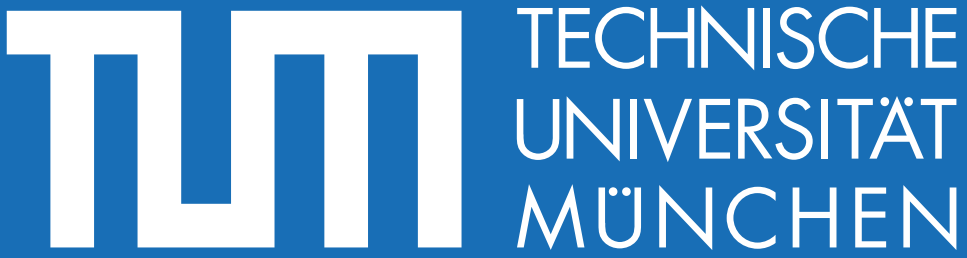


Software Engineering in der industriellen Praxis (SEIP)

Dr. Ralf S. Engelschall

ARCHITECTURE FUNDAMENTALS

Slice Architectures

(Patterns)

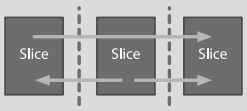
TECHNISCHE UNIVERSITÄT MÜNCHEN

AF 06.2

Slicing Principle

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

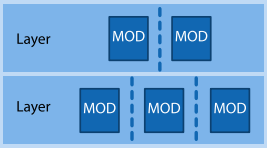
The particular slicing should minimize the total amount of individual relationships between the resulting slices. Per type of relationship, there should be no cycle in the transitive relationships.



MOD Concerned Module

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

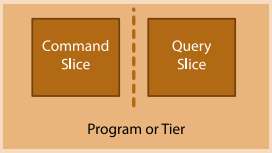
Rationale: Separation of Concern, Single Responsibility Principle, Mastering Complexity.



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).

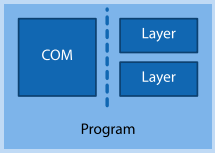
Rationale: Separated Scalability, Separated Data Access Patterns, Event Sourcing Approach.



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.

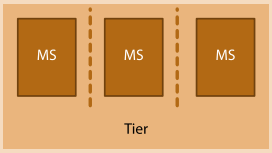
Rationale: Lack of Redundancy, Single Point of Truth, Reusability.



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.

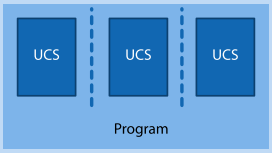
Rationale: Heterogeneity, Resilience, (Scalability), (Easy Deployment), (Organizational Alignment), (Composability), (Reusability), Replaceability.



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.

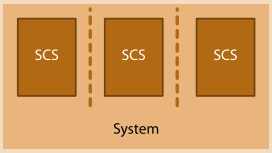
Rationale: Comprehensibility, Domain Alignment, Mastering Complexity.



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.



Beim **Slicing** werden Code oder Daten in zwei oder mehr logische (**logically**) — ggf. auch "physikalische" (**spatially**) — **Slices** geschnitten. Diese Slices sind klar unterscheidbar (**clearly distinct**), voneinander isoliert (**isolated**) und benannt (**named**). Slices werden immer **vertikal** gezeichnet.

Slices im selben Layer sollten möglichst unabhängig voneinander sein. Im Falle von Beziehungen sollte zumindest kein Zyklus existieren. Es gibt verschiedene spezielle Varianten von Slices, die jeweils einen eigenen Namen haben.

Concerned Modules sind Slices eines Layers, welche ein bestimmtes fachliches oder technisches Anliegen umsetzen. **Common Package** ist ein Slice eines Tiers, wohin Gemeinsamkeiten anderer Layer verschoben wurden. **Use-Case Slices** sind Slices eines Tiers, die sich jeweils dediziert um bestimmte fachliche Anwendungsfälle kümmern.

Bei der **Command-Query Responsibility Segregation** Architektur wird ein Tier in zwei Slices für Commands/Writes und Queries/Reads getrennt. Ein **Microservice** ist ein Slice eines Tiers, welcher als getrenntes Programm ausgeführt ist und sich um eine abgeschlossene fachliche Funktionalität kümmert. Ein **Self-Contained System** ist ein Slice eines gesamten Systems, welches als getrenntes Sub-System ausgeführt ist.

