TUM

TECHNISCHE
UNIVERSITÄT
MÜNCHEN

Software Engineering in der industriellen Praxis (SEIP)
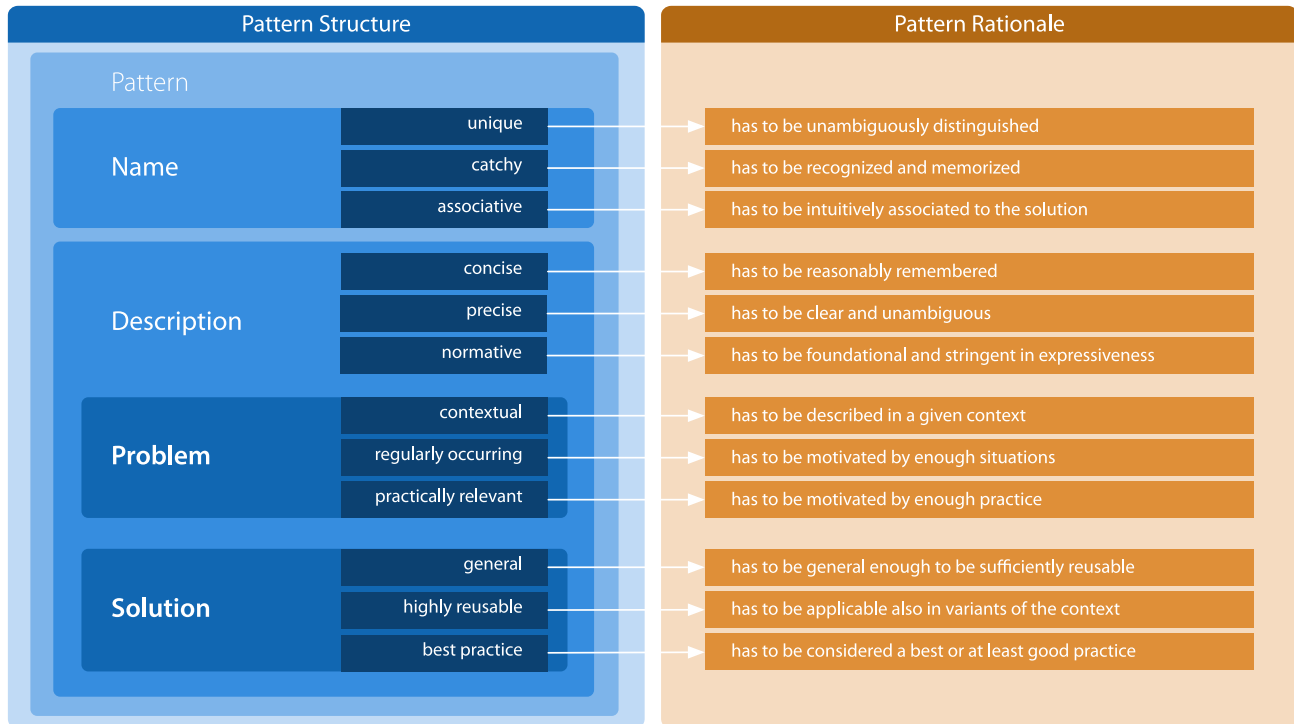
Dr. Ralf S. Engelschall

Architecture Patterns

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.

Pattern Structure

Pattern

Name
- unique
- catchy
- associative

Description
- concise
- precise
- normative

Problem
- contextual
- regularly occurring
- practically relevant

Solution
- general
- highly reusable
- best practice

Pattern Rationale

- has to be unambiguously distinguished
- has to be recognized and memorized
- has to be intuitively associated to the solution

- has to be reasonably remembered
- has to be clear and unambiguous
- has to be foundational and stringent in expressiveness

- has to be described in a given context
- has to be motivated by enough situations
- has to be motivated by enough practice

- has to be general enough to be sufficiently reusable
- has to be applicable also in variants of the context
- has to be considered a best or at least good practice

Definition of an **Architecture 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.

The rationales are that an **Architecure Pattern**: has to be unambiguously distinguished, has to be recognized and memorized, has to be intuitively associated to the solution, has to be reasonably remembered, has to be clear and unambiguous, has to be foundational and stringent in expressiveness, has to be described in a given context, has to be motivated by enough situations, has to be motivated by enough practice, has to be general enough to be sufficiently reusable, has to be applicable also in variants of the context, and has to be considered a best or at least good practice.

