TECHNISCHE UNIVERSITÄT MÜNCHEN

Software Engineering in der industriellen Praxis (SEIP)

Dr. Ralf S. Engelschall

ARCHITECTURE FUNDAMENTALS

Data Structure Architectures

(Patterns)

TECHNISCHE UNIVERSITÄT MÜNCHEN

Data Structure Types

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

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

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

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

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

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

Data Evolution Approaches

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

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

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

Operational Transformation (OT)
Modify data through applying journaled, domain-specific operational transformations.

Data Sharing Approaches

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

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

Data Store Types

Key-Value Store
Storage of values in an unordered manner, indexed and queried by key.
Redis, Riak, MemcacheD, RocksDB, LevelDB

Large-Object Store
Storage of unstructured binary-large object (BLOB) data and its associated meta-data, indexed and queried by unique id.
Minio, SeaweedFS, AWS S3

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

File-Tree Store
Storage of unstructured data as named files in a directory tree, indexed and queried by name path from root directory to leave file.
ZFS, XFS, UFS2, APFS

Graph Store
Storage of values as vertices and edges in a graph, both optionally referencing associated key/value pairs. Indexed and queried by key/value pairs and traversed by following edges.
Neo4J, OrientDB, ArangoDB

Document Store
Storage of structured "documents", indexed by id and key/value fields and queried by id and example documents.
MongoDB, CouchDB, RethinkDB

Relational/Table Store
Storage rows of fixed-size, typed value columns, indexed and queried by column values.
PostgreSQL, MariaDB, SQLite, H2, ORACLE DB, IBM DB2

Full-Text Store
Storage of unstructured text, indexed and queried by content words.
ElasticSearch, Solr, Groonga

Wide-Column Store
Distributed storage of rows of sparse (often untyped) value columns, indexed and queried by column values.
Cassandra, MonetDB, HBase, ScyllaDB

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

DataVault Store
Long-term historical storage of foreign, arbitrary relational data in a fixed schema of hubs, links and satellites, indexed and queried for analysis and reporting purposes.
DataVault 2.0

BlockChain Store
Storage of values in an unordered manner within information blocks which are cryptographically chained through their hash values and distributed in a peer-to-peer way.
Ethereum, Quorum, Tendermint, Hyperledger

The Software Architect distinguishes only 6 **Data Structure Types** for data elements: **Scalar** (e.g. 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) and **Graph** (unordered set of elements, each indexed by key or by following a link between elements). All complex specific data structures in practice, for the Software Architect, are only the combination of these 6 types.