Fragen

- Wie nennt man die entstehenden Einheiten, wenn Code oder Daten **vertikal** geschnitten werden?
- Wie nennt man die Slices eines Tiers, welche als getrennte Programme ausgeführt sind, die sich jeweils um bestimmte fachlich abgeschlossene Funktionalitäten kümmern?

ARCHITECTURE FUNDAMENTALS

Flow Architectures
(Patterns)

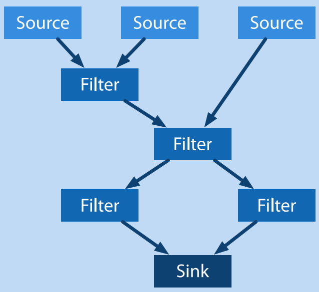
TECHNISCHE UNIVERSITÄT MÜNCHEN

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.

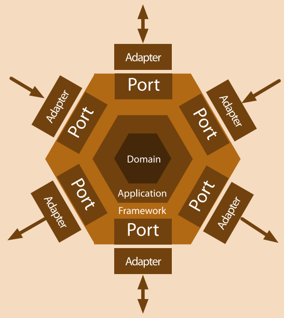
Examples: Unix commands with stdin/stdout/stderr and the Unix shell connecting them with pipes; Apache Spark or Apache Camel data stream processing pipelines.



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.

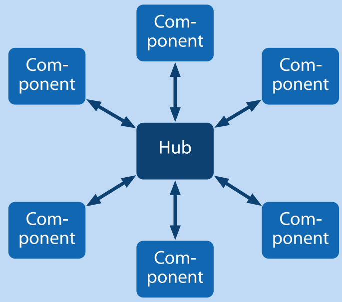
Examples: Message Queue, Enterprise Service Bus or Media Streaming Service internal realization.



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.



Bei den **Flow Architectures** geht es um den primären Datenfluss bzw. die primäre Kommunikation einer Anwendung. Hier gibt es u.a. die drei folgenden klassischen Architekturansätze.

Bei **Pipes & Filters** wird ein gerichteter **Graph** aufgebaut. Die Knoten des Graphen sind die **Components**, welche entweder vom Typ **Source** (Quelle), **Filter** (Filter) oder **Sink** (Senke) sind. Die Kanten des Graphen sind die **Pipes**: die Datenübertragungsstrecken zwischen den **Components**.

Bei dem speziellen **Ports & Adapters** (aka **Hexagonal Architecture**) wird eine "Hub & Spoke" Struktur aufgebaut. Der "Hub" sind Components des Anwendungskerns. Die "Spokes" bestehen jeweils aus einer **Component**, welche sich aus dem **Port** (das Interface) und dem **Adapter** (die Implementation) zusammensetzt.

Bei **Hub & Spoke** allgemein geht es darum, daß eine zentrale **Hub** Component als Kommunikationsmittelpunkt zwischen den sternförmig um den Hub angeordneten **Spoke** Components agiert. Der Knackpunkt dabei ist, daß die maximal NxN Kommunikationswege zwischen den Spoke Components dank des Hub auf nur N+M Kommunikationswege reduziert werden.

Fragen

❓ Mit Hilfe welcher **Flow Architecture** kann man N Komponenten so miteinander verschalten, daß statt NxN nur N+N Kommunikationswege entstehen?

ARCHITECTURE FUNDAMENTALS

Process Architectures

(Patterns)

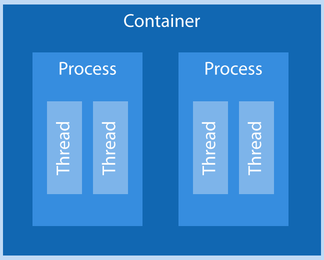
TECHNISCHE UNIVERSITÄT MÜNCHEN

Container, Process, Thread

The Operating System (OS) 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.