Architecture Patterns especially allow one to efficiently communicate (name) and benefit from their captured experience (best practice).

Questions

❓ Why are **Architecture Patterns** interesting?

Layer Architectures
(Patterns)

ARCHITECTURE FUNDAMENTALS

TECHNISCHE UNIVERSITÄT MÜNCHEN

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.
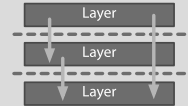

LR Layer

Split related code or data of a Program into two or more logically distinct domain- or technology-induced Layers.
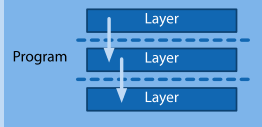
Rationale: Separation of Concern, Single Responsibility Principle, Mastering Complexity, Change Isolation, Functional Abstraction.


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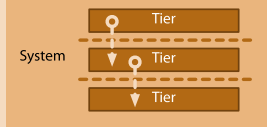
Rationale: Separation of Concern, Single Responsibility Principle, Mastering Complexity, Change Isolation, Functional Abstraction, Deployment Partitioning.


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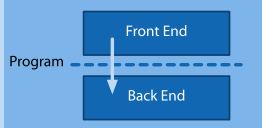
Rationale: Separation of Concern, Single Responsibility Principle, Mastering Complexity, Change Isolation, Functional Abstraction, Organisational Alignment.
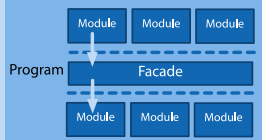

CS Client / Server

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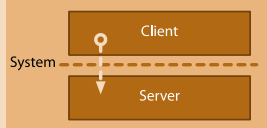
Rationale: Multi-User, User Computing Resource Leverage, Distributed Computing.


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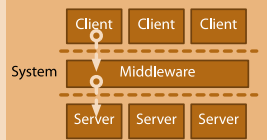
Rationale: Communication Peer Discovery Simplification, Transport Protocol Conversions, Network Topology Flexibility.

With **Layering**, code or data are cut into two or more **logically** — if necessary, also "physically" (**spatially**) — **Layer**. These layers are **clearly distinct**, **isolated** from each other, **named** and **ranked**. Layers are always drawn **horizontally**.

A layer has no **relationship** to, or **knowledge** about, any layers above him. In addition, he, in **Closed Layering**, has a relationship with, or knowledge about, the direct layer below him. In addition, he may have a relationship to, or knowledge about, any layer below him in **Open Layering** or **Leaky Abstraction**.

If the layering extends across network boundaries or a "physical" boundary, one no longer speaks of individual Layers, but of **Tiers**.

If a Program is split into a front or user interface focusing layer and a back or data focusing layer, the two layers are called **Front End** and **Back End** of the Program. This is not to be confused with **Client** and **Server**, which names two Tiers of a System through their special role. Both Client and Server are standalone Programs, each with a Front End and a Back End.

A very special and prominent layer is the **Facade**, which separates the Modules of two Layers within a Program. A variant of the Facade at the level of a System (instead of at the level of a Program) is the **Middleware**, which breaks apart a Client/Server communication.

Questions

❓ How do one call the resulting units if code or data is split **horizontally**?

❓ What is the difference between the Layer-pairs **Front/Back End** and **Client/Server**?

Slice Architectures

(Patterns)

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.
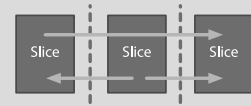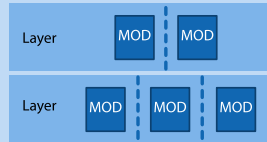

MOD — Concerned Module

Split related code or data (usually across a single Layer) into two or more logically distinct domain- or technology-induced Modules.
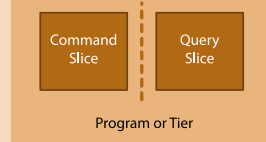
Rationale: Separation of Concern, Single Responsibility Principle, Mastering Complexity.

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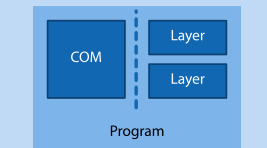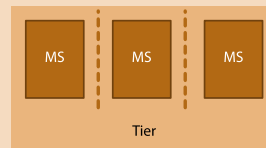
Rationale: Separated Scalability, Separated Data Access Patterns, Event Sourcing Approach.

