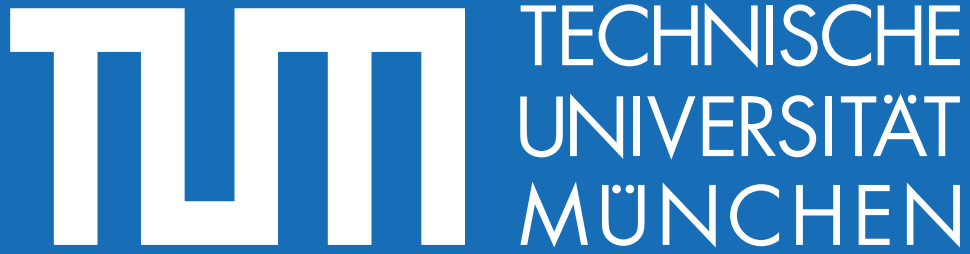


Software Engineering in der industriellen Praxis (SEIP)

Dr. Ralf S. Engelschall

ARCHITECTURE FUNDAMENTALS

Slice Architectures

(Patterns)

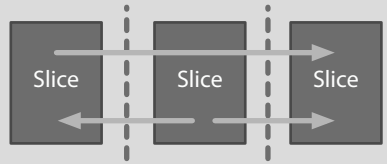
TECHNISCHE UNIVERSITÄT MÜNCHEN

AF 06.3

Slicing Principle

Vertically split code or data into two or more logically, optionally also spatially, clearly distinct, named, and unranked slices.

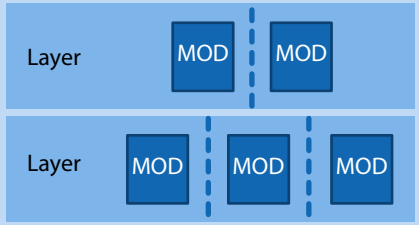
The particular slicing should minimize the total amount of individual relationships between the resulting slices. Per type of relationship, there should be no cycle in the transitive relationships.



MOD Concerned Module

Split related code or data (usually across a single Layer) into two or more logically distinct domain- or technology-induced Modules.

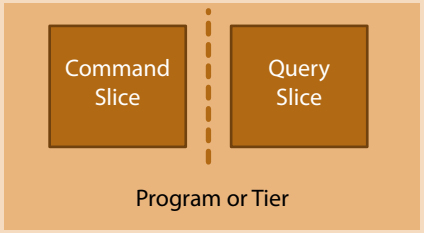
Rationale: Separation of Concern, Single Responsibility Principle, Mastering Complexity.



CQRS Command-Query Responsibility Segregation

Split code and data of a Program (across all Layers) or a Tier into exactly two slices to segregate operations that read data (queries) from the operations that update data (commands).

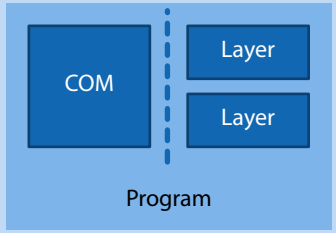
Rationale: Separated Scalability, Separated Data Access Patterns, Event Sourcing Approach.



COM Common Slice

Factor out common or cross concern code or data of a Program (across all Layers) into a single spatially distinct, separate slice.

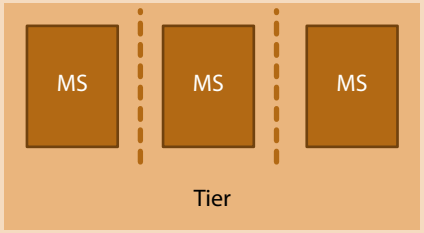
Rationale: Lack of Redundancy, Single Point of Truth, Reusability.



MS Microservice

Split code and data of a Tier (across all Layers) into two or more distinct, loosely-coupled, domain-enclosed, functional services, each forming a stand-alone Program.

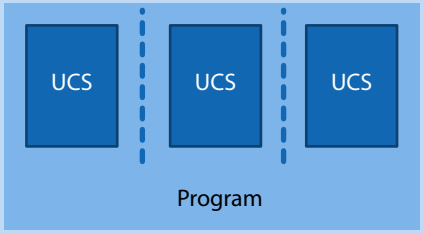
Rationale: Heterogeneity, Resilience, (Scalability), (Easy Deployment), (Organizational Alignment), (Composability), (Reusability), Replaceability.