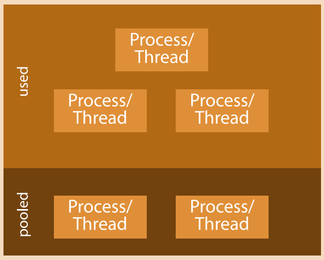


Process/Thread Pool

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

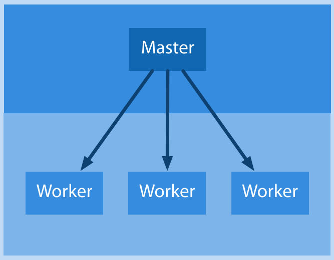


Master-Worker

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



Bei den **Process Architectures** geht es um das Zusammenspiel verschiedener **Container**, **Processes** oder **Threads**. Alle drei Konzepte kapseln Code und Daten. **Container** sind die stärkste Kapselung, bei der die Nutzung von sowohl CPU, RAM, Festplattenspeicher und Netzwerk gekapselt werden (z.B. Docker Container). Bei einem **Process** wird die Nutzung von CPU und RAM gekapselt (z.B. Unix Prozess). Bei einem **Thread**, der schwächsten Kapselung, wird nur noch die Nutzung der CPU gekapselt (z.B. Unix Thread).

Um gleichzeitig mehrere Anfragen beantworten zu können, werden in Server-Anwendungen mehrere Processes/Threads pro Anfrage genutzt. Da das ständige Erzeugen von solchen Processes/Threads nennenswert die Runtime-Performance reduziert und man die Hardware-Last üblicherweise limitieren und nicht vollkommen linear an die eingehenden Anfragen koppeln möchte, wird üblicherweise ein sogenannter **Pool** von einmalig erzeugten Worker-Processes/ Threads eingesetzt (z.B. Apache HTTPd oder NGINX).

Klassisch wird ein solcher Pool in einen **Master** Process/ Thread und viele **Worker** Processes/Threads unterteilt. Der permanent laufende Master erzeugt, steuert und stoppt die Worker. Üblicherweise sind die Worker ebenfalls permanent existent, werden im Falle von Fehlern aber vom Master ggf. aktiv gestoppt oder bei einem Crash automatisch neu gestartet (z.B. Node.js `cluster` Modul).

Fragen

- Mit welcher **Process Architecture** wird in der Praxis üblicherweise ein Process/Thread **Pool** verwaltet?

ARCHITECTURE FUNDAMENTALS

Cluster Architectures

(Patterns)

TECHNISCHE UNIVERSITÄT MÜNCHEN

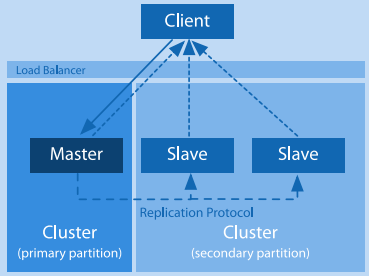
AF 07.3

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.

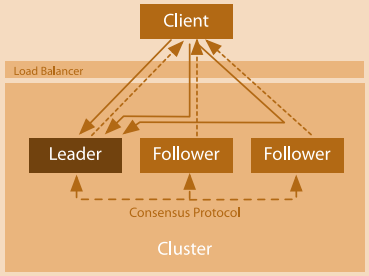


Leader-Follower (Dynamic Replication)

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.

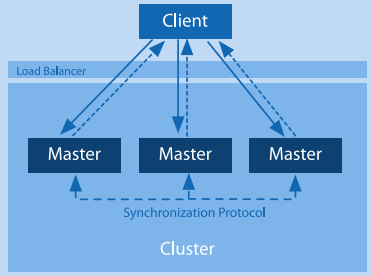


Master-Master (Synchronization)

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.



Bei den **Cluster Architectures** wird der Zusammenschluss von Rechnerknoten zu einem Cluster adressiert.

Bei der **Master-Slave** Architecture handelt es sich um eine statische Replikation der Daten von einem Master Server auf einen oder mehrere Slave Server. Die Clients können Lesezugriffe an alle Server senden, müssen aber Schreibzugriffe ausschließlich über den Master laufen lassen. Dies wird üblicherweise dazu verwendet, um die Read Performance zu steigern.

Bei der **Leader-Follower** Architecture handelt es sich um eine Art dynamische Replikation der Daten von einem Leader Server auf einen oder mehrere Follower Server. Die Clients können sowohl Lese- als auch Schreibzugriffe an alle Server senden. Da nur der Leader Server die Schreibzugriffe bearbeiten darf, leiten die Follower Server diese intern und für den Client transparent an den Leader Server weiter.

