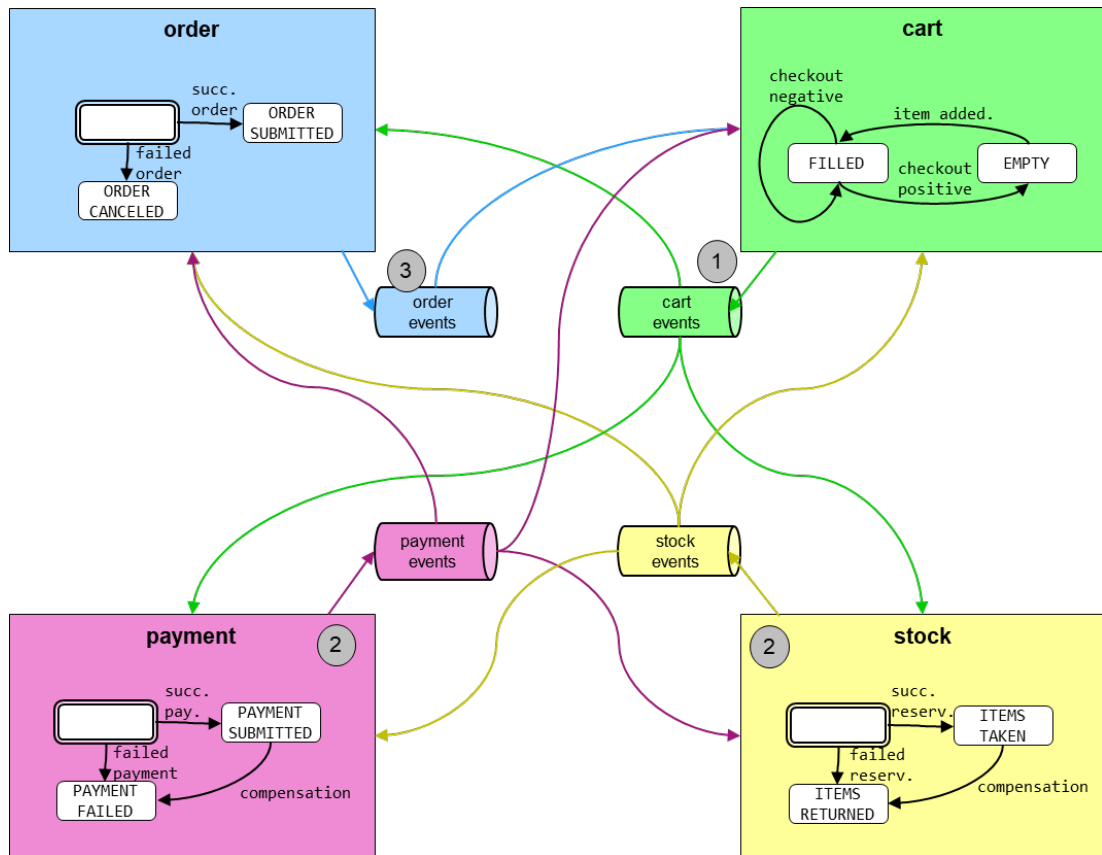


Figure 167: Checkout saga choreography - normal case

Figure 167 depicts the checkout as a saga choreography.

1. The choreography starts with the cart service sending a message to its message queue, which all other services have subscribed to.
2. The payment and the stock service now can proceed in parallel and put their results in their respective message queues, too.
3. Then the payment service collects all the messages and emits a final message, which is only received by the cart service.

Saga choreographies and orchestrations have opposite upsides and downsides: In the saga choreography, there is no central point of failure. For example, even if the order and the cart service have a failure, a payment will still be compensated if just the stock service fails and sends an according message.

On the downside, saga choreographies are usually harder to understand and therefore not endorsed for DDTs with many participants. They also tend to couple their participants tighter to each other. For example, the order service needs to know about the stock and payment service and their respective messages, as the order service may only proceed if positive payment and stock messages arrive. This tighter coupling is a little mitigated by the use of messaging instead of request-response communication.