UCS Use-Case Slice

Split the code and data of a Program (across all Layers) into two or more purely logical slices, one for each distinct, domain-specific Use-Case.

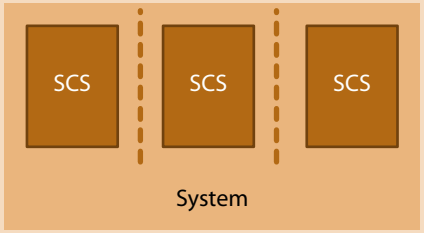
Rationale: Comprehensibility, Domain Alignment, Mastering Complexity.



SCS Self-Contained System

Split code and data of a System (across all Layers and Tiers) into two or more distinct, loosely-coupled, domain-enclosed, functional systems, each forming a stand-alone sub-System.

Rationale: Mastering Complexity, Heterogeneity, Resilience, Scalability, Easy Deployment, Organizational Alignment, Reusability, Replaceability.



ARCHITECTURE FUNDAMENTALS

Flow Architectures

(Patterns)

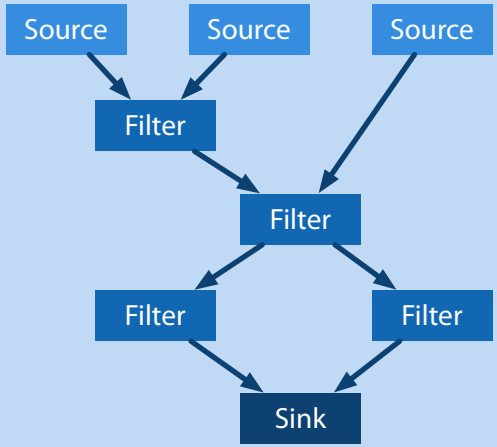
TECHNISCHE UNIVERSITÄT MÜNCHEN

AF 07.1

Pipes & Filters

Pass data through a directed graph of **Components** and connecting **Pipes**. The components can be **Sources**, where data is produced, **Filters**, where data is processed, or **Sinks**, where data is captured. Source and Filter components can have one or more output Pipes. Filter and Sink components can have one or more input Pipes. Components are independent processing units and operate fully asynchronously.

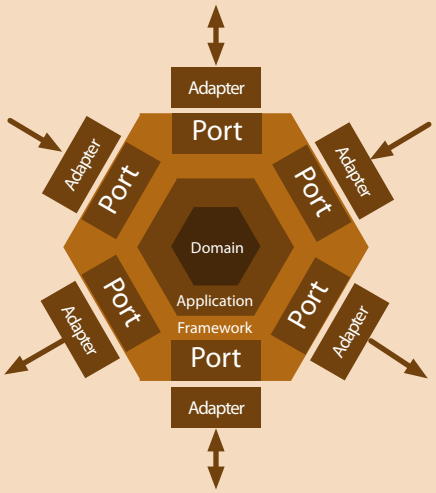
Examples: Unix commands with stdin/stdout/stderr and the Unix shell connecting them with pipes; Apache Spark or Apache Camel data stream processing pipelines.



Ports & Adapters (Hexagonal)

Perform communication in a Hub & Spoke fashion by structuring a solution into the three "Layers" **Domain**, **Application** and **Framework** and use the Framework layer to connect with the outside through **Ports** (general Interfaces) and **Adapters** (particular Implementations). Often some Ports & Adapters are user-facing sources and some are data-facing sinks, although the motivation for the Ports & Adapters architecture is to remove this distinction between user and data sides of a solution.

