

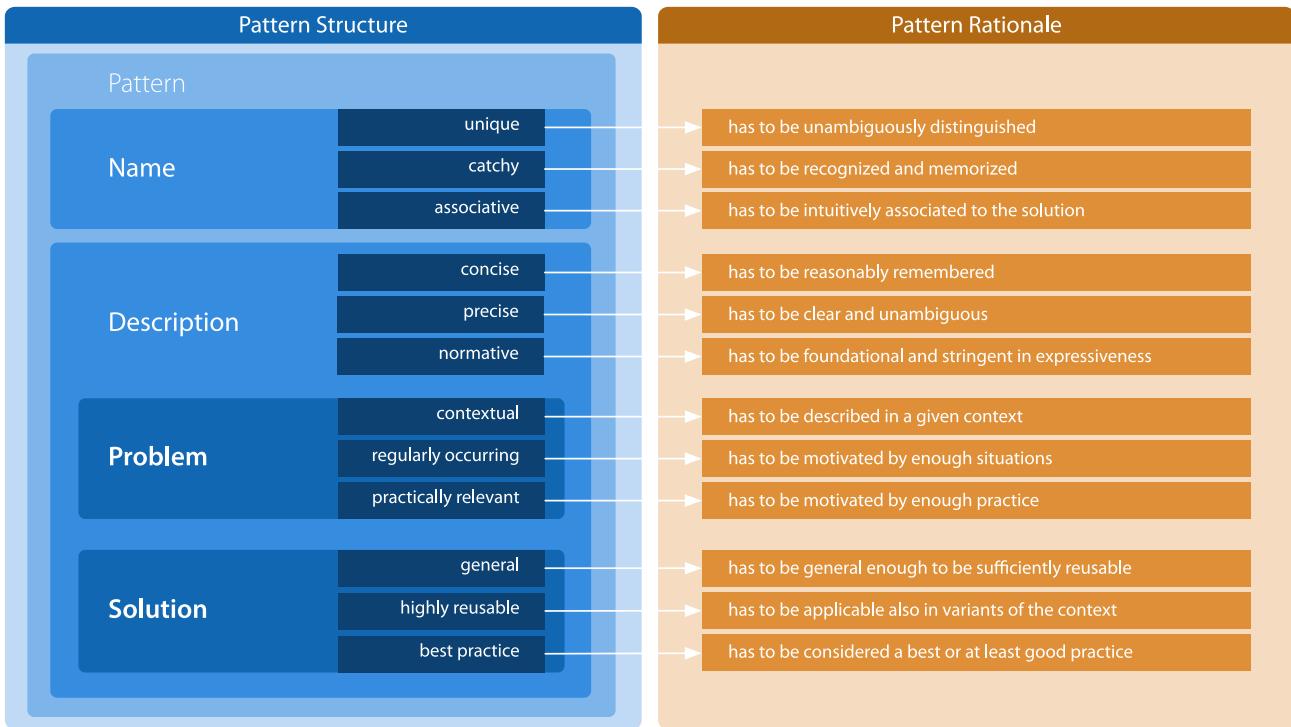
# Software Engineering in Industrial Practice (SEIP)

Dr. Ralf S. Engelschall



## Pattern Definition

**Pattern:** unique, catchy, and associative **Name** and concise, precise, and normative **Description** of a contextual, regularly occurring, and practically relevant **Problem** and a general, highly reusable, and best practice **Solution** for it.



Definition eines **Architekturmusters**: einzigartiger, eingängiger und assoziativer Name und prägnante, präzise und normative Beschreibung eines kontextuellen, regelmäßig auftretenden und praktisch relevanten Problems und einer allgemeinen, hochgradig wiederverwendbaren und praxisbewährten Lösung für dieses Problem.

Die Begründungen ist, daß ein **Architekturmuster**: eindeutig unterschieden werden muß, wiedererkannt und erinnert werden muß, intuitiv mit der Lösung assoziiert werden muß, sich gut merken lassen muß, klar und eindeutig sein muß, grundlegend und stringent in der Ausdrucksfähigkeit sein muß, in einem bestimmten Kontext beschrieben werden muß, durch genügend Situationen motiviert sein muß, durch genügend Praxis motiviert sein muß, um ausreichend wiederverwendbar zu sein auch allgemein genug sein muß, auch in Varianten des Kontexts anwendbar sein muß, und als beste oder zumindest gute Praxis angesehen werden muß.