37 References

- [1] L. Bossavit, *The Leprechauns of Software Engineering*. leanPub, 2015. [Online]. Available: <https://leanpub.com/leprechauns>
- [2] R. C. Martin, “Professionalism and Test-Driven Development,” pp. 32–36, Jun. 2007.
- [3] “Hype Cycle.” Oct. 2019. [Online]. Available: https://en.wikipedia.org/wiki/Hype_cycle
- [4] “Hype Cycle for Application Architecture and Development, 2019.” Oct. 2019. [Online]. Available: <https://www.gartner.com/en/documents/3955980/hype-cycle-for-application-architecture-and-development>
- [5] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, and D. Smith, “The Java® Virtual Machine Specification.” Feb. 2020. [Online]. Available: <https://docs.oracle.com/javase/specs/jvms/se14/html/index.html>
- [6] “LLVM Language Reference Manual.” [Online]. Available: <https://llvm.org/docs/LangRef.html>
- [7] “Compiler Explorer.” [Online]. Available: <https://godbolt.org/>
- [8] “Tool Interface Standard (TIS) Portable Formats Specification.” May 1995. [Online]. Available: <https://refspecs.linuxbase.org/elf/elf.pdf>
- [9] Microsoft, Ed., “PE Format.” Aug. 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/debug/pe-format>
- [10] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts, 9th edition*. Hoboken; NJ 07030: Wiley, 2013.
- [11] “Information technology – Open Systems Interconnection – Basic Reference Model: The basic model.” Jul. 1994. [Online]. Available: [https://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994\(E\).zip](https://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip)
- [12] ISO, “ISO/IEC/IEEE 29148:2018 - Systems and software engineering – Life cycle processes – Requirements engineering,” International Organization for Standardization.
- [13] M. Glinz, “A Risk-Based, Value-Oriented Approach to Quality Requirements,” Mar. 2008.
- [14] E. Evans, *Domain-Driven Design. Tackling Complexity in the Heart of Software*. Addison-Wesley, 2003.
- [15] *ISO/IEC 25010*, Nov. 2023. [Online]. Available: <https://iso25000.com/en/iso-25000-standards/iso-25010>
- [16] “W3C Accessibility Standards Overview.” W3C, Feb. 2024. [Online]. Available: <https://www.w3.org/WAI/standards-guidelines/>

- [17] D. Austin, "2023 Internet Minute Infographic." Apr. 2023. [Online]. Available: <https://ediscoverytoday.com/2023/04/20/2023-internet-minute-infographic-by-ediscovery-today-and-ltmg-ediscovery-trends/>
- [18] B. Gregg, "CPU Utilization is Wrong." May 2017. [Online]. Available: <https://www.brendangregg.com/blog/2017-05-09/cpu-utilization-is-wrong.html>
- [19] "IOPS." Apr. 2019. [Online]. Available: <https://en.wikipedia.org/wiki/IOPS>
- [20] I. Grigorik, *High Performance Browser Networking (1 ed.)*. O'Reilly and Associates, 2013. [Online]. Available: <https://hpbn.co/>
- [21] J. Nagle, "Congestion Control in IP/TCP Internetworks." Jan. 1984. [Online]. Available: <https://tools.ietf.org/html/rfc896>
- [22] "setsockopt function (winsock.h)." Dec. 2018. [Online]. Available: https://docs.microsoft.com/de-de/windows/win32/api/winsock/nf-winsock-setsockopt#remarks::text=TCP_NODELAY
- [23] "tcp(7) - Linux man page." [Online]. Available: https://linux.die.net/man/7/tcp#content::text=provided.-,TCP_NODELAY,If%20set%2C%20disable%20the%20Nagle%20algorithm
- [24] M. Kleppmann, *Designing Data-intensive Applications*. O'Reilly, 2017.
- [25] J. D. C. Little, "A Proof for the Queuing Formula: $L = \lambda W$," 1931. [Online]. Available: <http://fisherp.scripts.mit.edu/wordpress/wp-content/uploads/2015/11/ContentServer.pdf>
- [26] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013.
- [27] "Addressing Cascading Failures," O'Reilly, 2016. [Online]. Available: <https://sre.google/sre-book/addressing-cascading-failures>
- [28] P. Norvig, "Teach Yourself Programming in Ten Years." [Online]. Available: <http://norvig.com/21-days.html>
- [29] B. Brett, "Memory Performance in a Nutshell." Jun. 2016. [Online]. Available: <https://software.intel.com/content/www/us/en/develop/articles/memory-performance-in-a-nutshell.html>
- [30] "How long does it take to make a context switch?" [Online]. Available: <https://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html>
- [31] D. A. Patterson and J. L. Hennesy, *Computer Organisation and Design - The Hardware/Software Interface*, 5th ed. Morgan Kaufmann, 2014.
- [32] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why Do Internet Services Fail, and What Can Be Done About It?," 2003.
- [33] "Twitter thread about human errors." Apr. 2020. [Online]. Available: <https://mobile.twitter.com/ElleArmageddon/status/1255870742727585792>
- [34] N. Nottingham, "HTTP Cache-Control Extensions for Stale Content." May 2010. [Online]. Available: <https://tools.ietf.org/html/rfc5861>