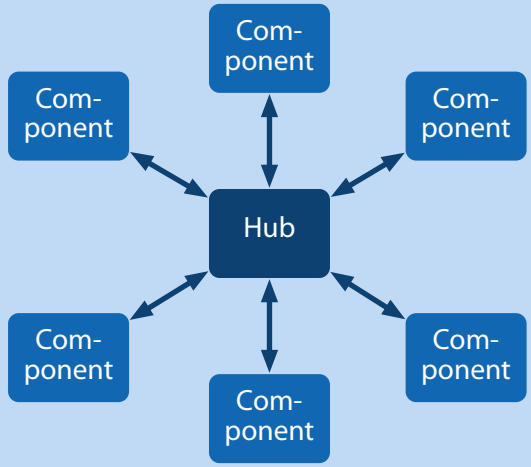
Examples: Message Queue, Enterprise Service Bus or Media Streaming Service internal realization.



Hub & Spoke

Perform communication (the **Spoke**) between multiple Components through a central **Hub** Component. Instead of having to communicate with N x (N-1) / 2 bi-directional interconnects between N Components, use the intermediate Hub to communicate with just N interconnects only. Sometimes one distinguishes between K (0 < K < N) source and N - K target Components and then K x (N - K) uni-directional interconnects are reduced to just N interconnects, too.

Examples: Message Queue, Enterprise Service Bus, Module Group Facade, GNU Compiler Collection, ImageMagick, etc.



ARCHITECTURE FUNDAMENTALS

Process Architectures

(Patterns)

TECHNISCHE UNIVERSITÄT MÜNCHEN

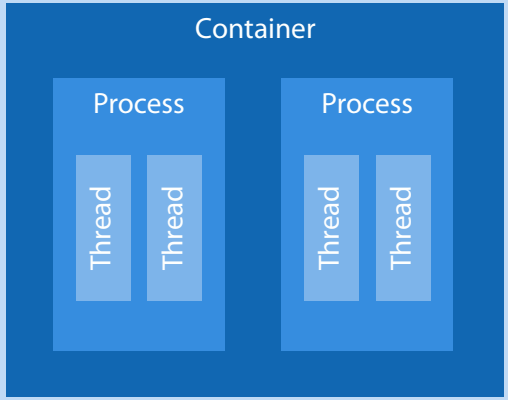
AF 07.2

Container, Process, Thread

The Operating System manages and orchestrates the run-time execution of applications in **Containers**, programs in **Processes** and control flows in **Threads**.

Containers are the ultimate enclosures, separating and controlling both the computing resources processor, memory, storage and network. Processes are the primary enclosures, still separating and controlling at least the computing resources processor and memory. Threads are the light-weight enclosures, just separating and controlling the computing resource processor. Containers can contain one or more Processes, and Processes can contain one or more Threads.

Examples: Docker Container, Unix Processes, POSIX Threads.

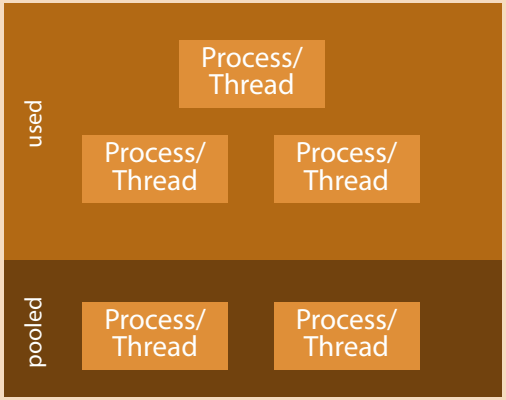


Process/Thread Pool

Instead of creating a Process/Thread for handling each incoming I/O request, pick a pre-created Process/Thread out of a resource **Pool** in order to increase performance and decouple I/O traffic (leading to threads of execution) from the actual computing resource usage and utilization.

The Process/Thread Pool usually has a lower and upper bound of processes/threads. The lower bound keeps the system "hot" between I/O requests. The upper bound limits the computing resource usage and avoids over-utilization.

