Architecture of Distributed Systems - Summary

1. Introduction to Distributed Systems

A distributed system can be defined as *'a collection of autonomous computing elements that appears to its users as a single coherence system'*. Distributed systems several _main characteristics_: they have no global notion of time or state, consist of heterogeneous resources, use network communication and can have their components fail independently *(i.e. without making the system fail)*. **Middleware** provides a layer between the operating system and the exposed interface that implements common functionality in a distributed system. The _basic ingredients_ of distributed systems are processing elements, a communication subsystem (or network) and software. The _design goals_ of distributed systems include the following

- provide the ability to **share resources**;
- provide **distribution transparency**;
- provide **scalability**;
- make the system more **open**; that is, allow it to be used, extended or integrated with by other components/systems.

There are 7 different types of distribution transparency:

1. **Access transparency**, which hides differences in data representation and the way in which objects can be accessed.
2. **Location transparency**, which hides the physical location of an object in the system[1].
3. **Relocation transparency**, which hides the fact that objects are moved to different (physical) locations in the system.
4. **Migration transparency**, which hides the fact that users of the system may move to different locations[2].
5. **Replication transparency**, which hides the fact that several copies/replicas of a resource exist, or that a process is executed several times to provide a failover mechanism.
6. **Concurrency transparency**, which hides that multiple independent users may share the same object/resource.
7. **Failure transparency**, which hides failure and recovery from the user; that is, the user or application should not notice that a part of the system does not work properly, and should also not notice the recovery from this failure.

Scalability can be measured along 3 different dimensions:

1. **Size scalability**, which describes that more users or resources can be added to the system without (noticeably) impacting performance;
2. **Geographical scalability**, which describes that users do not notice communication delays when users and resources in the system are far apart;
3. **Administrative scalability**, which describes that the system remains easy to manage even if it spans many independent administrative organizations.

There is also a distinction between vertical scalability and horizontal scalability. **Vertical scalability**, also known as _scaling up_, handles increased load by adding capacity to an existing physical machine (e.g. by adding memory, replacing CPUs, or changing network modules). **Horizontal scalability**, also known as _scaling out_, handles increased load by adding more machines to the system.

A system can be made in an open manner by taking the following into account:

- systems should conform to well-defined interfaces
- systems should easily interoperate
- systems should support portability of applications

---

[1] Naming can be used to provide location transparency. For example, a uniform resource locator (URL) provides no information *(at least, not directly)* on where the file/website behind that URL is hosted/located.

[2] For example, migration transparency can refer to the system not noticing that a user has moved to a different network/connection.

- systems should be easily extensible

Implementing openness requires policies and mechanisms. A **policy** typically decides what should be done, whereas a **mechanism** provides an implementation which provides basic facilities (while, ideally, leaving the choice of policy to the user). For example, a browser may provide facilities for storing documents *(a mechanism)*, while letting the user decide on which documents to store and for how long *(a policy)*.

There are several types of distributed systems:

- **High performance computing systems** include cluster computing, grid computing, and cloud computing systems;
- **Distributed information systems** allow for distributed transaction processing *(e.g. in a database)* and enterprise application integration;
- **Pervasive systems** include ubiquitous computing systems, mobile computing systems, and sensor networks.

There are 8 fallacies with respect to distributed computing:

1. The assumption that the network is reliable;
2. The assumption that the network is secure;
3. The assumption that the network is homogeneous;
4. The assumption that the topology does not change;
5. The assumption that the latency is zero;
6. The assumption that the bandwidth is infinite;
7. The assumption that the transport cost is zero;
8. The assumption that there is one administrator.

2. Introduction to architecture

System architectures serve several different purposes:

1. They may aid in <u>*communicating/understanding*</u> the system: that is, an architecture can be used to define the scope of the system and discuss it with stakeholders, developers and users/customers;
2. They allow for <u>*analyzing*</u> the system: that is, it becomes possible to evaluate candidate architectures on an abstract level and make design decisions based on this information;
3. They allow for <u>*synthesizing/constructing*</u> the system: that is, an architecture forms a floorplan/ blueprint of the overall structure and can serve as the basis for a detailed design by developers.

The IEEE standard 42010 defines several terms:

- **Architecting** is ‘the process of conceiving, defining, expressing, documenting, communicating, certifying proper implementation of, maintaining and improving an architecture throughout a system’s life cycle’;
- An **architecture** consists of ‘fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution’;
- An **architecture description (AD)** is a ‘work product used to express an architecture’;
- An **architecture framework** consists of ‘conventions, principles and practices for the description of architectures established within a specific domain of application and/or community of stakeholders’;
- The **environment** of a system consists of the ‘context determining the setting and circumstances of all influences upon a system’;
- The **stakeholders** of a system are individuals, teams, organizations (or classes thereof) holding concerns with respect to a system;
- A **concern** is an ‘interest in a system relevant to one or more of its stakeholders’;
- An **architecture view** is a ‘work product expressing the architecture of a system from the perspective of specific system concerns’;
- An **architecture viewpoint** is a ‘work product establishing the conventions for the construction, interpretation and use of architecture views to frame specific system concerns’;
- A **model kind** consists of ‘conventions for a type of modelling’, and can include data flow diagrams, class diagrams, Petri nets, balance sheets, organization charts and state transition models.