**Architekturmuster** ermöglichen es einem insbesondere effizient zu kommunizieren (Name) und von den eingeflossenen Erfahrungen zu profitieren (Best Practice).

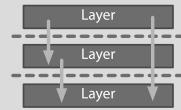
## Fragen

❓ Warum sind **Architekturmuster** interessant?

## Layering Principle

Horizontally split code or data into two or more logically, optionally also spatially, clearly distinct, isolating, named, and ranked Layers.

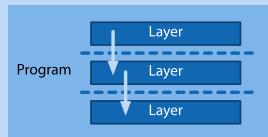
A Layer is not allowed to have relationships to or knowledge about any upper Layers. Additionally, for **Closed Layering**, each Layer is allowed to have relationships to and knowledge about the *directly* lower Layer only. In contrast to **Open Layering** or **Leaky Abstraction**, where each Layer is allowed to have relationships to and knowledge about *any* lower Layer.



### LR Layer

Split related code or data of a Program into two or more logically distinct domain- or technology-induced Layers.

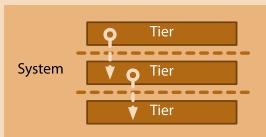
Rationale: Separation of Concern, Single Responsibility Principle, Mastering Complexity, Change Isolation, Functional Abstraction.



### TR Tier

Split related code or data of a System into two, three or more logically and spatially distinct, network-connected, domain- or technology-induced Tiers.

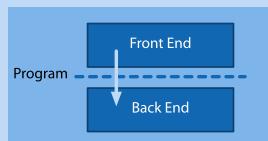
Rationale: Separation of Concern, Single Responsibility Principle, Mastering Complexity, Change Isolation, Functional Abstraction, Deployment Partitioning.



### FB Front End / Back End

Split the code of a Program into exactly two logical Layers: a user-facing Front End and a data-facing Back End.

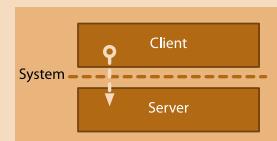
Rationale: Separation of Concern, Single Responsibility Principle, Mastering Complexity, Change Isolation, Functional Abstraction, Organisational Alignment.



### CS Client / Server

Split the code of a System into two spatially distinct, network-connected Layers, each forming a stand-alone Program: a user-facing and multi-instantiated (Rich) Client and a data-facing (and logically) single-instantiated (Thin) Server. Both contain a Front/Back End.

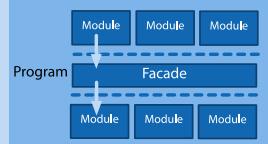
Rationale: Multi-User, User Computing Resource Leverage, Distributed Computing.



### FD Facade

Splice a domain-specific Facade Layer into two Layers of two or more Modules. The extra Facade Layer acts as a broker between the Modules.

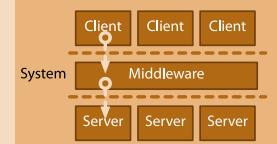
Rationale: Information Hiding, Cross-Cutting Concern Centralization, Functionality Orchestration, Authorization, Validation, Conversion.



### MW Middleware

Splice a domain-unspecific Middleware Layer into a Client/Server communication. The extra Layer is a stand-alone Program Tier and acts as a broker between Client and Server.

Rationale: Communication Peer Discovery Simplification, Transport Protocol Conversions, Network Topology Flexibility.



Beim **Layering** werden Code oder Daten in zwei oder mehr logische (**logically**) — ggf. auch “physikalische” (**spatially**) — Layer geschnitten. Diese Layer sind klar unterscheidbar (**clearly distinct**), von einander isoliert (**isolated**), benannt (**named**) und stehen in einer Rangordnung (**ranked**). Layer werden immer **horizontal** gezeichnet.

Ein Layer hat keine Beziehung (**relationship**) zu, oder Wissen (**knowledge**) über, irgendwelche Layer über ihm. Zusätzlich darf er, beim **Closed Layering**, eine Beziehung zu oder Wissen über den direkten Layer unter ihm haben. Außerdem darf er beim **Open Layering** bzw. der **Leaky Abstraction** sogar eine Beziehung zu oder Wissen über irgendwelche Layer unter ihm haben.

Falls sich das Layering über Netzwerk-Grenzen erstreckt oder eine “physikalische” Grenze durchbricht, spricht man nicht mehr von einzelnen Layern, sondern von **Tiers**.