Examples: Apache HTTP Daemon

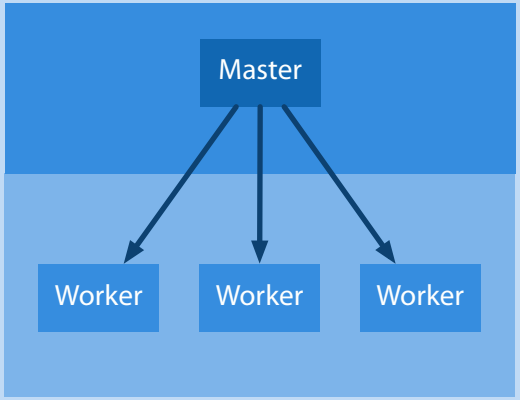


Master-Worker

The system has a single permanent **Master** container/process/thread and a Pool of many ephemeral **Worker** containers/processes/threads. The Master starts, restarts, pauses, resumes and stops the Workers and usually also delegates incoming I/O requests to them. The Workers process the I/O requests and deliver the responses.

Starting the Master usually implicitly starts an initial set of Workers (the initial Pool), stopping the Master implicitly stops all still pending Workers.

Examples: Unix init(8) daemon, Apache HTTP Daemon, SupervisorD, Node.js Cluster module



ARCHITECTURE FUNDAMENTALS

Cluster Architectures

(Patterns)

TECHNISCHE UNIVERSITÄT MÜNCHEN

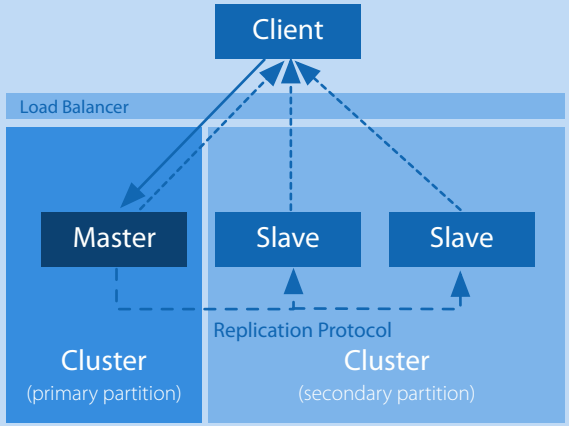
AF 07.3

Master-Slave (Static Replication)

Cluster of a single **Master** and multiple **Slave** nodes, where data is continuously copied from the Master to the Slave nodes in order to support high-availability (where a Slave will take over the Master role) in case of a Master outage and increased read performance (where regular read requests are also served by the Slaves).

In this static replication scenario the Master is usually assigned statically and in case of outages has to be reassigned usually semi-manually. Especially, the full reestablishment of the original Master assignment after a Master recovery usually is a manual process.

Examples: OpenLDAP Replication, PostgreSQL WAL Replication.

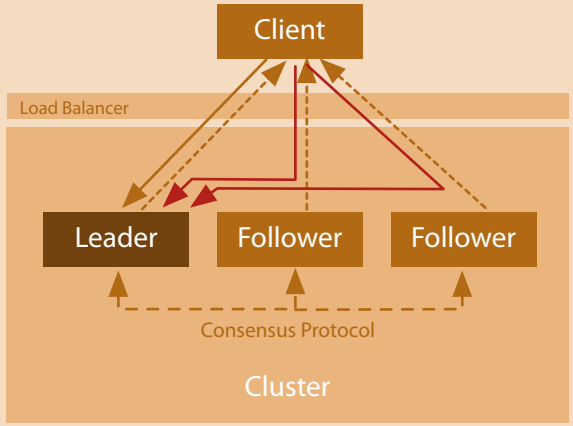


Leader-Follower (Dynamic Replication)

Cluster of a single **Leader** and multiple **Follower** nodes, where data is written on the current Leader node and data read on both the current Leader and all Follower nodes. For writing data to the cluster, the Leader node performs a consensus protocol (e.g. RAFT, Paxos or at least Two-Phase-Commit) with the Followers and this way automatically and consistently replicates the data to the Followers.

In this dynamic replication scenario the Leader is usually automatically assigned by the cluster nodes through an election protocol and in case of outages is automatically re-assigned. There is usually no re-establishment of the original Leader assignment.

Examples: Apache Zookeeper, Consul, EtcD, CockroachDB, InfluxDB.