Dies ist auch der Unterschied zu Master-Slave: der Leader wird über ein Leader Election Protocol (bei einem Ausfall des aktuellen Leader Servers) automatisch und dynamisch zwischen allen Servern festgelegt. Der Vorteil ist, daß sich Leader-Follower für die Clients wie Master-Master anfühlt, der Cluster aber keine aufwändige Konfliktlösungsstrategie wie bei Master-Master umsetzen muss.

Bei der **Master-Master** Architecture handelt es sich um eine echte Synchronisation der Daten zwischen zwei oder mehr gleichberechtigten Master Server. Die Clients können sowohl Lese- als auch Schreibzugriffe an beliebige Master Server richten. Die Master Server müssen dazu aber intern eine aufwändige Konfliktlösungsstrategie umsetzen, um gleichzeitige Änderungen an denselben Daten aufzulösen.

Fragen

- Welche einfache **Cluster Architecture** kann man einsetzen, wenn die Read-Performance einer Server-Anwendung gesteigert werden soll?

ARCHITECTURE FUNDAMENTALS

Networking Architectures

(Patterns)

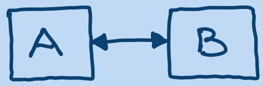
TECHNISCHE UNIVERSITÄT MÜNCHEN

AF 08.1

PTP Point-to-Point

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

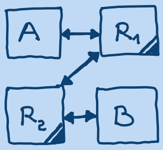
Rationale: simple communication where both nodes know about each other and can directly reach each other.



RTG Routing

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

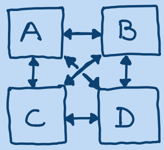
Rationale: simple communication where both nodes know about each other, but cannot directly reach each other.



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.

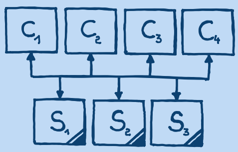
Rationale: communication without central control (although a seed peer is required).



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).

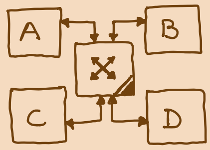
Rationale: communication with central orchestration, control and data storage.



BUS Bus/Broker/Relay

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

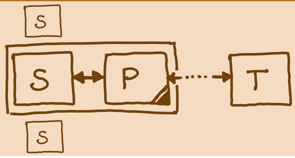
Rationale: decouple communication nodes: instead of Point-to-Point (PTP) communications between all nodes, there are just PTP communications with the hub.



FPR (Forward) Proxy

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

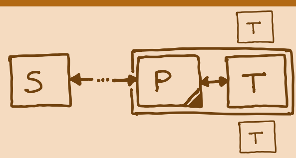
Rationale: bridge network topology constraints (segmented networks); caching at source side; auditing of communication.



RPR Reverse Proxy

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

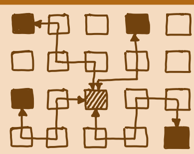
Rationale: load balancing for multiple target nodes; caching at target side; auditing of communication; security shielding of target nodes; protocol conversions.



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.



Bei den **Networking Architectures** wird die Netzwerk-topologische Kommunikation zwischen Rechnerknoten adressiert. Der einfachste Weg ist eine **Point-to-Point** Kommunikation über eine direkte Verbindung der Knoten.

Üblicherweise geht die Kommunikation heute aber über ein Netzwerk von Knoten, bei dem die einzelnen Nachrichten mit Hilfe von **Routing** über Zwischenknoten ausgetauscht werden.

Kommunizieren alle Knoten in sowohl Client- als auch Server-Rolle direkt miteinander, so spricht man von einer **Peer-to-Peer** Architektur. Sind einige Knoten nur in der Client-Rolle und andere nur in der Server-Rolle, so spricht man von einer **Client/Server** Architektur.

Um mehrere Knoten miteinander kommunizieren zu lassen, ohne daß diese sich jeweils kennen und adressieren müssen, nutzt man üblicherweise einen zentralen **Bus/Broker** Knoten und eine Stern-Topologie.