We have that an architecture rationale justifies an architecture decision, which may pertain to and/or raise concerns. Architecture decisions affect architecture description (AD) elements.
An **architecture description language** could be described as an architecture framework. An architecture description languages must identify stakeholders, concerns, and at least one model kind. Architecture description language do not provide requirements for viewpoints.

The stakeholders and their concerns together define the environment of the system. With respect to an architecture, stakeholders are responsible for communicating their concerns to the architect, reviewing the architectural description, and deciding on the acceptance of the architecture *(as well as other issues concerning the system)*.

The Rozanski-Woods book classifies stakeholders into the following categories:

- **Acquirers**, who are overseeing the procurement of the system;
- **Assessors**, who are overseeing the conformance to standards and legal regulations;
- **Communicators**, who explain the system via documentation and training;
- **Developers**, who construct and deploy the system from its specifications;
- **Maintainers**, who manage the evolution of the system once it is operational;
- **Suppliers**, who provide the hardware/software platform on which the system will run;
- **Support staff**, who assist users in making use of the running system;
- **System administrators**, who run the system once it has been deployed;
- **Testers**, who check whether the system meets its (functional and extra-functional) specifications;
- **Users**, who define the system's functionality and use it once it is running.

The Rozanski-Woods book considers all systems to have an architecture, albeit not necessarily an explicit one.

An architectural **view** is the description of the whole system architecture from the perspective of a set of concerns. On the other hand, an architectural **viewpoint** defines the rule for constructing, interpreting and using a view. When interpreting a view as a map, a viewpoint can be seen as a legend for that map.

Kruchten defines 4+1 view(point)s[3]:

1. **Logical** viewpoint, where the main concern is the *use* of the system and the *associated qualities* of the system. In particular, this viewpoint describes the externally visible structure and functionality of the system. **Users** form an important stakeholder in this viewpoint.
2. **Development** viewpoint, where the main concerns are *implementing and modifying* the system. This can encompass aspects such as the system's decomposition into subsystems and the organization of the system into files, components and modules. **Developers** form an important stakeholder in this viewpoint.
3. **Process** viewpoint, where the main concerns are *performance aspects* and the *interactions between parts* of the system. Aspects such as which units of deployment (e.g. programs, components) and units of concurrency (e.g. processes, threads) to use are relevant in this viewpoint. **System integrators** form an important stakeholder in this viewpoint.
4. **Deployment** viewpoint, where the main concerns are related to *installing, deploying and realizing* the system. This requires consideration of which computers, networks, infrastructure, protocols and distribution to use, as well as how to map software to hardware. **System engineers** form an important stakeholder in this viewpoint.
5. Scenarios, which discuss use cases for the system. This viewpoint is considered out of scope for this course.

UML diagrams can be used to provide views, although the correspondence to the viewpoints described by Kruchten is not *exact*. Classes, interfaces and collaborations are relevant to the logical viewpoint; components are relevant to the development viewpoint.

[3] Note that, although Kruchten defines these as views, they are viewpoints according to the definition given in the IEEE standard.

Extra-functional requirements, which often specify emergent system properties, are addressed through the architecture. Many extra-functional requirements have a name ending in '-ity'. Examples of extra-functional requirements include:

Accessibility	Responsiveness	Extensibility	Modularity
Understandability	Scalability	Modifiability	Reusability
Usability	Fault tolerance	Consistency	Availability
Generality	Timeliness	Openness	Confidentiality
Simplicity	CPU utilization	Interoperability	Integrity
Portability	Latency	Completeness	Maintainability
Accuracy	Throughput	Correctness	Reliability
Efficiency	Concurrency	Testability	Security
Footprint	Flexibility	Coherence	Affordability

An architecture defines a class of acceptable solutions and is technology-independent, whereas a design defines a single specific solution and conforms to an architecture.

A design process iteratively translates a design problem into (a model of) a solution. Building blocks are different for each level in the design process. A design process consists of four elements:

1. Domain analysis, which consists of increasing the knowledge on the problem, making models, and soliciting feedback from stakeholders;
2. Applying strategies, which consists of hierarchical decomposition (in a top-down or bottom-up manner), applying patterns, styles and tactics, as well as generating alternatives;
3. Synthesis, which consists of evaluating and choosing alternatives, and combining partial solutions;
4. Verification, which consists of checking whether the system adheres to its specification.