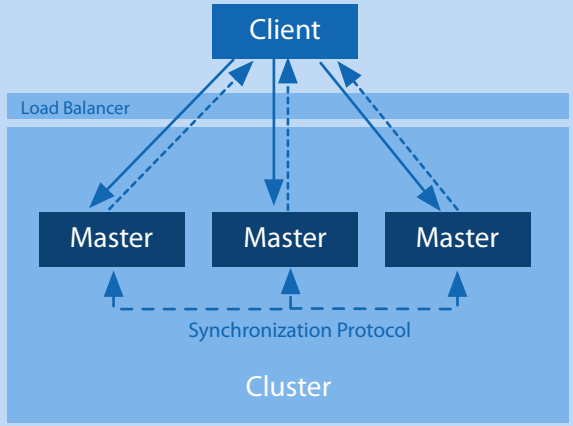


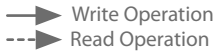
Master-Master (Synchronization)

Cluster of multiple **Master** nodes, where data is read and written on any Master node concurrently. The Master nodes either use Strict Consistency through writing to a mutual-exclusion-locked shared storage concurrently or use Eventual Consistency in a Shared Nothing storage scenario where they continuously synchronize their local data state to all other nodes with the help of a synchronization protocol.

The synchronization protocol usually is based on either Conflict-Free Replicated Data Types (CRDT) or at least Operational Transformation (OT). In any scenario, data update conflicts are explicitly avoided.

Examples: ORACLE RAC, MySQL/MariaDB Galera Cluster, Riak, Automerge/Hypermerge.





ARCHITECTURE FUNDAMENTALS

Networking Architectures

(Patterns)

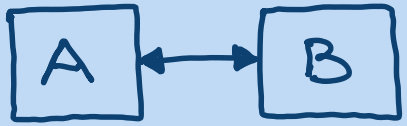
TECHNISCHE UNIVERSITÄT MÜNCHEN

AF 08.1

PTP Point-to-Point

Communicate between two network nodes in a point-to-point fashion, usually through a direct link.

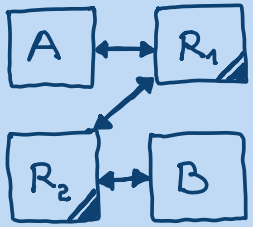
Rationale: simple communication where both nodes know about each other and can directly reach each other.



RTG Routing

Communicate between two network nodes in a point-to-point fashion, but by routing the network packets over intermediate forwarding nodes (routers).

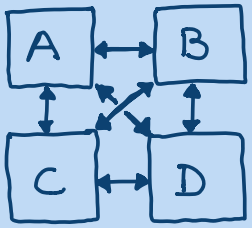
Rationale: simple communication where both nodes know about each other, but cannot directly reach each other.



P2P Peer-to-Peer

Communicate between multiple network nodes (usually all in the client and server role at the same time) without involving a central hub node (in the role of a server) — except for the initial network entry discovery.

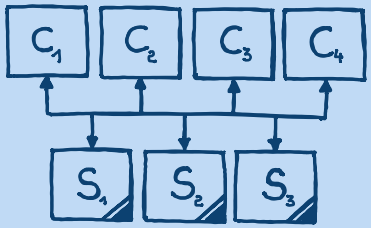
Rationale: communication without central control (although a seed peer is required).



C/S Client/Server

Communicate between multiple nodes in the client role (making requests, and usually with ephemeral addresses) and multiple nodes in the server role (serving responses, and usually with fixed addresses).

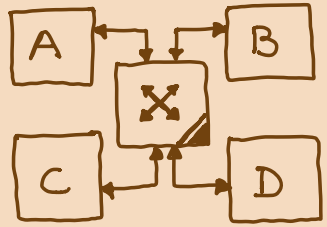
Rationale: communication with central orchestration, control and data storage.



BUS Bus/Broker/Relay

Communicate between multiple nodes with the help of a central packet forwarding hub node in a star network topology.

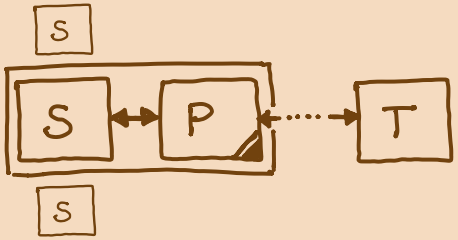
Rationale: decouple communication nodes: instead of Point-to-Point (PTP) communications between all nodes, there are just PTP communications with the hub.