- [35] “ISO/IEC 25012.” International Organization for Standardization, Dec. 2008. [Online]. Available: <https://iso25000.com/index.php/en/iso-25000-standards/iso-25012>
- [36] D. J. Abadi, “Consistency Tradeoffs in Modern Distributed Database System Design,” *IEEE Computer*, vol. 45, no. 2, pp. 37–42, 2012.
- [37] P. Bourgon, “Metrics, tracing, and logging.” Feb. 2017. [Online]. Available: <https://peter.bourgon.org/blog/2017/02/21/metrics-tracing-and-logging.html>
- [38] C. Lilienthal, *Langlebige Software-Architekturen*. dpunkt.verlag GmbH, 2017.
- [39] G. A. Miller, “The magical number seven, plus or minus two: Some limits on our capacity for processing information.,” vol. 2, pp. 81–97, 1956.
- [40] “Microsoft Threat Modeling Tool threats.” Aug. 2017. [Online]. Available: <https://docs.microsoft.com/en-us/azure/security/develop/threat-modeling-tool-threats>
- [41] “44 U.S. Code § 3542 - Definitions.” [Online]. Available: <https://www.govinfo.gov/content/pkg/USCODE-2011-title44/pdf/USCODE-2011-title44-chap35-subchapIII-sec3542.pdf>
- [42] A. van der Stock, B. Glas, N. Smithline, and T. Gigler, “The Open Web Application Security Project (OWASP).” 2017. [Online]. Available: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf
- [43] “Google Cloud Pricing Calculator.” [Online]. Available: <https://cloud.google.com/products/calculator>
- [44] “AWS Pricing Calculator.” [Online]. Available: <https://calculator.aws>
- [45] “Azure Pricing Calculator.” [Online]. Available: <https://azure.microsoft.com/en-gb/pricing/calculator/>
- [46] B. Len, P. Clements, and R. Kazman, *Software Architecture in Practice, 3rd edition*. Addison-Wesley Professional, 2012.
- [47] “User Stories.” [Online]. Available: https://en.wikipedia.org/wiki/User_story
- [48] “Language Server Protocol.” [Online]. Available: <https://langserver.org/>
- [49] M. Page-Jones, “Comparing Techniques by Means of Encapsulation and Connascence,” pp. 147–151, Sep. 1992.
- [50] “connascence.io.” Nov. 2019. [Online]. Available: <https://connascence.io/>
- [51] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*. Prentice Hall, 1994.
- [52] “Cloud Design Patterns.” Mar. 2018. [Online]. Available: <https://docs.microsoft.com/en-us/azure/architecture/patterns/index-patterns>
- [53] M. Fowler, *Patterns of Enterprise Application Architecture*. Pearson Education, 2003.
- [54] G. Hohpe, *Enterprise Integration Patterns, 1st ed.* Addison-Wesley Professional, 2003. [Online]. Available: <https://www.enterpriseintegrationpatterns.com/patterns/messaging/>
- [55] B. Burns, *Designing Distributed Systems*. O’Reilly, 2018.