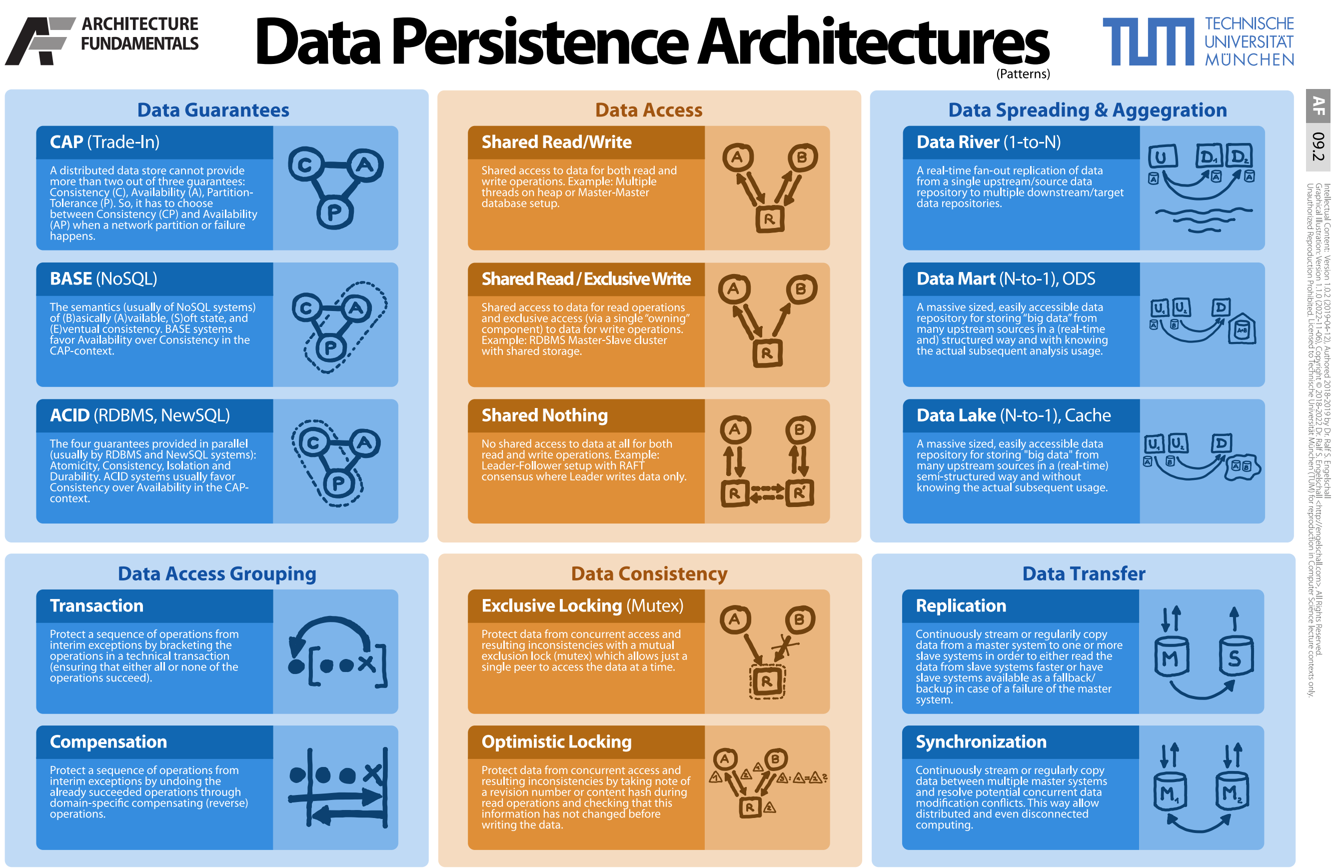
There are numerous **Data Evolution Approaches**, with which data can change over time: in the simplest case, **In-Place Editing**, data is simply changed directly. Access to previous states does not exist. If one wants to access previous states, one can use **Stacking Revisions**, in which the entire data set is copied before each change. So that the entire data record does not have to be copied, **Structural Difference** stores only a technical difference between the old and the new data records. Alternatively, with **Operational Transformation**, the technical change operations can be stored as a journal.

If such a journal is used also to keep replicas of the data sets up-to-date, one refers to it as **Event Sourcing**. If the journal is used as the protocol of so-called **Conflict-Free Replicated Data-Types (CRDT)**, instead of (unidirectional) replication, a (bidirectional) synchronization can be achieved. If several processes/ threads logically operate on copies, but physically on the same data sets, **Copy-on-Write** and **Reference Counting** can be used to achieve common access and the life cycle of the data sets can nevertheless be reasonably controlled.

For the storage of data in databases, there are numerous **Data Store Types**. These differ primarily in the type and flexibility of the data structure and the guarantees provided. The most common type is the **Relational/Table Store**. The most elegant type is the **Graph Store**. The most convenient is the **Document Store**.

Questions

? Name 3 **Data Evolution Approaches**, each of which allows to access the previous states of the data?



In the area of **Data Guarantees**, there are three main aspects: The **CAP** theorem addresses the so-called “trade-in”: In practice, one usually has to choose between Consistency + Partition-Tolerance (CP) or Availability + Partition-Tolerance (AP). Both at the same time is not possible. With **BASE** systems, AP is usually favored. For a traditional RDBMS with **ACID** guarantees, one usually favors CP.