Sind zwischen Quelle (Source) und Ziel (Target) Zwischenknoten aktiv, welche als Stellvertreter (**Proxy**) in der Kommunikation agieren und nicht nur wie ein **Router** die Netzwerk-Pakete weiterleiten, spricht man entweder von einer **(Forward) Proxy** oder **Reverse Proxy** Situation. Ersteres, wenn der Proxy auf Seite des Quellknotens agiert, letzteres, wenn der Proxy als Stellvertreter des Zielknotens agiert.

Zusätzlich kann ein sogenanntes **Virtual Private Network** etabliert werden, bei dem ein logisches abgesichertes "Overlay Network" über ein physikalisches Netzwerk gelegt wird.

Fragen

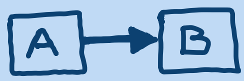
- Mit welcher **Network Architecture** kann man mehrere Knoten miteinander kommunizieren lassen, ohne daß diese Knoten sich jeweils untereinander genau kennen müssen?
- Wie nennt man einen Rechnerknoten, der als Stellvertreter für einen Zielknoten agiert?

Communication Architectures
(Patterns)

UCT Unicast (one-to-one)

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

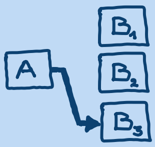
Rationale: private communication between exactly two nodes which both know each other beforehand.



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.

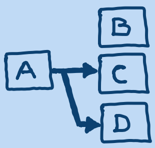
Rationale: Unicast, optimized for network failover scenarios, load balancing and CDNs.



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.

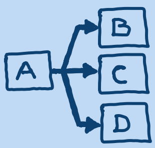
Rationale: node communication where destination nodes dynamically change or where total traffic should be reduced.



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.

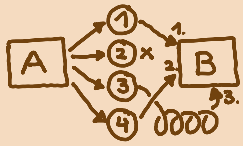
Rationale: spreading out messages to all available nodes for potential responses.



DGR Datagram (Single Packet)

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

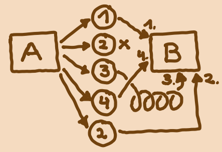
Rationale: simple low-overhead communication without prior communication establishment (handshake).



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).

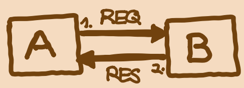
Rationale: reliable communication between nodes.



PLL Pull (Request/Response, RPC)

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

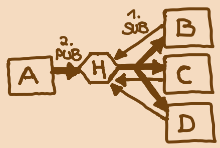
Rationale: Remote Procedure Call (RPC) like Unicast or Anycast communication.



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.



Bei den **Communication Architectures** wird die Kommunikationsart zwischen Komponenten adressiert. Man unterscheidet primär vier verschiedene Arten des Nachrichtenversands: beim **Unicast** sendet ein Quellknoten an exakt einen direkt adressierten Zielknoten. Beim **Anycast** sendet ein Quellknoten an eine Gruppe von potenziellen Zielknoten, die Nachricht wird aber nur an einem Zielknoten aus der Gruppe zugestellt.

Beim **Multicast** sendet ein Quellknoten ebenfalls an eine Gruppe von Zielknoten, die Nachricht wird aber an allen Zielknoten aus der Gruppe zugestellt. Beim **Broadcast** sendet ein Quellknoten an alle erreichbaren Zielknoten, ohne daß diese konkreten Zielknoten dem Quellknoten bekannt sind.

Bei der Art der Nachrichten unterscheidet man zwei Varianten: beim **Datagram** besteht jede Nachricht aus exakt einem Netzwerkpaket und beim Versand mehrerer Nachrichten werden keine Garantien gegeben, ob und in welcher Reihenfolge die Nachrichten beim Zielknoten ankommen werden. Im Gegensatz dazu, besteht beim **Stream** eine Nachricht aus einer Sequenz von Netzwerkpaketen und dabei werden verschiedene Garantien gegeben:

Bei Paketstaus auf Zwischenknoten wird beim **Stream** ggf. die Quelle gedrosselt. Bei Paketverlust werden Pakete erneut versandt. Und man erhält ggf. die Kontrolle darüber, ob das Paket am Zielknoten maximal einmal, genau einmal, oder mindestens einmal zugestellt wird.