- [56] C. Richardson, *Microservice Patterns*. Manning Publications Co., 2019.
- [57] C. Richardson, “microservices.io.” [Online]. Available: <https://microservices.io/patterns/>
- [58] M. Richards and N. Ford, *Fundamentals of Software Architecture*. O’Reilly Media, 2020. [Online]. Available: <http://fundamentalsofsoftwarearchitecture.com/>
- [59] P. B. Kruchten, “The 4+1 View Model of architecture,” vol. 12, pp. 42–50, Nov. 1995.
- [60] N. Rozanski and E. Woods, *Software Systems Architecture*. Addison Wesley, 2011. [Online]. Available: <https://www.viewpoints-and-perspectives.info/>
- [61] *Unified Modeling Language 2.5.1*. Object Management Group, 2017. [Online]. Available: <https://www.omg.org/spec/UML/2.5.1/PDF>
- [62] J. Gosling *et al.*, “The Java® Language Specification Java SE 17 Edition.” Aug. 2021. [Online]. Available: <https://docs.oracle.com/javase/specs/jls/se17/jls17.pdf>
- [63] S. Brown, “The C4 Model for Visualizing Software Architecture.” [Online]. Available: <https://c4model.com/>
- [64] Paul Hammant, “Cookie Cutter Scaling,” Paul Hammant’s Blog. [Online]. Available: <https://paulhammant.com/2011/11/29/cookie-cutter-scaling/>
- [65] Google, “Application deployment and testing strategies.” Feb. 2020. [Online]. Available: <https://cloud.google.com/architecture/application-deployment-and-testing-strategies>
- [66] M. Gorman, “Understanding The Linux Virtual Memory Manager.” Jul. 2007. [Online]. Available: <https://www.kernel.org/doc/gorman/pdf/understand.pdf>
- [67] “Regular expression Denial of Service - ReDoS.” [Online]. Available: https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS
- [68] P. Webb and J. S. D. Long, “Spring Boot Reference Guide.” Feb. 2020. [Online]. Available: <https://docs.spring.io/spring-boot/docs/2.3.x/reference/htmlsingle/#server-properties>
- [69] C. Bird, “Conway’s Corollary,” O’Reilly, 2010, pp. 187–205.
- [70] C. Jackson, “Micro Frontends.” Jun. 2019. [Online]. Available: <https://martinfowler.com/articles/micro-frontends.html>
- [71] “Self-Contained Systems.” [Online]. Available: <http://scs-architecture.org/index.html>
- [72] R. Potvin and J. Levenberg, “Why Google Stores Billions of Lines of Code in a Single Repository,” pp. 78–87, Jul. 2016.
- [73] R. C. Martin, *Clean Architecture*. Prentice Hall, 2018.
- [74] “The OpenGroup.” Dec. 1191. [Online]. Available: <https://pubs.opengroup.org/onlinepubs/009680699/toc.pdf>
- [75] M. Fowler, “Microservices and the First Law of Distributed Objects.” Aug. 2014. [Online]. Available: <https://www.martinfowler.com/articles/distributed-objects-microservices.html>