Zerfällt ein Programm in einen vorderen bzw. auf das User-Interface fokussierenden Layer und einen hinteren bzw. auf Daten fokussierenden Layer, so spricht man bei den beiden Layern auch von **Front End** und **Back End** des Programms. Dies ist nicht zu verwechseln mit **Client** und **Server**, welche die zwei Tiers eines Systems über ihre speziellen Rollen benannt. Sowohl der Client als auch der Server sind eigenständige Programme mit jeweils einem Front End und einem Back End.

Eine sehr spezieller und prominenter Layer ist die **Facade**, welche innerhalb eines Programms die Module von zwei anderen Layern von einander trennt. Eine Variante der Facade auf Ebene eines Systems (statt auf der Ebene eines Programms) ist die **Middleware**, welche eine Client/Server-Kommunikation auf trennt.

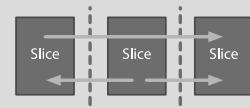
## Fragen

- ?
- Wie nennt man die entstehenden Einheiten, wenn Code oder Daten **horizontal** geschnitten werden?
- ?
- Was ist der Unterschied zwischen den Layer-Pärchen **Front/Back End** und **Client/Server**?

## Slicing Principle

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

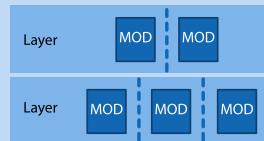
The particular slicing should minimize the total amount of individual relationships between the resulting slices. Per type of relationship, there should be no cycle in the transitive relationships.



### MOD Concerned Module

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

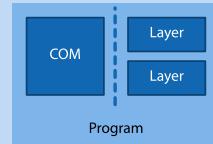
Rationale: Separation of Concern, Single Responsibility Principle, Mastering Complexity.



### 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.

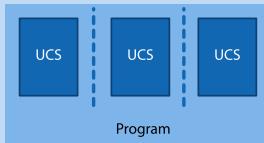
Rationale: Lack of Redundancy, Single Point of Truth, Reusability.



### 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.

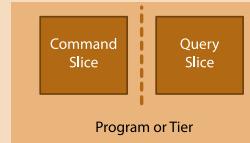
Rationale: Comprehensibility, Domain Alignment, Mastering Complexity.



### 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).

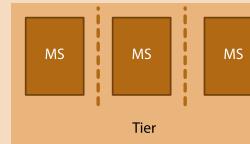
Rationale: Separated Scalability, Separated Data Access Patterns, Event Sourcing Approach.



### 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.

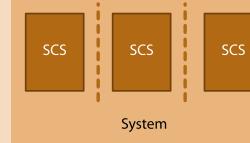
Rationale: Heterogeneity, Long-Term Large-Scale Maintenance, Replaceability, Resilience, (Scalability), (Easy Deployment), (Organizational Alignment), (Composability), (Reusability).



### 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, 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.

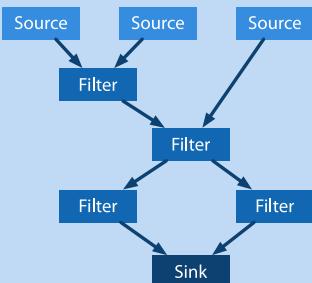
## Fragen

- ?
- 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?

## 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.

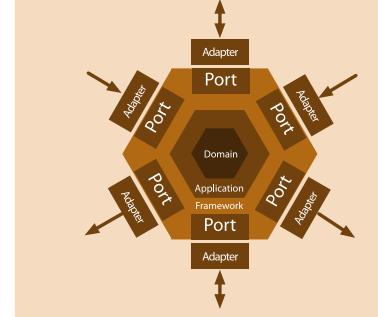
Examples: Unix commands with stdin/stdout/stderr and the Unix shell connecting them with pipes; Apache Spark or Apache Camel data stream processing pipelines.



## 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.

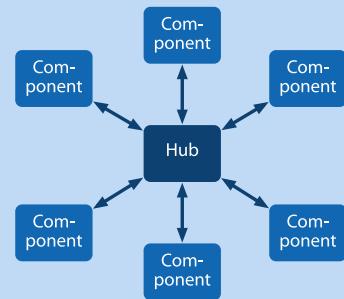
Examples: Message Queue, Enterprise Service Bus or Media Streaming Service internal realization.