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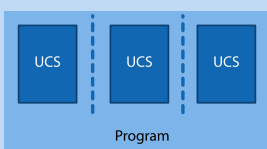
Rationale: Heterogeneity, Long-Term Large-Scale Maintenance, Replaceability, Resilience, (Scalability), (Easy Deployment), (Organizational Alignment), (Composability), (Reusability).

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, Resilience, Scalability, Easy Deployment, Organizational Alignment, Reusability, Replaceability.
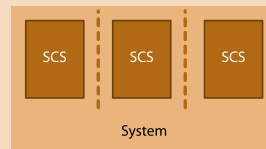
When **Slicing**, code or data are split into two or more **logically** — if necessary also "physically" (**spatially**) — **Slices**. These slices are **clearly distinct**, **isolated** from each other, and **named**. Slices are always drawn **vertically**.

Slices in the same Layer should be as independent of each other as possible. In the case of relationships, at least no cycle should exist. There are different special variants of slices, each of which has its own name.

Concerned Modules are Slices of a Layer that realize a specific domain-specific or technical concern. **Common Package** is a Slice of a Tier, where commonalities of other Layers were moved to. **Use-Case Slices** are Slices of a Tier that are dedicated to certain domain-specific use cases.

With the **Command-Query Responsibility Segregation** architecture, a Tier is split into two Slices for Commands/Writes and Queries/Reads. A **Microservice** is a Slice of a Tier, which is executed as a separate Program and which is concerned with a closed domain-specific functionality. A **Self-Contained System** is a Slice of a whole System that is executed as a separate Sub-System.

Questions

❓ What does one call the resulting units when code or data is split **vertically**?

❓ What does one call the Slices of a Tier, which are executed as separate Programs and which are concerned with closed domain-specific functionalities?

Flow Architectures
(Patterns)

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.

The **Flow Architectures** are concerned with the primary data flow or the primary communication of an application. Here the following three classical architectural approaches exist.

With **Pipes & Filters** a directed **Graph** is built. The nodes of the graph are the **Components**, which are either of type **Source**, **Filter** or **Sink**. The edges of the graph are the **Pipes**: the data transmission links between the **Components**.

With the special **Ports & Adapters** (aka **Hexagonal Architecture**) a "Hub & Spoke" structure is set up. The "Hub" are the Components of the application core. The "Spokes" each consist of a **Component**, which is composed of the **Port** (the interface) and the **Adapter** (the implementation).

With **Hub & Spoke** in general, a central **Hub** Component acts as the communication center between **Spoke** components which are star-shaped around the **Hub**. The crux is that the maximum N x (N - 1) / 2 communication paths between the **Spoke** components, thanks to the **Hub** component, can be reduced to just N communication paths.

Questions

❓ With the help of which **Flow Architecture** can N components be connected with eachother in a way that instead of N x (N - 1) / 2 communication paths only N are created?

Process Architectures
(Patterns)

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.
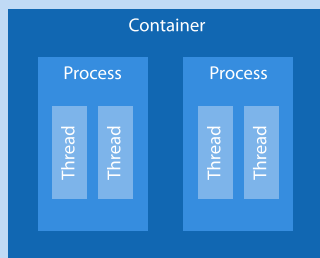
Process/Thread Pool

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
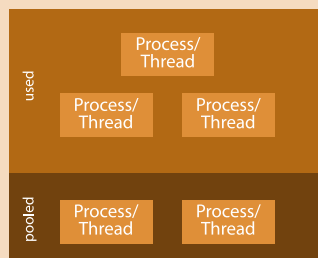
Master-Worker

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
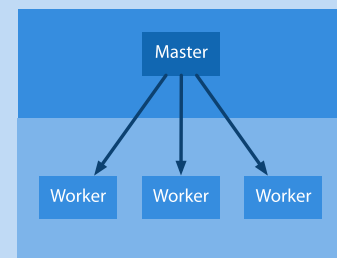
The **Process Architectures** are all about the interaction between different **Containers**, **Processes** or **Threads**. All three concepts encapsulate code and data. **Containers** are the strongest capsule, which encapsulates both CPU, RAM, hard disk, and network (e.g. Docker Container). A **Process** encapsulates CPU and RAM (e.g., Unix process). In the case of a **Thread**, the weakest capsule, only the CPU is encapsulated (e.g., Unix thread).