FPR (Forward) Proxy

Communicate between two nodes by using an intermediate forwarding proxy node in front of the source node.

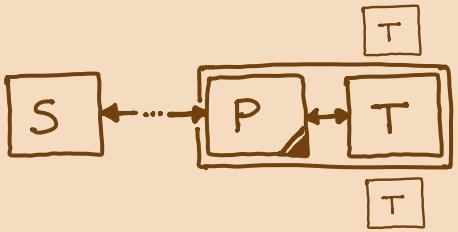
Rationale: bridge network topology constraints (segmented networks); caching at source side; auditing of communication.



RPR Reverse Proxy

Communicate between a source and a target node by using a masquerading proxy node directly in front of the target node.

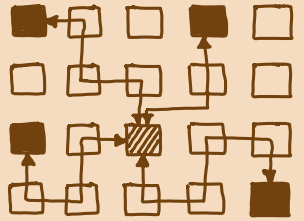
Rationale: load balancing for multiple target nodes; caching at target side; auditing of communication; security shielding of target nodes; protocol conversions.



VPN Virtual (Private) Network

Communicate between nodes in a logical star network topology on top of an arbitrary physical routed network topology.

Rationale: secure private network overlaying an unsecure public network; simplify network topology.



ARCHITECTURE FUNDAMENTALS

Communication Architectures

(Patterns)

TECHNISCHE UNIVERSITÄT MÜNCHEN

AF 08.2

UCT Unicast (one-to-one)

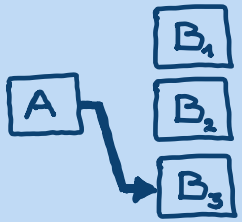
Communicate messages from one source to exactly one destination node. The destination node is explicitly and individually addressed.

Rationale: private communication between exactly two nodes which both know each other beforehand.

ACT Anycast (one-to-any)

Communicate messages from one source to one of many destination nodes. The picked destination node usually is the network-topology-wise "nearest" or least utilized node in a group of nodes.

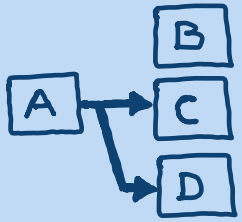
Rationale: Unicast, optimized for network failover scenarios, load balancing and CDNs.



MCT Multicast (one-to-many)

Communicate messages from one source to many destination nodes. The destination nodes usually form a group and are usually not individually addressed.

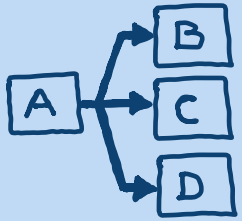
Rationale: node communication where destination nodes dynamically change or where total traffic should be reduced.



BCT Broadcast (one-to-all)

Communicate messages from one source to all available destination nodes. The destination nodes usually are implicitly defined by the extend of the local communication network segment.

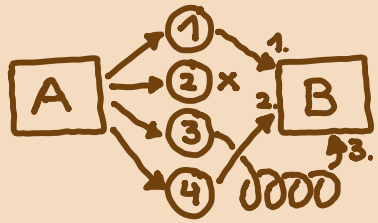
Rationale: spreading out messages to all available nodes for potential responses.



DGR Datagram (Single Packet)

Communicate messages as an unordered set of single packets, usually without any network congestion control, retries or other delivery guarantees.

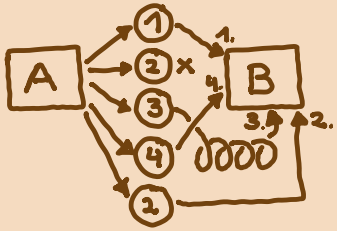
Rationale: simple low-overhead communication without prior communication establishment (handshake).



STR Stream (Sequence of Packets)

Communicate messages as an ordered sequence (stream) of packets, usually with network congestion control, retries and delivery guarantees (at-most-once, exactly-once, at-least-once).

Rationale: reliable communication between nodes.