3. Architectural styles

An **architectural style** is formulated in terms of components, and describes how these components are connected to each other, how they exchange data, and how they are jointly configured into a system. Architectural styles are described using the following general concepts:

- Components, which are modular units with well-defined interfaces;
- Interfaces, which are shared boundaries across which two or more separate components of a system exchange information;
- **Connectors**, which are mechanisms that mediate communication, coordination, or cooperation among components. Connectors allow for the flow of *control* and *data* between components.

The appropriate architectural style to use can be chosen based on extra-functional requirements such as costs, scalability, performance, reliability, fault tolerance, maintainability, usability, and reusability. The relative importance of viewpoints often determines the choice of architectural styles.

In this course, architectural styles are described in terms of the following elements:

- Motivation
- Vocabulary
- Structure
- Typical behaviour
- Rules
- Weaknesses
- Examples

Several architectural styles are important to know in this course:

- The **layered style** is **motivated** by the desire to separate concerns, limit dependencies, and allow for independent development and evolution (of separate layers). A system in the layered style is **structured** into independent layers of abstraction, which are determined at development time. Three types of layering are distinguished:
  - **Strict layering** requires downcalls to go exactly one layer down and upcalls to go exactly one layer up;
  - **Non-strict layering** allows for bridging calls, which can bypass one or more layers;
  - **Sidecar layering**, which is used for modules addressing crosscutting concerns.

Typically, layer N calls upon the API of layer $N-1$; layer $N-1$ is then regarded by layer N as the single service provider of all lower functionality. Furthermore, layer N can register call-back routines with layer $N-1$; upon occurrence of an event, layer $N-1$ invokes the call-back routine referred to by the **handle**.
The **rules** of the layered style state that layer N may only use lower layers (not vice versa) (and, in strict layering layer N may only use layer $N-1$), that no 'use'-restrictions apply for modules within a layer, and that control flow may be both synchronous and asynchronous.
The main **weakness** of the layered style is that each layer adds additional complexity; this may lead to additional code and longer calling sequences, in addition to the style becoming an anti-pattern when used in a setting that is too simple.

Several definitions are important for the layered style. A **service** is *'a contractually specified overall functionality (semantics) of an entity'*. A service interface (API) provides actions and *(possibly autonomous[4])* responses that make the service available, in addition to a specification describing effects on state variables and parameters, the results of an service, rules regarding how and in what sequence to call services, and the functional and non-functional properties of sequences of calls. A **protocol** is *'a formal set of rules that dictates how information exchange, as well as interaction between entities should take place'*. A protocol's rules specify a message format, a finite state machine, timing constraints and possibly other quality properties. A realization of a protocol is said to rely on a **carrier** if it relies on another service and its corresponding protocol. A **binding** provides the set of rules that specify how a protocol is mapped onto a carrier, and generally refers to establishing a relation between a reference and a referenced object.

- The **client-server style** is **motivated** by the desires to achieve a separation of concerns (between user interaction and application logic), to share localised resources, to protect and manage content, to delay binding, and to decrease dependencies. A system in the client-server style is dynamically **structured**; that is, connections only arise at run-time, in a many-to-one fashion. The server location could already have been determined at deployment time, and it should be noted that a server may be a distributed entity. **Typically**, clients find the access point through a process known as **server discovery**. After connecting to the access point, communication takes the form of a series of synchronous request-reply pairs, which possibly occur as part of a session and make a server stateful.
  The **rules** of the client-server style state that servers are passive and clients are active. This means that:
  - Servers provide a service upon request from clients. They do not know their clients. Furthermore, servers handle data access and integrity.
  - Clients initiate actions; in particular, they discover servers and issue requests.

  In addition, the **rules** of the client-server style state that there is no connection among clients, and that a server maintains no or limited state per client.
  The client-server style has several **weaknesses**:
  - A single server may become a performance bottleneck. Note that, even when the server is replicated, there is still only a single **access point** to the server and its replicas (which can then become the bottleneck).
  - A server can be a single point of failure.
  - Stateful servers scale poorly.

  Logically, the client-server style is organised into three layers: the user interface layer, the application layer, and the database layer. These layers have to be divided between two physical tiers (client and server), and this can be done at any point in the layers. Placing only the user interface layer *(or part of it)* on the client leads to what is known as a **thin client**. Placing the application *(or part of it)* on the client leads to what is known as a **fat client**. It is also possible to have the client contain the database engine *(and let the fileserver only be responsible for storing the database tables)*.
- The **peer-to-peer style** is motivated by the desires to share resources and content, cooperate in communities, have symmetry in roles, increase concurrency, and provide horizontal scalability.

[4] That is, 'responses' can also be given through events or callbacks.