With **Data Access Grouping** one knows about **Transaction** and **Compensation**. The former is a “technical bracket” that allows you to revert to the previous state in case of an error. The latter is a “domain-specific bracket,” where so-called compensation operations allow to “cancel” the earlier changes in order to regain a previous consistent state.

With the **Data Access** of two or more processes/threads on the same data one distinguishes between the approaches **Shared Read/Write** (all read and write the same data), **Shared Read/Exclusive Write** (all read and and only one writes the same data) and **Shared Nothing** (all read and write to the equal synchronized data).

With the **Data Consistency** one knows about **Exclusive Locking** (per time unit only one writes) and **Optimistic Locking** (all try to write, but recognize and resolve a conflict).

With **Data Spreading & Aggregation** one differentiates three kinds: with the **Data River** the data are replicated from a master system to many slave systems to achieve, among other things, a higher read performance. With the **Data Mart** (structured data) and **Data Lake** (semi-structured data), data is replicated from one master system to many slave systems in order to centrally report or cache the data.

With the **Data Transfer** we finally distinguish between the unidirectional and conflict-free **Replication** and the bidirectional and conflict-rich **Synchronization**.

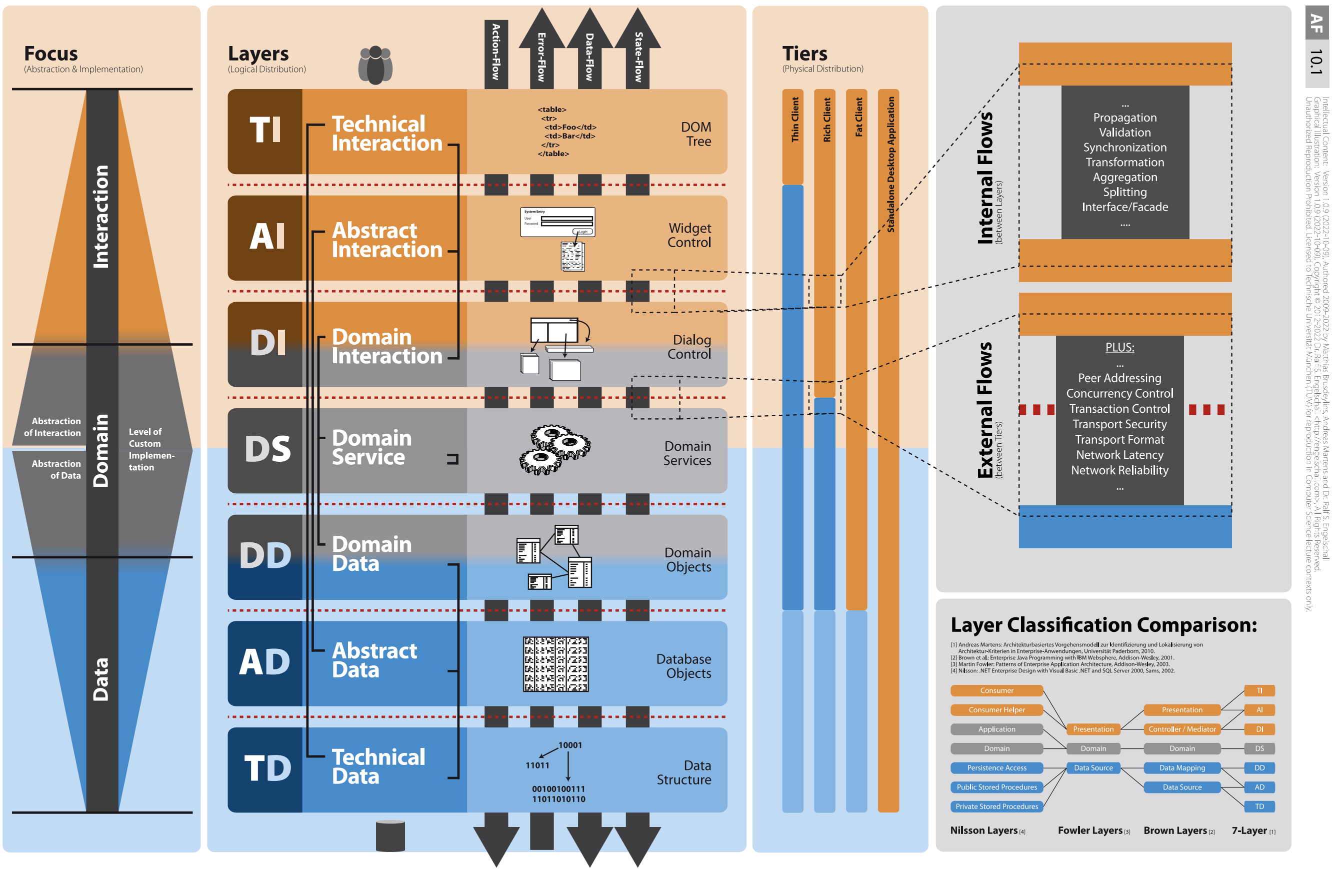
Questions

- What is the name of the approach in which data is replicated from a master system to many slave systems?

ARCHITECTURE FUNDAMENTALS

Application Reference Architecture

TECHNISCHE UNIVERSITÄT MÜNCHEN



In an application, it is possible to distinguish 7 logical layers, each grouped in two ways: on the one hand, there are the three sequential layer groups **Technical/ Abstract/Domain Interaction**, **Domain Service** and **Domain/Abstract/Technical Data**, on the other hand, there are the three nested layer groups **Technical Interaction/Data**, **Abstract Interaction/Data** and **Domain Interaction/Service/Data**.

In addition, one can distinguish 4 primary flows in an application: the **Action Flow** consequently runs from top to bottom only because all actions at the top are triggered by the user (or neighboring systems); the **Error Flow** consistently runs only in the opposite direction, i.e., from the bottom to the top, because errors, in the worst case, must be reported to the user; the (domain-specific) **Data Flow** and the (technical) **State Flow** run in both directions because data and states have to be persisted as well as displayed.

The abstraction of Interaction/Data in the layers increases from the top/bottom towards the middle, so most of the functional code of an application is written there. For the upper/lower layers, one usually massively relies on Open Source libraries/frameworks.