- [76] M. Fowler, “Microservice Trade-Offs.” Jul. 2015. [Online]. Available: <https://martinfowler.com/articles/microservice-trade-offs.html>
- [77] V. Figueira, “Microservices – architecture nihilism in minimalism’s clothes.” Sep. 2020. [Online]. Available: <https://vfig.me/posts/microservices>
- [78] I. Fette and A. Melnikov, “The WebSocket Protocol.” Dec. 2011. [Online]. Available: <https://tools.ietf.org/html/rfc6455>
- [79] S. Loreto, P. Saint-Andre, S. Salsano, and G. Wilkins, “Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP.” Apr. 2011. [Online]. Available: <https://tools.ietf.org/html/rfc6202>
- [80] W. C. Chan and E. Burns, “Java Servlet Specification.” Jul. 2017. [Online]. Available: <https://download.oracle.com/otndocs/jcp/servlet-4-final-eval-spec/index.html>
- [81] D. Kristol and L. Montulli, “HTTP State Management Mechanism.” Oct. 2000. [Online]. Available: <https://tools.ietf.org/html/rfc2965>
- [82] W. Eddy, “Transmission Control Protocol (TCP).” Aug. 2022. [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc9293>
- [83] J. Iyengar and M. Thomson, “QUIC: A UDP-Based Multiplexed and Secure Transport.” May 2021. [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc9000>
- [84] M. Belshe, R. Peon, and M. Thomson, “Hypertext Transfer Protocol Version 2 (HTTP/2).” May 2015. [Online]. Available: <https://tools.ietf.org/html/rfc7540>
- [85] Fielding, R., Reschke, J., and Nottingham, M., *HTTP Semantics*, 9110, Jun. 2022. Accessed: Mar. 27, 2026. [Online]. Available: <https://www.ietf.org/rfc/rfc9110>
- [86] Polli, R. and Martinez, A., *RateLimit Header Fields for HTTP*, May 30, 2021. Accessed: Mar. 27, 2026. [Online]. Available: <https://www.ietf.org/archive/id/draft-polli-ratelimit-headers-05.html>
- [87] “Rate Limiting,” KongDeveloper. Accessed: Mar. 27, 2026. [Online]. Available: <https://developer.konghq.com/plugins/rate-limiting/>
- [88] J. Backus, “Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs,” vol. 21, no. 8, pp. 613–641, Aug. 1978.
- [89] I. Ostrovsky, “Gallery of Processor Cache Effects.” Jan. 2010. [Online]. Available: <http://igoro.com/archive/gallery-of-processor-cache-effects/>
- [90] “Memcached.” Apr. 2019. [Online]. Available: <https://memcached.org/>
- [91] M. Bilal and S. Kang, “Time Aware Least Recent Used (TLRU) cache management policy in ICN,” pp. 528–532, 2014.
- [92] “TypeScript.” [Online]. Available: <https://www.typescriptlang.org/>
- [93] “NodeJS.” [Online]. Available: <https://nodejs.org/en/>
- [94] “React.” [Online]. Available: <https://reactjs.org/>
- [95] “Redis.” Apr. 2019. [Online]. Available: <https://redis.io/>
- [96] “Apache ActiveMQ.” [Online]. Available: <http://activemq.apache.org/>

- [97] Lewis, E., "The Role of Wildcards in the Domain Name System." Network Working Group, Jun. 2006. [Online]. Available: <https://www.rfc-editor.org/rfc/rfc4592>
- [98] "List of DNS record types." Apr. 2019. [Online]. Available: https://en.wikipedia.org/wiki/List_of_DNS_record_types
- [99] R. Fielding, N. Nottingham, and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Caching." Jun. 2014. [Online]. Available: <https://tools.ietf.org/html/rfc7234>
- [100] P. Mell, Grance, and Tim, "The NIST Definition of Cloud Computing." Sep. 2011. [Online]. Available: <https://doi.org/10.6028/NIST.SP.800-145>
- [101] "Heroku.com." [Online]. Available: <https://heroku.com>
- [102] "Google App Engine." Dec. 2019. [Online]. Available: <https://cloud.google.com/appengine/>
- [103] "Google Compute Engine." Dec. 2019. [Online]. Available: <https://cloud.google.com/compute/>
- [104] "Bin packing problem." Apr. 2019. [Online]. Available: https://en.wikipedia.org/wiki/Bin_packing_problem
- [105] "Public Cloud Services Comparison." Mar. 2019. [Online]. Available: <http://comparecloud.in/>
- [106] "AI winter." [Online]. Available: https://en.wikipedia.org/wiki/AI_winter
- [107] "18 U.S. Code § 2713 - Required preservation and disclosure of communications and records." Mar. 2018. [Online]. Available: <https://www.law.cornell.edu/uscode/text/18/2713>
- [108] F. Richter, "The Big Three Stay Ahead in Ever-Growing Cloud Market." Aug. 2025. [Online]. Available: https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/?srsltid=AfmBOorcp1zrAvob_xUMonah9brZ0Xyg1K5WXcj-kO1oLmot4wN3lamp
- [109] "CNCF Cloud Native Definition v1.1." Feb. 2024. [Online]. Available: <https://github.com/cncf/toc/blob/master/DEFINITION.md>
- [110] S. Rose, O. Borchert, S. Mitchell, and S. Conelly, "Zero Trust Architecture." National Institute of Standards and Technology, Feb. 2020.
- [111] R. Ward and B. Beyer, "BeyondCorp - A New Approach to Enterprise Security," vol. 39, no. 6, Dec. 2014.
- [112] "BeyondProd: A new approach to cloud-native security." Apr. 2020. [Online]. Available: <https://cloud.google.com/security/beyondprod>
- [113] "Google Cloud Platform Documentation." Apr. 2019. [Online]. Available: <https://cloud.google.com/docs/>
- [114] "Google Cloud Platform Tutorials." Apr. 2019. [Online]. Available: <https://cloud.google.com/docs/tutorials>
- [115] "Cloud SDK - gcloud reference." Apr. 2019. [Online]. Available: <https://cloud.google.com/sdk/gcloud/reference/>