- REpresentational State Transfer (REST) style
- Publish/subscribe style

In cloud computing, there is a distinction between four layers:
1. The hardware layer, which contains processors, routers, power, and cooling systems. Customers normally never get to see these.
2. The infrastructure layer, which deploys virtualization techniques and concerns the allocation and management of virtual storage devices and virtual servers.
3. The platform layer, which provides higher-level abstractions for storage and such. For example: the Amazon S3 storage system offers an API for files to be organized and stored in so-called buckets.
4. The application layer, which contains actual applications such as office suites and is comparable to the suite of apps shipped with operating systems.

Edge servers can be used as a sort of intermediary server between devices on the local network and the cloud. Edge servers are particularly useful in the case of power-constrained devices (as in the internet of things).

The life cycle of a product or system is the series of stages it goes through from inception to decommissioning.

4. Interaction styles

Interaction styles pertain to communication aspects in the views, which are typically addressed in the process view. In layered protocols, like the OSI reference model, each layer provides an interface to layers above itself, and uses interfaces from layers below itself. The OSI reference model distinguishes 7 layers:
1. Physical layer, which sends bits on the medium;
2. Data link layer, which detects and correct errors in frames, and delivers frames in a one-hop neighborhood;
3. Network layer, which sends packets from sender to receiver machines using multi-hop routing;
4. Transport layer, which breaks messages into packets, provides delivery guarantees and multiplexes ports;
5. Session layer, which provides dialog control and synchronization facilities (in the form of checkpoints);
6. Presentation layer, which structures information and attaches meaning (semantics, e.g. names, addresses, amount of money);
7. Application layer, which provides network applications (including standard ones like email and file transfer).

The n-th level unit of communication is the payload of the $n-1$-th level unit of communication. In most distributed systems, the transport layer provides the actual communication facilities. Standard transport layer protocols are **TCP**, which is connection-oriented, reliable, and stream-oriented, and **UDP**, which provides unreliable (best-effort) datagram communication.

Coordination between systems can use two different types of coupling: **referential coupling** and **temporal coupling**. Two processes are referentially coupled if they explicitly need to have an explicit reference to each other *(e.g. an identifier or name)* for them to communicate. Two processes are temporally coupled if both of them have to be up and running *(at the same time)* for them to communicate. Typical examples of communication involving these types of coupling are given in the table below:

	Temporally coupled	Temporally decoupled
Referentially coupled	Direct communication	Mailbox
Referentially decoupled	Event-based communication	Shared data space

A **middleware layer** provides common services and protocols that can be used by many different applications. This typically includes:
- A rich set of communication protocols;

- A method for (un)marshaling data;
- Naming protocols, which allow for easy sharing of resources;
- Security protocols to secure communication;
- Scaling mechanisms, e.g. for replication and caching.

Middleware attempts to solve what is known as the **hourglass problem**. The hourglass problem states that connecting N applications through M different physical layers (which means that $N \times M$ different types of connections are made) can be done using only $N + M$ interfaces by connecting everything through a common transport protocol, which then provides interoperability.

There are two kinds of interaction patterns for middleware:
1. In the first pattern, middleware only provides a communication service *(in the form of message passing)*. In this pattern, each application maintains its own specific protocol, and logic is not shared.
2. In the second pattern, middleware provides advanced services, in the form of an application-independent protocol between middleware layers. This allows standard functions to be shared, and makes direct access to the remote service possible. It also supports a thin-client approach.

A middleware system can be described as consisting of four different layers:
1. Hardware layer, which uses physical and link-level protocols;
2. Operating system layer, which uses a host-to-host protocol;
3. Middleware layer, which uses a middleware protocol;
4. Application layer, which uses an application protocol.

Communication can be divided into several different types:
- A distinction can be made between **transient communication** and **persistent communication**. In transient communication, a server discards the message when it cannot be delivered to the next server or the receiver. In persistent communication, a server stores a message for as long as it takes to deliver said message.
- A distinction can also be made between **synchronous communication** and **asynchronous communication**. In synchronous communication, the sender is blocked until its request is known to be accepted. In asynchronous communication, the sender continues immediately after it has submitted its message for transmission.

These different types of communication allow for synchronizing between the client and server at different times.
- Synchronization can be done at request Submission (SaS), which means the client can continue as soon as its request has been submitted for transmission.
- Synchronization can be done at request Delivery (SaD), which means the client blocks until it receives confirmation that the server has received the message.
- Synchronization can be done after the request has been Processed by the server (SaP), which means the client blocks until it receives confirmation that the server has processed the message.