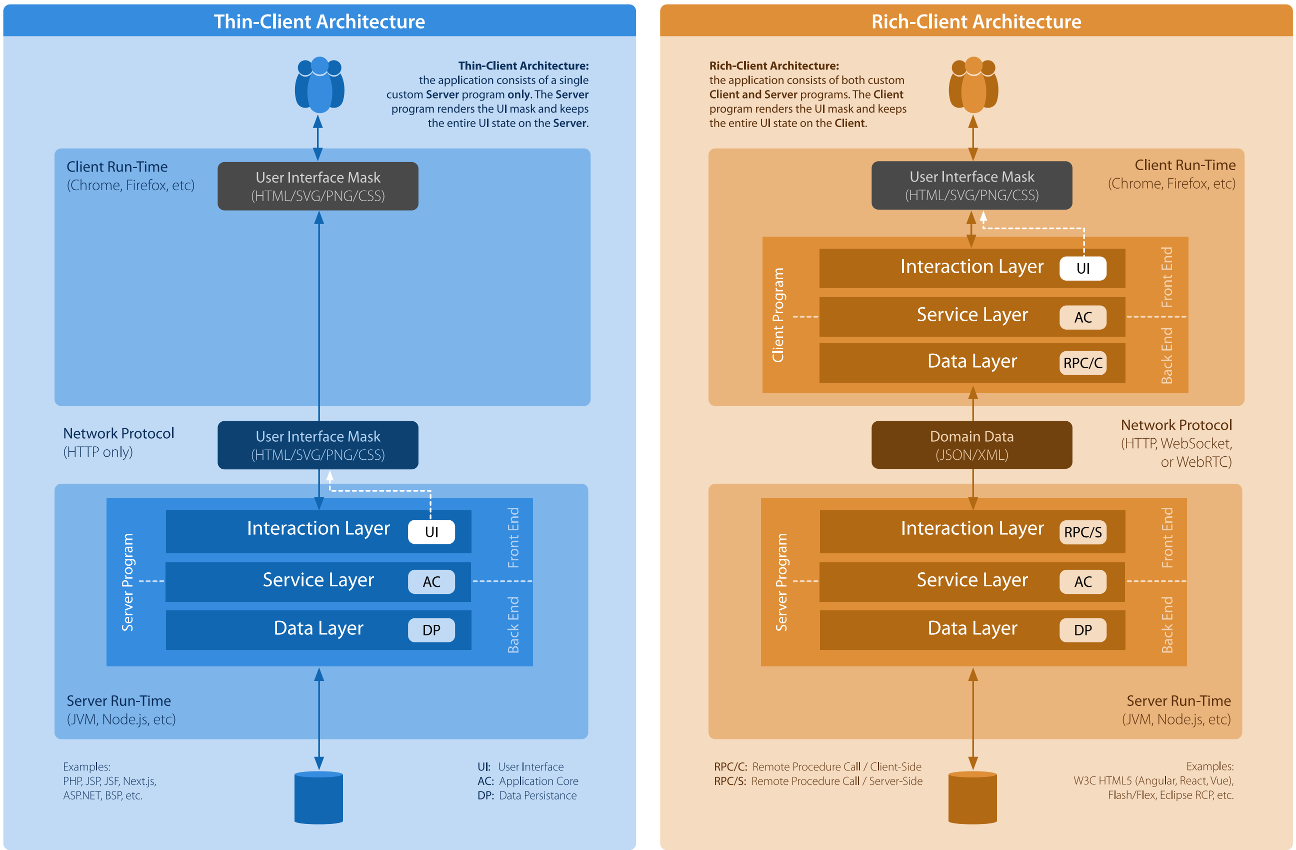
If instead of a logical cut (resulting in an **Internal Flow** between the layers) between two layers, one makes a physical cut (which then results in an **External Flow**), i.e., one distributes the application into single programs on different computers, then the resulting architecture is called according to the scope and responsibility of the client.

With **Thin Client**, only the **Technical Interaction** is offloaded to the client, while with **Rich Client** the entire user interface (i.e., all three layers **Technical/Abstract/ Domain Interaction**) is autonomously offloaded to the client (usually as a so-called “HTML5 Single-Page-Application”), with **Fat Client** there is no more associated server at all, and with the **Standalone** application, there is only one single program.

Questions

- What is the name of the application architecture in which the entire user interface runs autonomously on the client, while the server only provides purely functional services?
- What are the web applications named that implement a **Rich Client** architecture?

Client-Server Architecture



In the **Thin-Client Architecture**, the application consists of a single custom Server program only. This Server program renders the User Interface Mask and keeps the entire state of the User Interface on the Server.

The advantage of this architecture is that the application can be updated very easily. The disadvantage of this architecture is that the user interface reacts sluggishly, and the state of the user interfaces of all clients must be kept on the server, which can make the server a bottleneck.