Man unterscheidet üblicherweise zwei Modi der Client/Server-Kommunikation: beim **Pull**-Modus sendet der Client eine Anfrage und der Server sendet eine Antwort. Der Server kann hierbei insbesondere nicht (ohne vorherige Anfrage) eine Nachricht proaktiv versenden. Beim **Push**-Modus sendet der Client vorab eine Nachricht an den Server, um bestimmte Arten von Nachrichten zu abonnieren. Danach kann jederzeit der Server eine Nachricht an alle dafür abonnierten Clients senden.

Üblicherweise wird **Pull** über **Unicast**/**Anycast** und als **Stream** implementiert, beispielsweise im Protokoll HTTP. **Push** wird dagegen üblicherweise über **Multicast**/**Broadcast** als **Datagram** implementiert, beispielsweise im Protokoll DHCP.

Fragen

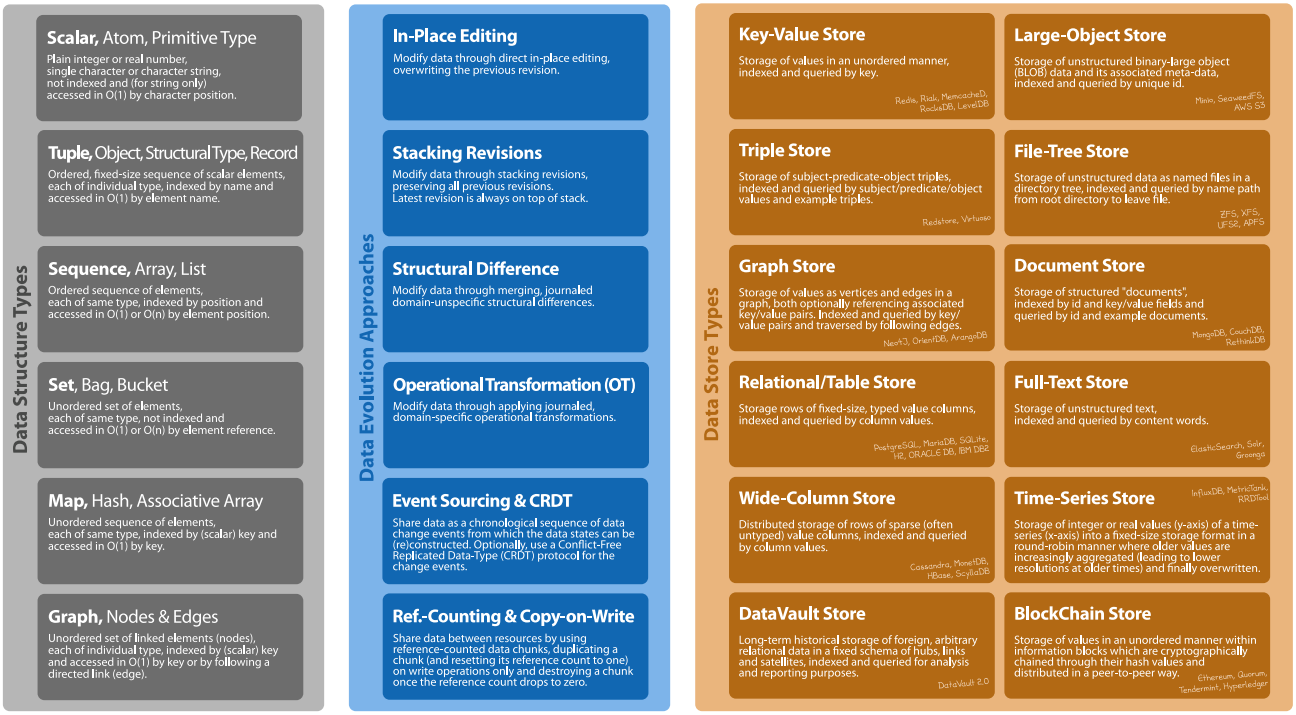
- Welches bekannte Web-Protokoll nutzt eine auf **Unicast**, **Stream** und **Pull** basierende Kommunikation?