Remote procedure calls (RPCs) allow a client to call procedures available on (an object on) a server. The architecture of an RPC consists of three main elements on each (client and server) machine: the operating system, the process, and the **stub**. When a client wants to call a procedure on a remote server, it calls the ***client stub***. The client stub then builds a message and calls the local OS to send the message to the remote OS over the network. The remote OS then gives the message to the ***server stub***, which unpacks the parameters, and calls the procedure locally. When the procedure returns from the ***server process***, the stub builds another message, which leads to a call to the ***server OS*** to send this message to the ***client OS***. The client OS then gives the message to the stub, which then unpacks the result and returns it to the ***client process***.

Several extra-functional requirements form concerns for RPCs: when many clients are calling the server, this can lead to *scalability* issues. Furthermore, an increased dependence on the availability of the server may lead to *reliability* issues. Finally, if all procedures are called separately *(i.e. they are not bundled in one message)*, then the additional round-trip times may lead to *performance* problems.

Realizing an RPC implementation requires several aspects to be taken into account, such as how to marshal data, a protocol definition, how to deal with parameters, how RPCs should be included in a program, how and when to bind client and server, server discovery, semantics under (partial) failures, and synchronization.
Marshaling is typically required because the representation of number, characters and other data items on machines may differ. A distinction is made between serialization and marshaling.

- **Serialization** is *'the activity by which the state of an object is converted into a byte stream in such a way that the byte stream can be converted back into a copy of the object'*.
- **Marshaling** is *'the activity by which a stub converts local application data into network data XDR (eXternal Data Representation) and packages the network data into packets for transmission'*. In other words, *'marshaling is the more general process of converting data in one format (e.g. objects in memory) into another format, which can be either binary or text-based'*.

Parameters in RPCs must be specified; that is, there should be a known message format and data structure representation (which involves e.g. how characters and numbers are encoded). The sequence in which messages are exchanged, and the transport protocol being used are also relevant. One typical issue with RPCs is that passing parameters cannot be done by simply passing a reference, as the client and server do not share a (memory) address space. Alternatives consist of either passing copies of the parameters around or allowing remote references *(the latter of which introduces further complexities, although they may be necessary for RPCs to be useful)*.

There are several variants of RPC:

- Traditional RPCs require waiting for the result.
- **Asynchronous RPC**s only require waiting for a confirmation that the request was accepted. Asynchronous RPCs are useful when there is no result that needs to be returned.
- **Deferred Synchronous RPC**s allow for doing something useful while waiting. In effect, a deferred synchronous RPC consists of two asynchronous RPCs, where the first response confirms the request has been received, and the second returns the response to the request.
- **Multicast RPC**s have the client send a request to multiple servers. The client may wait for all servers to return *(although, in principle, it should also be possible to wait for the first server to return a response)*.

In the DCE RPC framework, the client and client stub C code are first compiled into object files, which are then combined into a client binary through a linker. The framework uses discovery and registration services at runtime.

To handle partial failures of RPCs, it is needed to create **failure models**. There are several such models:

1. Requests can be lost or duplicated;
2. Responses can be lost or duplicated;
3. Servers may crash;
4. Clients may crash.

   (We generally assume the environment to not be malicious.)

RPC failures can be handled in the following manners:

1. failures can be detected via timeouts and explicit acknowledgements;
2. semantics can be defined explicitly; this can specify that execution should take place:
   1. at most once,
   2. at least once,
   3. exactly once *(which is ideal, but difficult to implement)*
3. requests can be made idempotent *(that is, the effect of executing the request multiple times is the same as that of executing it once)*;
4. timestamping or versioning can be used to prevent the server from re-executing a call,
5. orphan[5] tasks can be exterminated by dividing time into epochs and associating these epochs with client reincarnations.

[5] An orphan task is a task running *(on the server)* for a client which has crashed.

Implementing remote method invocation involves several plumbing elements:

- A **proxy** is a component that acts on behalf of another component, implementing the same interface. Proxies are capable of implementing filtering policies and sometimes caching. Typically, a proxy refers to a client-side entity; a **reverse proxy** is placed at the server side.
- In RPCs and RMIs (remote method invocations), a **client stub** transparently implements an interface for a remote object. It is also responsible for the messaging (and hence sometimes called a proxy).
- A **server stub** (or **skeleton**) transparently performs the calls of a client stub and handles the messaging.
- A remote object is a distributed object whose state is not distributed.
- An **adapter** is a component that relays calls to an object interface and manages it.
- A **broker** is a component that handles and translates calls/messages between two or more parties, and manages the binding between references and objects. (This binding can be dynamic, based on interface inspection. This dynamic binding becomes visible by explicit invocation (instead of a transparent method call).)

Remote method invocations are very similar to remote procedure calls, although there are several differences. In particular, in RMIs, an object represents a context (state) together with operations on it. RMIs provide a clear separation between interface and implementation, where the actual implementation is hidden (and can be implemented in any programming language). In RMIs, references to objects occur naturally in a program.
RMIs pass local objects by value/result, and pass remote objects by reference.