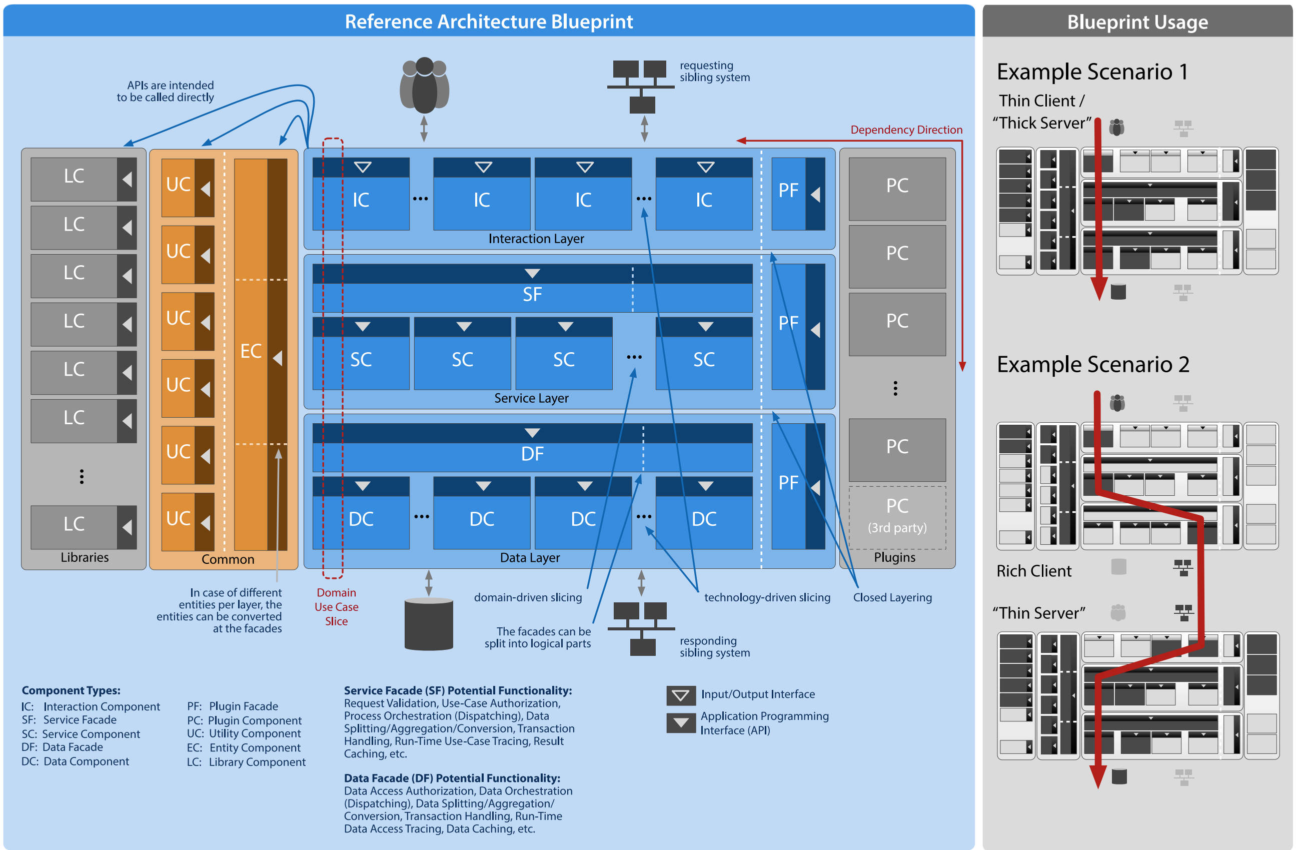
In the **Rich-Client Architecture**, the application consists of both custom Client and Server programs. The Client program renders the User Interface mask and keeps the entire state of the User Interface on the Client.

The advantage of this architecture is that the user interface is highly responsive, only domain-specific data has to be exchanged between the client and the server and the server server becomes less of a bottleneck. The disadvantage of this architecture is that, if necessary, the client has to be updated explicitly via an installation procedure.

Questions

- With which Client Architecture does the User Interface offer the higher responsiveness?

Information System Architecture



A (business) Information System usually follows a stringent component-based reference architecture. This is represented "full blown" and can be arbitrarily "slimmed down."

First, this reference architecture consists of 3 substantial Layers: the **Interaction Layer** with the (technically cut) components, which provide the I/O-based interfaces to the user (User Interface) and/or requesting neighboring systems (via Network interface), the **Service Layer** with the (domain-specifically cut) service components (also called Application Core) and the **Data Layer** with the (technically cut) components that provide the connection to the own database and/or neighboring systems to be queried.

Note that the "docking position" of a neighboring system depends on its roles: if it requests, it docks to the Interaction Layer; if it is queried, it docks at the Data Layer. If it happens to have both roles, it docks twice. The other view is that both the user and the database can be understood as special "neighbor systems."