- [116] A. M. Joy, “Performance comparison between Linux containers and virtual machines,” Ghaziabad, India, 2015.
- [117] “CVE-2016-5195.” Oct. 2016. [Online]. Available: <https://access.redhat.com/security/cve/cve-2016-5195>
- [118] “Namespace manpage.” Mar. 2019. [Online]. Available: <http://man7.org/linux/man-pages/man7/namespaces.7.html>
- [119] “cgroups manpage.” Jun. 2019. [Online]. Available: <http://man7.org/linux/man-pages/man7/cgroups.7.html>
- [120] “OCI Image Format Specification.” [Online]. Available: <https://github.com/opencontainers/image-spec>
- [121] “containerd.” Jun. 2019. [Online]. Available: <https://containerd.io>
- [122] “Kubernetes.” Jun. 2019. [Online]. Available: <https://kubernetes.io>
- [123] A. Kammer, C. Koppelt, J. Müller, H. Prinz, C. Schmidt, and E. Wolff, “servicemesh.es.” [Online]. Available: <https://servicemesh.es/>
- [124] “Apache Kafka.” Apr. 2019. [Online]. Available: <https://kafka.apache.org/>
- [125] A. Jacobs, “The Pathologies of Big Data,” pp. 36–44, 2009.
- [126] T. Berners-Lee, R. Fielding, and L. Masinter, “Uniform Resource Identifier (URI): Generic Syntax.” Jan. 2005. [Online]. Available: <https://tools.ietf.org/html/rfc3986>
- [127] L. Dusseault and J. Snell, “PATCH Method for HTTP.” Mar. 2010. [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc5789>
- [128] I. Grigorik, “HTTP Caching.” [Online]. Available: <https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/http-caching>
- [129] R. Fielding and J. Reschke, “Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing.” Jun. 2014. [Online]. Available: <https://tools.ietf.org/html/rfc7230>
- [130] “ChatGPT.” [Online]. Available: <https://chat.openai.com/>
- [131] I. Hickson, “Server-Sent Events.” Feb. 2015. [Online]. Available: <https://www.w3.org/TR/eventsource/>
- [132] “Common Object Request Broker: Core Specification.” Mar. 2004. [Online]. Available: <http://www.omg.org/cgi-bin/doc?formal/04-03-12.pdf>
- [133] “Protocol Buffers.” [Online]. Available: <https://developers.google.com/protocol-buffers>
- [134] “WebSocket Protocol Registries.” Internet Assigned Numbers Authority (IANA), Aug. 2024. [Online]. Available: <https://www.iana.org/assignments/websocket/websocket.xml>
- [135] I. Hickson, “The WebSocket API.” Sep. 2012. [Online]. Available: <https://www.w3.org/TR/websockets/>