ARCHITECTURE FUNDAMENTALS

Data Structure Architectures

(Patterns)

TECHNISCHE UNIVERSITÄT MÜNCHEN



AF 09.1

Der Software Architect unterscheidet nur 6 **Data Structure Types** für Daten-Elemente: **Scalar** (z.B. Integer, String, etc), **Tuple** (ordered fixed-size sequence of Scalars), **Sequence** (ordered sequence of elements), **Set** (unordered set of elements), **Map** (unordered set of elements, each indexed by key) und **Graph** (unordered set of elements, each indexed by key or by following a link between elements). Alle komplexen spezifischen Datenstrukturen in der Praxis sind für den Software Architect nur die Kombination dieser 6 Typen.

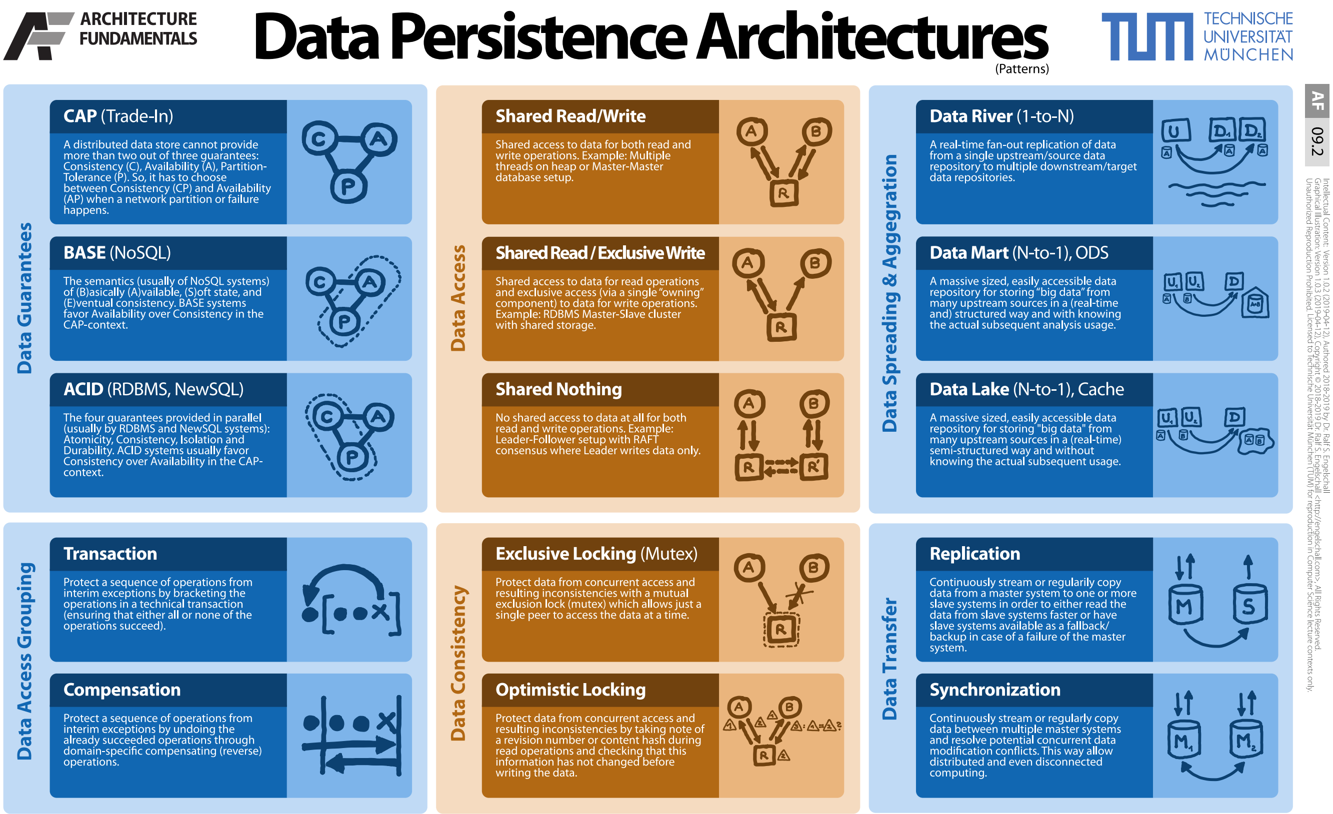
Es gibt zahlreiche **Data Evolution Approaches**, mit denen sich Daten über die Zeit verändern können: im einfachsten Fall, **In-Place Editing**, werden Daten einfach direkt geändert. Ein Zugriff auf frühere Stände gibt es hierbei nicht. Soll auf frühere Stände zugegriffen werden können, so kann man **Stacking Revisions** einsetzen, bei dem vor jeder Änderung der gesamte Datensatz zuerst kopiert wird. Damit nicht der gesamte Datensatz kopiert werden muss, wird bei **Structural Difference** nur eine technische Differenz von altem und neuem Datensatz gespeichert. Alternativ können bei **Operational Transformation** die fachlichen Änderungsoperationen als Journal gespeichert werden.