To connect the N **Interaction Components** (IC) with M **Service Components** (SC) a decoupling **Service Facade** (SF) is usually inserted. For the same reason, there is usually also a **Data Facade**.

The Data Model is offloaded to common **Entity Components** (EC). Together with possibly shared code, both live in one **Common Slice**. Libraries and Plugins are also offloaded to separate slices, but there are two major differences: Libraries are passive and provide their functionality to the application via their interfaces. Plugins are active and control the application in that they hook into the application via Service Provider Interfaces (SPI) of the **Plugin Facades**.

In the application, there has to be only exactly one **Dependency Direction** so that the application (in the opposite direction of the dependencies) can be built cleanly. The reference architecture is usually also instantiated twice in order to design both a Rich Client and an associated "Thin Server" from it.

Questions

- With which layer pattern can in an Information System the **Interaction Components** be decoupled from the **Service Components**?
- In which order are the components of an application built?

Reactive System Architecture

ARCHITECTURE FUNDAMENTALS

TECHNISCHE UNIVERSITÄT MÜNCHEN

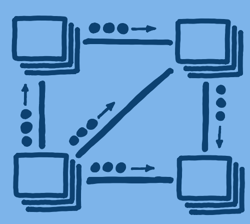
AF 10.4

Architecture & Systems

DEF Definition

Reactive System Architecture enables the realization of *Reactive Systems.*

Reactive Systems are in *subordinated interaction* with their *dominating environment*. They *continuously* process *endless data streams* as *small messages*, react at *any time* and respond within *tight time limits*. For this, they continuously *observe* their *environment* and adapt their *behaviour* to the current situation.



Demand & Deliverables

CTX Context

Real-time communication in the context of Digitization, Internet, Internet of Things (IoT), Systems of Engagement, Media and Analytics.

VAL Values

Non-blocking input/output data processing, fast responses within *tight time limits*, and *continuous availability* of the provided services.

REQ Requirements

Services are *elastic* and provide high scalability, and are *resilient* and provide high fault tolerance.

PRP Properties

Services run *fully autonomously, monitor themselves*, and *automatically adapt* to changes in the environment.

Principles

Stay Responsive Always respond in a timely manner.

Accept Uncertainty Build reliability despite unreliable foundations.

Embrace Failure Expect things to go wrong and design for resilience.

Assert Autonomy Design components that act independently and interact collaboratively.

Tailor Consistency Individualize consistency per component to balance availability and performance.

Decouple Time Process asynchronously to avoid coordination and waiting.

Decouple Space Create flexibility by embracing the network.

Handle Dynamics Continuously adapt to varying demand and resources.

Patterns & Paradigms

ARC Architecture

Microservices, Cloud-Native Architecture (CNA), Event-Driven Architecture (EDA).

COM Communication

Asynchronous Communication, Non-Blocking I/O, Sequence, Push, Backpressure, Quality of Service (QoS).

DAT Data

Semantical Event, Small Message, Endless Stream.

STY Style

Functional Programming, Asynchronous Programming.

EXE Execution

Parallelization, Concurrency, Actors, Threads, Thread-Pool, Event-Loop.

INF Infrastructure

Message Queue (MQ), Load Balancer, Reverse Proxy, Service Mesh, Virtual Private Network (VPN).

PRC Processing

Complex Event Processing (CEP), EAI Patterns, Stream Processing (map, flatMap, filter, reduce), Event Sourcing.

ASY Asynchronism

Callback, Promise/Future, Observable, Publish & Subscribe.

Reactive System Architecture enables the realization of **Reactive Systems**. Reactive Systems are in **subordinated interaction** with their **dominating environment**. They **continuously process endless data streams** as **small messages**, react at **any time** and respond within **tight time limits**. For this, they continuously **observe** their **environment** and **adapt** their **behaviour** to the current situation.

Reactive Systems are primarily used in the context of real-time communication where services are provided, which have to be **elastic** and provide high scalability, and which have to be **reliable** and provide high fault tolerance.

Questions

- Which two essential requirements do Reactive Systems fulfill?
- What characterizes Reactive Systems in respect to their data processing?