RPCs and RMIs are tightly coupled synchronous mechanisms; that is, they exhibit functional, temporal and referential coupling *(respectively: as the caller has an interest in the result, the caller blocks until reply and holds a reference to the callee)*. **Message Oriented Middleware** aims at high-level persistent asynchronous communication, and archives this by having processes send messages to each other, which can be queued and do not require the sender to wait for a reply. Middleware is used to ensure fault tolerance.
The TCP service API specifies 8 functions:

Operation	**Used on**	**Description**
socket	Client & Server	Create a new communication end point
bind	Server	Attach a local address to a socket
listen	Server	Tell operating system what the maximum number of pending connection requests should be
accept	Server	Block caller until a connection request arrives
connect	Client	Actively attempt to establish a connection
send	Client & Server	Send some data over the connection
receive	Client & Server	Receive some data over the connection
close	Client & Server	Release the connection

ZeroMQ provides a higher level of expression by pairing sockets: it creates one socket for sending messages at process P and one socket for receiving messages at process Q. All communication in ZeroMQ is asynchronous.

There are three patterns for using sockets: request-reply, publish-subscribe and pipeline. Queue-based messaging can have any combination of sender and receiver running or being passive when sending messages. Message queuing systems provide asynchronous persistent communication through support of middleware level **queues**, which correspond to buffers at communication servers. Four primitive operations are supported by queues:

Primitive	Meaning
Put	Append a message to a specified queue
Get	Block until the specified queue is nonempty, and remove the first message
Poll	Check the specified queue for messages, and remove the first. Never block
Notify	Install a handler to be called when a message is put into the specified queue

A queue manager is a process or library that puts messages only into a *local* queue. As getting messages is only possible by extracting them from a local queue, queue managers need to route messages. Queuing systems may involve routers.

Message queuing (MQ) systems assume a common messaging protocol to be present; that is, all applications are assumed to agree on a message format. **Brokers** handle application heterogeneity in a message queuing system by transforming incoming messages into the target format. Brokers typically act as an application gateway and may provide subject-based routing capabilities.

In **stream-oriented communication**, such as media streaming, temporal relationships between data items play a crucial role. There are two transmission modes:

- **Synchronous transmission mode** bounds the end-to-end delay (also known as **latency**);
- **Isochronous transmission mode** bounds the inter-packet delay (also known as **jitter**).

Stream-oriented communication should occur softly in real-time; that is, when a packet is missed, this should lead to a graceful reduction in quality of the service (without blocking). Quality of service in a stream-oriented communication is not discrete (i.e. either success or failure), but can be expressed as a range of qualities for which a choice can dynamically be made.
There are several techniques/tactics to enhance quality of service:

- **buffering**, which smooths the variation in delay;
- **priority routing**, which reduces both latency and jitter;
- flow control, which applies to both the route and the rate of the stream;
- resolution and quantization scaling, which effectively means that the quality of the streamed items are adjusted, and can improve throughput;
- using quality modes, where the signal is divided into multiple streams[6], and the mode decided which streams are sent.

5. Naming

Service discovery describes the process of making two parties meet one another (e.g. for communication). This involves tasks such as advertisement, propagation, matching and evaluation. To find the **access point** (AP) of a server, the client can either directly contact the known access point of the server, or contact a known access point of a mediator to receive the access point of the service.
A server may register itself at a repository, after which the client can contact the repository to obtain the access point. An alternative mechanism consists of using a so-called *superserver*, which listens to many access points and hands over the request to the appropriate server's access point.
The goal of a cluster is typically to improve performance, reliability and availability. Building a cluster introduces several problems: its use should be transparent *(towards the client)*, requests should be dispatched from the logical switch *(which handles incoming requests)* to an appropriate application server, and security and management should not become too complicated.
It should be noted that having the first tier handle all communication from/to the cluster may lead to a bottleneck. This could be solved using TCP handoff, where the switch hands off the connection to a server (and then stops being involved in it).

[6] One way (not mentioned in the slides) of doing this would be to separate the audio and images of a video stream into separate streams; this is often done nowadays.

To prevent the front end from getting overloaded when requests are dispatched, special measures may be needed. One method is to use **application-aware distribution**; in this method, the front end reads the content of the request and then selects the best server. Another method uses **transport-layer switching**, in which the front end simply passes the TCP request to one of the servers, taking some performance metrics into account.
An alternative method of server discovery is to use multicasting/broadcasting, as in e.g. mDNS or DHCP. Advantages include this mechanism being fully distributed, not requiring centralized state maintenance and requiring zero configuration. Its main disadvantage is its limited scope, i.e. it can only reach the scope of the multicast/broadcast network. Other disadvantages include the requirement for each node to implement the entire protocol and store the data.