PLL Pull (Request/Response, RPC)

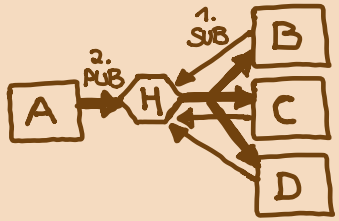
Communicate by performing a request (from the client node) and pulling a corresponding response (from the server node).

Rationale: Remote Procedure Call (RPC) like Unicast or Anycast communication.

PSH Push (Publish/Subscribe, Events)

Communicate by "subscribing" to "channels" of messages (on one or more receiver nodes or on an intermediate hub) once and then publishing events to those "channels" (on the sender node) multiple times.

Rationale: event-based Multicast or Broadcast communication.



ARCHITECTURE FUNDAMENTALS

Data Structure Architectures

(Patterns)

TECHNISCHE UNIVERSITÄT MÜNCHEN

AF 09.1

Data Structure Types

Scalar, Atom, Primitive Type

Plain integer or real number, single character or character string, not indexed and (for string only) accessed in O(1) by character position.

Tuple, Object, Structural Type, Record

Ordered, fixed-size sequence of scalar elements, each of individual type, indexed by name and accessed in O(1) by element name.

Sequence, Array, List

Ordered sequence of elements, each of same type, indexed by position and accessed in O(1) or O(n) by element position.

Set, Bag, Bucket

Unordered set of elements, each of same type, not indexed and accessed in O(1) or O(n) by element reference.

Map, Hash, Associative Array

Unordered sequence of elements, each of same type, indexed by (scalar) key and accessed in O(1) by key.

Graph, Nodes & Edges

Unordered set of linked elements (nodes), each of individual type, indexed by (scalar) key and accessed in O(1) by key or by following a directed link (edge).

Data Evolution Approaches

In-Place Editing

Modify data through direct in-place editing, overwriting the previous revision.

Stacking Revisions

Modify data through stacking revisions, preserving all previous revisions. Latest revision is always on top of stack.

Structural Difference

Modify data through merging, journaled domain-unspecific structural differences.

Operational Transformation (OT)

Modify data through applying journaled, domain-specific operational transformations.

Event Sourcing & CRDT

Share data as a chronological sequence of data change events from which the data states can be (re)constructed. Optionally, use a Conflict-Free Replicated Data-Type (CRDT) protocol for the change events.

Ref.-Counting & Copy-on-Write

Share data between resources by using reference-counted data chunks, duplicating a chunk (and resetting its reference count to one) on write operations only and destroying a chunk once the reference count drops to zero.

Data Store Types

Key-Value Store

Storage of values in an unordered manner, indexed and queried by key.

Redis, Riak, MemcacheD, RocksDB, LevelDB

Large-Object Store

Storage of unstructured binary-large object (BLOB) data and its associated meta-data, indexed and queried by unique id.

Minio, SeaweedFS, AWS S3

Triple Store

Storage of subject-predicate-object triples, indexed and queried by subject/predicate/object values and example triples.

Redstore, Virtuoso

File-Tree Store

Storage of unstructured data as named files in a directory tree, indexed and queried by name path from root directory to leave file.

ZFS, XFS, UFS2, APFS

Graph Store

Storage of values as vertices and edges in a graph, both optionally referencing associated key/value pairs. Indexed and queried by key/value pairs and traversed by following edges.

Neo4J, OrientDB, ArangoDB

Document Store

Storage of structured "documents", indexed by id and key/value fields and queried by id and example documents.

MongoDB, CouchDB, RethinkDB

Relational/Table Store

Storage rows of fixed-size, typed value columns, indexed and queried by column values.

PostgreSQL, MariaDB, SQLite, H2, ORACLE DB, IBM DB2

Full-Text Store

Storage of unstructured text, indexed and queried by content words.

ElasticSearch, Solr, Groonga

Wide-Column Store

Distributed storage of rows of sparse (often untyped) value columns, indexed and queried by column values.