## Hub & Spoke

Perform communication (the **Spoke**) between multiple Components through a central **Hub** Component. Instead of having to communicate with  $N \times (N-1) / 2$  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  $K \times (N - K)$  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  $N \times (N - 1) / 2$  Kommunikationswege zwischen den **Spoke**-Components dank der **Hub**-Component auf nur  $N$  Kommunikationswege reduziert werden.

## Fragen

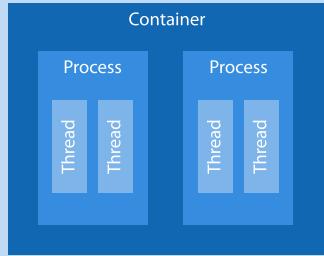
- ❓ Mit Hilfe welcher **Flow Architecture** kann man  $N$  Komponenten so miteinander verschalten, daß statt  $N \times (N - 1) / 2$  nur  $N$  Kommunikationswege entstehen?

### 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.

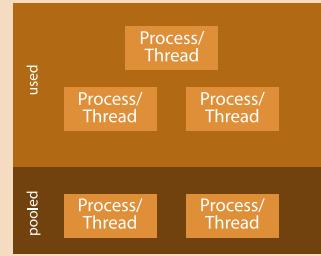


### Process/Thread Pool

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

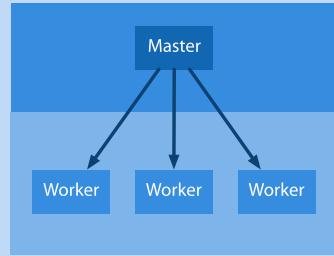


### Master-Worker

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



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

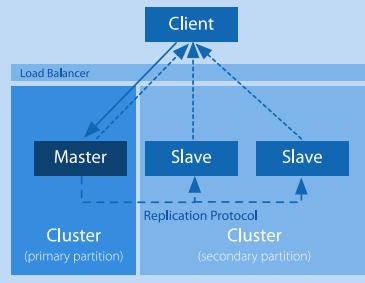
- ❓ Mit welcher **Process Architecture** wird in der Praxis üblicherweise ein **Process/Thread Pool** verwaltet?

## 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.

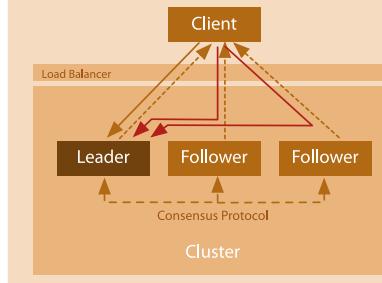


## Leader-Follower (Dynamic Replication)

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.

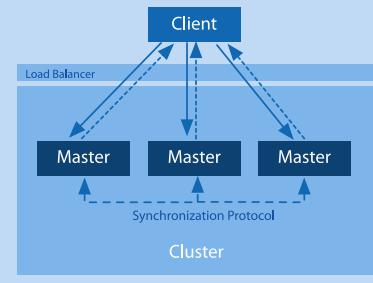


## Master-Master (Synchronization)

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, Autmerge/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.