In order to be able to answer several requests at the same time, server applications use multiple processes/threads per request. Since the constant creation of such processes/threads noteworthy reduces the runtime performance and the hardware load typically should be limited and not linearly be coupled to the incoming requests, a so-called **Pool** of one-time created worker processes/threads is used (e.g., Apache HTTPd or NGINX).

Classically, such a pool is split into a single **Master** Process/Thread and multiple **Worker** Processes/Threads. The permanently running Master generates, controls, and stops the Workers. Usually, the Workers are also permanently existent, but in the event of errors, the Master will actively stop them, or in case of a crash, automatically restart them (e.g., Node.js `cluster` module).

Questions

❷ With which **Process Architecture** is in practice a Process/Thread **Pool** usually managed?

Cluster Architectures

(Patterns)

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.
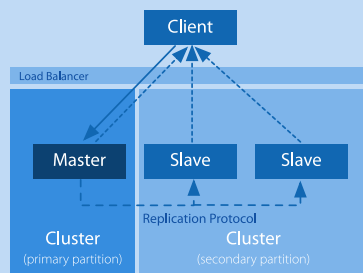
Leader-Follower (Dynamic Replication)

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.

→ Write Operation
--→ Read Operation
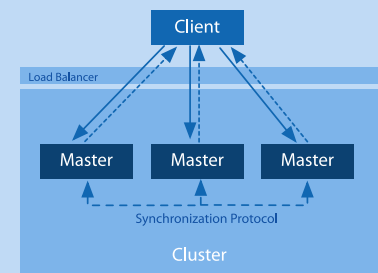
In **Cluster Architectures**, the merger of compute nodes to a cluster is addressed.

The **Master-Slave** architecture is a static replication of data from a Master server to one or more Slave servers. The Clients can send read requests to all Servers, but write requests must be run exclusively via the Master. This is usually used to increase the Read Performance.

The **Leader-Follower** architecture is a kind of dynamic replication of data from a Leader server to multiple Follower servers. The Clients can send read and write requests to all servers. Since only the Leader server can handle write requests, the Follower servers, internally and intransparently for the Client, forward these to the Leader server.

This is also the difference to Master-Slave: the Leader is selected automatically and dynamically between all servers via a Leader Election Protocol (in the event of a failure of the current Leader server). The advantage is that Leader-Follower to Clients feels like Master-Master, but the cluster does not require any complex conflict resolution strategy as is the case with Master-Master.

The **Master-Master** Architecture is a genuine synchronization of data between two or more equal Master servers. The Clients can send both read and write requests to any Master server. However, the Master servers internally must implement an elaborate conflict resolution strategy in order to resolve simultaneous changes to the same data.
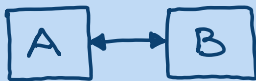
Questions

❷ Which simple **Cluster Architecture** can be used if the read performance of a server application should be increased?

Networking Architectures
(Patterns)

TECHNISCHE UNIVERSITÄT MÜNCHEN

PTP   Point-to-Point

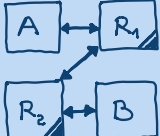Communicate between two network nodes in a point-to-point fashion, usually through a direct link.

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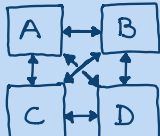
Rationale: simple communication where both nodes know about each other, but cannot directly reach each other.

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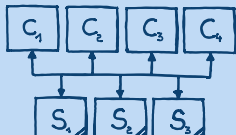
Rationale: communication without central control (although a seed peer is required).

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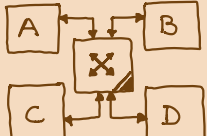
Rationale: communication with central orchestration, control and data storage.

BUS   Bus/Broker/Relay

Communicate between multiple nodes with the help of a central packet forwarding hub node in a star network topology.
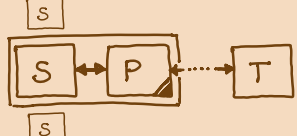
Rationale: decouple communication nodes: instead of Point-to-Point (PTP) communications between all nodes, there are just PTP communications with the hub.

FPR   (Forward) Proxy

Communicate between two nodes by using an intermediate forwarding proxy node in front of the source node.
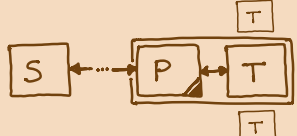
Rationale: bridge network topology constraints (segmented networks); caching at source side; auditing of communication.