A **name** serves as a reference to an entity. Names are organized into a **name space**. There are several motivations for the use of names:
- they make it easy to *identify* entities;
- they allow *binding* to be *delayed*, providing migration transparency and replication transparency;
- they can be *looked up*, providing location transparency (name serves as a means to locate an entity);
- they are *human-friendly*.

Entities are the collection of 'things' that need to be referred to. **Identifiers** form a namespace for entities. An identifier is said to be a **true identifier**, if it has the following properties:
1. each identifier refers to at most one entity;
2. each entity is referred to by at most one identifier;
3. an identifier always refers to the same entity, i.e. it is never reused.

An **access point** represents a place where an entity can be accessed. An entity can have several access points, and can change its access point(s) over time *(e.g. in the case of mobile entities)*.
Addresses form a name space for access points. The address of an access point is unique, although the addresses of access points may change over time and can be reused.
A **name system** maintains a name-to-address binding. A **name service** answers queries for attributes *(typically, the address)* of a named entity. A name server may only answer queries for part of the name space; more specifically, for the part that belongs to its domain of authority. However, a name server may invoke other servers to answer a given query.

A **name space** is a set of valid names. There are two methods of generating/validating names. The first one is **flat naming**, where a name is an unstructured, meaningless sequence of bits, bytes or characters. **Structured naming** can define a hierarchy of names (where a distinction can be made between relative names (which are relative with respect to a given context) and absolute names, which are complete) or use prefixes to embed a set of names into a larger set in a unique manner. Hierarchical name spaces have the advantages of being unbounded in size and allowing for better management. Their main disadvantage is that it may take longer to realize that a given name is invalid *(as this requires partially going down the hierarchy)*.
A **binding** is the relationship between a reference (name) and a referred object (another name or an access point). Bindings can be symbolic, when the referred object is a similar type of reference. Aliasing is the phenomenon where there are several names for the same object.
Resolution is the process in which, given a name, the object it refers to is determined. There are two types of resolution: in **structured resolution**, the structure and level in a naming hierarchy need to be followed, whereas **flat resolution**, which is used in flat naming, does not support a hierarchy. A **closure mechanism** describes where and how to start resolution. For example, a closure mechanism may indicate the initial node in a name space, or what type of name (e.g. a phone number) needs to be resolved.

Generally, there are four types of naming:
1. **Flat naming**, where identifiers are simply random bit strings;
2. **Structured-based naming**, where names are composed from simple, human-readable names, as in e.g. a tree;
3. **Attribute-based naming**, where an entity is associated with a collection of attributes;
4. **Named-data networking**, where one can get a copy of an attribute to access it locally when needed.

Flat names can be resolved by searching. Searching can take several different forms: broadcasting, multicasting, or doing a table lookup (possibly involving hashes or trees).

Chord is a form of distributed storage, which implements a **distributed hash table** (DHT). Chord has a flat name space of m-bit identifiers, a set of nodes N, and a set of other entities E. An entity with key k falls under the jurisdiction of the node with the smallest identifier id such that $id \geq k$. This node is also called the successor of k, known as $succ(k)$. Each node p maintains a **finger table** $FT_p[]$ with at most m entries, which are defined as $FT_p[i] = succ\left(p + 2^{i-1}\right)$. This means that the i-th entry of the finger table of node p points to the first node succeeding p by at least 2^{i-1}. To look up a key k, the node p forwards the request to the node q with index j in p's finger table satisfying $q = FT_p\left[j\right] \leq k < FT_p\left[j+1\right]$. As an exception, when $p < k < FT_p\left[1\right]$, the request is also forwarded to $q = FT_p\left[1\right]$. Distributed location lookup can be implemented using linear search, where a node p stores the addresses of $succ\left(p+1\right)$ and $pred\left(p\right)$. Alternatively, as is done in the case of Chord, distributed location lookup is implemented using 'binary' search, in which a node p stores addresses of nodes that take large steps *(chords)* along the circle. This uses a routing table with entries for $succ\left(p + 2^{i-1}\right)$, for $1 \leq i \leq m$. This routing table is known as the finger table in Chord. Issues may arise when inserting entities, or, for a more difficult case, when inserting nodes. Inserting a node requires updating tables in the distributed hash table and relocating entities.
One method to simplify the implementation of mobile nodes is to leave so-called 'follow-me' messages at each node. These point to the new location of a given entity and can be implemented through middleware. As a downside, they are somewhat sensitive to broken links and long chains of 'follow-me's. These downsides can be accommodated for through pointer fusion or by updating the first indirection/follow-me. An implementation of 'follow-me's can be obtained through stubs which replace the mobile object and forward invocations to the new location of the object. In pointer fusion, the final stub responds with a reference to the object's new location (after which the client updates its location), although it must be noted that this is not transparent and requires garbage collection. Similar mechanisms are used in mobile IP.