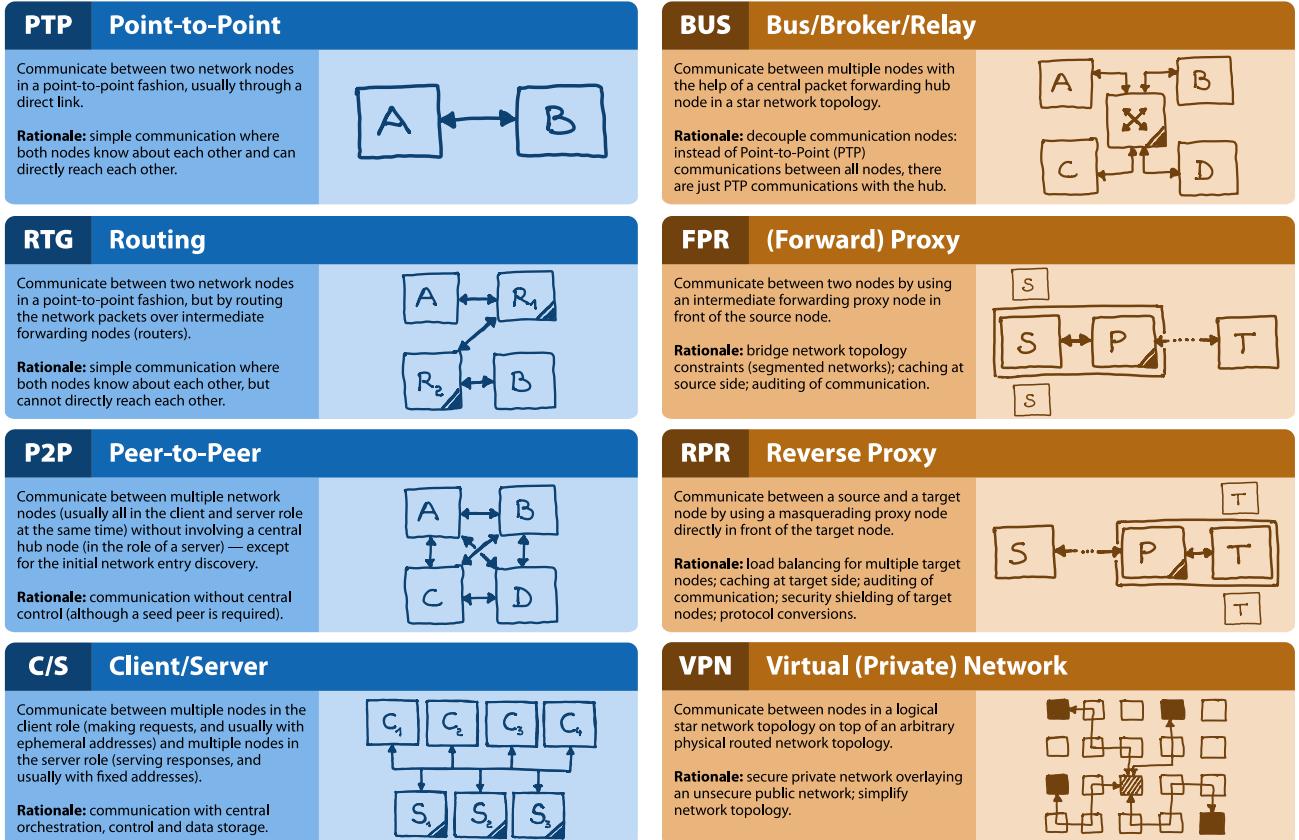
Bei der **Leader-Follower** Architecture handelt es sich um eine Art dynamische Replikation der Daten von einem Leader Server auf einen oder mehrere Follower Server. Die Clients können sowohl Lese- als auch Schreibzugriffe an alle Server senden. Da nur der Leader Server die Schreibzugriffe bearbeiten 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?



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?

<b>UCT</b>	<b>Unicast</b> (one-to-one)
	<p>Communicate messages from one source to exactly one destination node. The destination node is explicitly and individually addressed.</p> <p><b>Rationale:</b> private communication between exactly two nodes which both know each other beforehand.</p>
<b>ACT</b>	<b>Anycast</b> (one-to-any)
	<p>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.</p> <p><b>Rationale:</b> Unicast, optimized for network failover scenarios, load balancing and CDNs.</p>
<b>MCT</b>	<b>Multicast</b> (one-to-many)
	<p>Communicate messages from one source to many destination nodes. The destination nodes usually form a group and are usually not individually addressed.</p> <p><b>Rationale:</b> node communication where destination nodes dynamically change or where total traffic should be reduced.</p>
<b>BCT</b>	<b>Broadcast</b> (one-to-all)
	<p>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.</p> <p><b>Rationale:</b> spreading out messages to all available nodes for potential responses.</p>
	<b>DGR</b> <b>Datagram</b> (Single Packet)
	<p>Communicate messages as an unordered set of single packets, usually without any network congestion control, retries or other delivery guarantees.</p> <p><b>Rationale:</b> simple low-overhead communication without prior communication establishment (handshake).</p>
	<b>STR</b> <b>Stream</b> (Sequence of Packets)
	<p>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).</p> <p><b>Rationale:</b> reliable communication between nodes.</p>
	<b>PLL</b> <b>Pull</b> (Request/Response, RPC)
	<p>Communicate by performing a request (from the client node) and pulling a corresponding response (from the server node).</p> <p><b>Rationale:</b> Remote Procedure Call (RPC) like Unicast or Anycast communication.</p>
	<b>PSH</b> <b>Push</b> (Publish/Subscribe, Events)
	<p>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.</p> <p><b>Rationale:</b> event-based Multicast or Broadcast communication.</p>

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?