Cassandra, MonetDB, HBase, ScyllaDB

Time-Series Store

InfluxDB, MetricTank, RRDTool

Storage of integer or real values (y-axis) of a time-series (x-axis) into a fixed-size storage format in a round-robin manner where older values are increasingly aggregated (leading to lower resolutions at older times) and finally overwritten.

DataVault Store

Long-term historical storage of foreign, arbitrary relational data in a fixed schema of hubs, links and satellites, indexed and queried for analysis and reporting purposes.

DataVault 2.0

BlockChain Store

Storage of values in an unordered manner within information blocks which are cryptographically chained through their hash values and distributed in a peer-to-peer way.

Ethereum, Quorum, Tendermint, Hyperledger

ARCHITECTURE FUNDAMENTALS

Data Persistence Architectures

(Patterns)

TECHNISCHE UNIVERSITÄT MÜNCHEN

AF 09.2

Data Guarantees

CAP (Trade-In)

A distributed data store cannot provide more than two out of three guarantees: Consistency (C), Availability (A), Partition-Tolerance (P). So, it has to choose between Consistency (CP) and Availability (AP) when a network partition or failure happens.

BASE (NoSQL)

The semantics (usually of NoSQL systems) of (B)asically (A)vailable, (S)oft state, and (E)ventual consistency. BASE systems favor Availability over Consistency in the CAP-context.

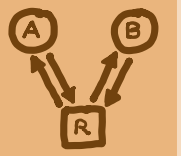
ACID (RDBMS, NewSQL)

The four guarantees provided in parallel (usually by RDBMS and NewSQL systems): Atomicity, Consistency, Isolation and Durability. ACID systems usually favor Consistency over Availability in the CAP-context.

Data Access

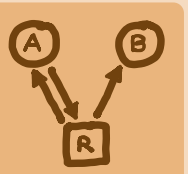
Shared Read/Write

Shared access to data for both read and write operations. Example: Multiple threads on heap or Master-Master database setup.



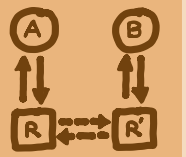
Shared Read / Exclusive Write

Shared access to data for read operations and exclusive access (via a single "owning" component) to data for write operations. Example: RDBMS Master-Slave cluster with shared storage.



Shared Nothing

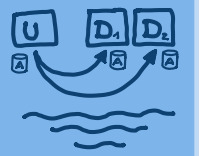
No shared access to data at all for both read and write operations. Example: Leader-Follower setup with RAFT consensus where Leader writes data only.



Data Spreading & Aggregation

Data River (1-to-N)

A real-time fan-out replication of data from a single upstream/source data repository to multiple downstream/target data repositories.



Data Mart (N-to-1), ODS

A massive sized, easily accessible data repository for storing "big data" from many upstream sources in a (real-time and) structured way and with knowing the actual subsequent analysis usage.

Data Lake (N-to-1), Cache

A massive sized, easily accessible data repository for storing "big data" from many upstream sources in a (real-time) semi-structured way and without knowing the actual subsequent usage.

Data Access Grouping

Transaction

Protect a sequence of operations from interim exceptions by bracketing the operations in a technical transaction (ensuring that either all or none of the operations succeed).

Compensation

Protect a sequence of operations from interim exceptions by undoing the already succeeded operations through domain-specific compensating (reverse) operations.

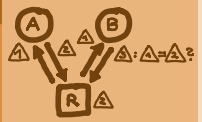
Data Consistency

Exclusive Locking (Mutex)

Protect data from concurrent access and resulting inconsistencies with a mutual exclusion lock (mutex) which allows just a single peer to access the data at a time.

Optimistic Locking

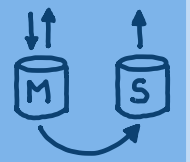
Protect data from concurrent access and resulting inconsistencies by taking note of a revision number or content hash during read operations and checking that this information has not changed before writing the data.



Data Transfer

Replication

Continuously stream or regularly copy data from a master system to one or more slave systems in order to either read the data from slave systems faster or have slave systems available as a fallback/ backup in case of a failure of the master system.



Synchronization

Continuously stream or regularly copy data between multiple master systems and resolve potential concurrent data modification conflicts. This way allow distributed and even disconnected computing.