RPR   Reverse Proxy

Communicate between a source and a target node by using a masquerading proxy node directly in front of the target node.
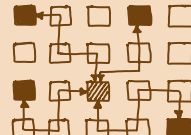
Rationale: load balancing for multiple target nodes; caching at target side; auditing of communication; security shielding of target nodes; protocol conversions.

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.

In **Networking Architectures**, the network-topological communication between computer nodes is addressed. The simplest way is **Point-to-Point** communication via a direct connection of the nodes.

Usually, however, the communication today goes over a network of nodes, where the individual messages are exchanged with the help of **routing** via intermediate nodes.

If all nodes in both client and server roles communicate directly with each other, it is called a **Peer-to-Peer** architecture. If some nodes are only in the client role and others are only in the server role, it is called a **Client/Server** architecture.

In order to let several nodes communicate with each other, without these having to know and address each other, one usually uses a central **Bus/Broker**. and a star topology.

If between source and target intermediate nodes are active, which act as **Proxy** in the communication and not only forward the network packets like a **Router**, one speaks of either a **(Forward) Proxy** or a **Reverse Proxy** situation. The former, if the proxy acts on the side of the source node, the latter, if the proxy acts as a proxy of the destination node.

In addition, a so-called **Virtual Private Network** can be established, in which a logical secure "overlay network" is placed over a physical network.

Questions

❓ With which **Network Architecture** can several nodes communicate with each other without these nodes having to know each other exactly?

❓ What do you call a computer node that acts on behalf of a target node?

Communication Architectures
(Patterns)

TECHNISCHE UNIVERSITÄT MÜNCHEN

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.
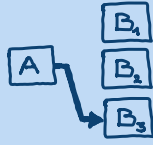
MCT — Multicast (one-to-many)

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.
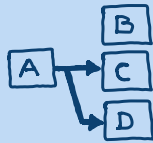
BCT — Broadcast (one-to-all)

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.
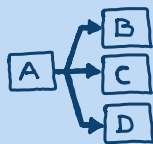
DGR — Datagram (Single Packet)

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).
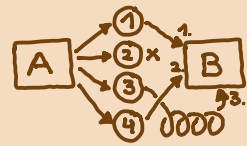
STR — Stream (Sequence of Packets)

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.
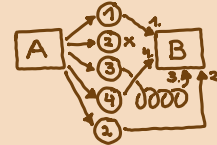
PLL — Pull (Request/Response, RPC)

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.
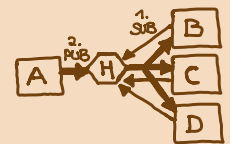
The **Communication Architectures** address the kind of communication between components. One distinguishes primarily four different kinds of message transmission: with **Unicast**, a source node sends to exactly one directly addressed target node. With **Anycast**, a source node sends to a group of potential destination nodes, but the message is delivered to one destination node in the group only.

With **Multicast**, a source node also sends to a group of target nodes, but the message is delivered to all target nodes in the group. With **Broadcast**, a source node sends to all reachable destination nodes without these particular destination nodes being known to the source node.

With the kind of messages, one differentiates two variants: with **Datagram**, each message consists of exactly one network packet, and when sending, no guarantees are given whether and in which order the messages will arrive at the destination node. In contrast, with **Stream**, a message consists of a sequence of network packets and different guarantees are given:

In case of packet congestion on intermediate nodes, the source of the **Stream** may be throttled. In case of packet loss, packets are resent. And one might get control over whether the packet will be delivered at most once, exactly once, or at least once at the destination node.

There are usually two modes of client/server communication: in **Pull** mode, the client sends a request, and the server sends a response. The server cannot proactively (without a prior request) send a message. In **Push** mode, the client sends a message in advance to the server to subscribe to certain types of messages. After that, the server can send a message to all subscribed clients at any time.

Usually, **Pull** is implemented via **Unicast/Anycast** and as a **Stream**, for example, in the HTTP protocol. On the other hand, **Push** is usually implemented via **Multicast/Broadcast** as a **Datagram**, for example, in the DHCP protocol.

Questions

❓ Which well known Web-protocol uses a communication based on **Unicast**, **Stream** and **Pull**?