Wird so ein Journal benutzt, um auch Replika der Datensätze aktuell zu halten, spricht man von **Event Sourcing**. Falls das Journal als dem Protokoll von sogenannten **Conflict-Free Replicated Data-Types (CRDT)** aufgebaut ist, lässt sich statt (unidirektionaler) Replikation auch eine (bidirektionale) Synchronisation erzielen. Falls mehrere konkurrierende Prozesse/Threads logisch auf Kopien, aber physikalisch auf denselben Datensätzen operieren, lässt sich mit **Copy-on-Write** und **Reference Counting** ein gemeinsamer Zugriff und der Lebenszyklus der Datensätze dennoch sinnvoll steuern.

Zur Speicherung von Daten in Datenbanken gibt es zahlreiche **Data Store Types**. Diese unterschieden sich primär in der Art und Flexibilität der Datenstruktur und den gewährten Garantien. Der verbreitetste Typ ist der **Relational/Table Store**. Der eleganteste Typ ist der **Graph Store**. Der bequemste ist der **Document Store**.

Fragen

- Nennen sie 3 **Data Evolution Approaches**, die es jeweils erlauben, auf die früheren Stände der Daten zuzugreifen?



Im Bereich der **Data Guarantees** gibt es drei wesentliche Aspekte: Das **CAP** Theorem adressiert den sog. "Trade-In": Man muss sich in der Praxis üblicherweise zwischen Consistency + Partition-Tolerance (CP) oder Availability + Partition-Tolerance (AP) entscheiden. Beides gleichzeitig geht nicht. Bei **BASE**-Systemen favorisiert man üblicherweise AP. Bei einem traditionellen RDBMS mit **ACID** Garantien favorisiert man üblicherweise CP.

Beim **Data Access Grouping** kennt man **Transaction** und **Compensation**. Ersteres ist eine "technische Klammer", die einem erlaubt, in Fehlerfall wieder zu dem früheren Zustand zurückzukehren. Letzteres ist eine "fachliche Klammer", bei der sog. Kompensationsoperationen einem erlauben die früheren Änderungen zu "stornieren", um so einen früheren konsistenten Zustand wiederzuerlangen.

Beim **Data Access** von zwei oder mehr Prozessen/Threads auf die gleichen Daten unterscheidet man die Ansätze **Shared Read/Write** (alle lesen und schreiben dieselben Daten), **Shared Read / Exclusive Write** (alle lesen und nur einer schreibt dieselben Daten) und **Shared Nothing** (alle lesen und schreiben auf die gleichen synchronisierten Daten).

Bei der **Data Consistency** kennt man **Exclusive Locking** (pro Zeiteinheit schreibt nur einer) und **Optimistic Locking** (alle versuchen zu schreiben, erkennen und lösen aber einen Konflikt).

Beim **Data Spreading & Aggregation** unterscheidet man drei Arten: beim **Data River** werden die Daten von einem Master-System auf viele Slave-Systeme repliziert, um u.a. eine höhere Read-Performance zu erzielen. Beim **Data Mart** (strukturierte Daten) und **Data Lake** (semi-strukturierte Daten) werden dagegen die Daten von vielen Master-Systemen auf ein Slave-System repliziert, um u.a. die Daten zentral auswerten oder "cachen" zu können.

Beim **Data Transfer** wird zuletzt noch zwischen der unidirektionalen und konfliktfreien **Replication** und der bidirektionalen und konfliktreichen **Synchronisation** unterschieden.

Fragen

? Wie nennt man den Ansatz, bei dem Daten von einem Master-System auf viele Slave-Systeme repliziert werden?