- [136] “Spring WebSocket Support.” [Online]. Available: <https://docs.spring.io/spring/docs/5.0.0.BUILD-SNAPSHOT/spring-framework-reference/html/websocket.html>
- [137] S. Ghasemshirazi and P. Heydarabadi, “Exploring the Attack Surface of WebSocket.” [Online]. Available: <https://arxiv.org/abs/2104.05324>
- [138] R. Fielding, *REST: Architectural Styles and the Design of Network-based Software*. University of California, Irvine, 2000.
- [139] M. Kelly, “HAL - Hypertext Application Language.” Jul. 2015. [Online]. Available: <https://tools.ietf.org/html/draft-kelly-json-hal-07>
- [140] M. Amundsen, “The HAL-FORMS Media Type.” Oct. 2019. [Online]. Available: <https://rwcbook.github.io/hal-forms/>
- [141] L. Richardson, “Richardson Maturity Model.” 2009. [Online]. Available: <https://www.crummy.com/writing/speaking/2008-QCon/act3.html>
- [142] M. Fowler, “Richardson Maturity Model - steps toward the glory of REST.” Mar. 2010. [Online]. Available: <https://martinfowler.com/articles/richardsonMaturityModel.html>
- [143] “OpenAPI Specification.” [Online]. Available: <https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.2.md>
- [144] “OpenAPI Specification - Petstore Example.” [Online]. Available: <https://github.com/OAI/OpenAPI-Specification/blob/master/examples/v3.0/petstore.yaml>
- [145] “GraphQL - Specification.” Jun. 2018. [Online]. Available: <https://graphql.github.io/graphql-spec/June2018/>
- [146] “JSON-RPC 2.0 Specification.” JSON-RPC Working Group, Jan. 2013. [Online]. Available: <https://www.jsonrpc.org/specification>
- [147] “JSON Schema Specification.” Open Collective, Jun. 2022. [Online]. Available: <https://json-schema.org/specification>
- [148] E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.3.” Aug. 2018. [Online]. Available: <https://tools.ietf.org/html/rfc8446>
- [149] D. Eastlake 3rd and T. Hansen, “US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF).” May 2011. [Online]. Available: <https://tools.ietf.org/html/rfc6234>
- [150] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, Housley, and W. R. Polk, “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.” May 2008. [Online]. Available: <https://tools.ietf.org/html/rfc5280>
- [151] “Abstract Syntax Notation One (ASN.1) Recommendations.” [Online]. Available: <https://www.itu.int/ITU-T/studygroups/com17/languages/>
- [152] C. Newman, “Using TLS with IMAP, POP3 and ACAP.” Jun. 1999. [Online]. Available: <https://tools.ietf.org/html/rfc2595>
- [153] “Usage statistics of Default protocol https for websites.” [Online]. Available: <https://w3techs.com/technologies/details/ce-httpsdefault>

- [154] Z. A. Bahajji and G. Illyes, “HTTPS as a ranking signal.” Aug. 2014. [Online]. Available: <https://developers.google.com/search/blog/2014/08/https-as-ranking-signal>
- [155] A. Langley, M. Hamburg, and S. Turner, “Elliptic Curves for Security.” Jan. 2016. [Online]. Available: <https://tools.ietf.org/html/rfc7748>
- [156] T. School, “A complete overview of SSL/TLS and its cryptographic system.” Apr. 2020. [Online]. Available: <https://dev.to/techschoolguru/a-complete-overview-of-ssl-tls-and-its-cryptographic-system-36pd>
- [157] M. Driscoll, “The New Illustrated TLS Connection.” [Online]. Available: <https://tls13.ulfheim.net/>
- [158] D. (Ed) Hardt, “The OAuth 2.0 Authorization Framework.” Oct. 2012. [Online]. Available: <https://tools.ietf.org/html/rfc6749>
- [159] J. Richter, “OAuth 2.0 Token Introspection.” Oct. 2015. [Online]. Available: <https://tools.ietf.org/html/rfc7662>
- [160] M. Jones, J. Bradley, and N. Sakimura, “JSON Web Signature (JWS).” May 2015. [Online]. Available: <https://tools.ietf.org/html/rfc7515>
- [161] N. Sakimura, J. Bradley, M. B. Jones, and E. Jay, “OpenID Connect Discovery 1.0 incorporating errata set 1.” Nov. 2014. [Online]. Available: https://openid.net/specs/openid-connect-discovery-1_0.html
- [162] J. Franks *et al.*, “HTTP Authentication: Basic and Digest Access Authentication.” Jun. 1999. [Online]. Available: <https://tools.ietf.org/html/rfc2617>
- [163] M. Jones, “JSON Web Key (JWK).” May 2015. [Online]. Available: <https://tools.ietf.org/html/rfc7517>
- [164] N. Sakimura, J. Bradley, M. B. Jones, B. de Medeiros, and C. Mortimore, “OpenID Connect Core 1.0 specification.” Nov. 2014. [Online]. Available: https://openid.net/specs/openid-connect-core-1_0.html
- [165] T. Haynes and D. Novek, “Network File System (NFS) Version 4 Protocol.” [Online]. Available: <https://tools.ietf.org/html/rfc7530>