Structured names can be resolved by interpreting the naming structure as a directed acyclic graph. Leaf nodes in such a graph provide the access point (or its address) for a given name, whereas internal/directory nodes provide an access point for a directory structure. Edge labels in such a graph form partial names.
Hierarchical name spaces are used in many different systems, including file systems, IP addresses, ARP, DNS, and for FTP. In a file system, edges in a graph contain directory names, whereas the nodes contain either a file or the contents of a directory. Hard links in such a graph create additional edges (which should not introduce any cycles), whereas soft links only contain a reference to a different path *(that is, they only say that another path needs to be evaluated, and do not modify the graph itself)*. Soft links are a form of **symbolic linking**. An alternative to symbolic linking is the use of **mounting**. Mounting has the effect of changing the resolution procedure to resolve paths with a certain prefix in a different manner; that is, a different closure is used from this prefix onward. The main difference between symbolic linking and mounting is that the former stores the actual location/name in a file (which then implicitly tells the resolver to resolve that path), whereas the latter actually modifies the resolver itself.

Naming services provide operations for adding, removing and looking up names. They also need a closure mechanism. *(It should be noted that a closure mechanism must be included wherever two namespaces are joined.)* For performance, scalability, and dependability, however, it is useful to distribute the data (directory nodes and subgraphs), as well as to distribute the process of name resolution.
The DNS name space can be divided into a global layer (top level domains), an administrational layer (consisting of zones, which are delegated portions of a domain), and a managerial layer.

The **global layer** has worldwide scope, and has only few nodes in the graph. This layer has many replicas. Since updates to this layer happen infrequently, responses can be cached and update propagation can be done lazily. Responses in this layer may take seconds.
The **administrational layer** has a single organization as its scope, and can contain many nodes in the graph. Updates are propagated almost immediately, client-side caching is typically used, and there are typically at most a few replicas. Responses in this layer may take milliseconds.
The **managerial layer**, *which is not a part of DNS*, has a single department as its scope, and has vast numbers of nodes in the graph. Client-side caching is sometimes applied to this layer, and updates are propagated immediately. Replicas are not used in this layer. Responses in this layer are typically immediate.
DNS is organized as a worldwide collection of name servers, which are collectively responsible for maintaining the tree-like database of DNS, and for responding to queries aimed at finding a host IP address. Some other services (e.g. email) can also have name lookups performed via DNS. DNS allows for modification by different administrations, and shows extreme robustness, in addition to scalability. Scalability is achieved by partitioning the name space, replicating servers, and performing caching.
Name servers in DNS servers a certain zone, for which they are **authoritative**. An authoritative server always maintains a record for the hosts for which it is authoritative. The root servers are authoritative for the root and the top-level domains, and name servers are configured with their addresses. Name servers for a domain know about their child domains and are typically replicated. DNS is often made available to applications as a middleware service.
There are two methods of resolving a given query:

1. **Iterative resolution**, where the client first makes a request to the root name server (to obtain the name server of the top level domain), then makes a request to the name server of the top level domain (to obtain the name server of the second level domain), then makes a request to the name server of the second level domain, and so on, until the name server for the desired hostname is reached. A major disadvantage of this method is its large communication overhead.
2. **Recursive resolution**, where the client makes a request to the root name server, which then queries the top-level domain name server, which then in turn queries the second level domain name server, and so on; the answer of the final name server (i.e. for the desired hostname) is returned along the chain of name servers, until it arrives back at the root name server, which returns the response to the client. A disadvantage of this method (as described here) is that it requires the name servers close to the root to make many additional requests (due to the indirection involved).

Typically, caching is used to optimize the name resolution process. In recursive resolution, servers in the chain of name servers can cache responses and return them whenever they are relevant. In iterative resolution, the iterative server caches all information.
In practice, a client asks a local name server for names in a recursive way, after which this local name server operates iteratively to improve caching.
We note that, even though the number of communications/connections is the same in both iterative and recursive name resolution, the sum of the round trip times is much larger for iterative name resolution when the name servers are much closer to each other than to the client.
Each domain has a set of associated resource **records** (e.g. an A record for an IP address). A name server can give an authoritative answer about resource records it manages; for the zone it manages, it has a start of authority record. Records contain a domain name, time to live (TTL), class *(which is typically IN, for internet)*, type, and value.
To improve the scaling of DNS, several strategies are applied. First, servers are replicated, e.g. using anycast and geoDNS (i.e. responding with the IP of a server close to the request source). As a closure mechanism, DNS typically starts resolution at a local server which caches responses. Caching can be quite efficient because addresses are assumed not to change very frequently.

6. Virtualization

Virtualization partitions a set of physical resources into multiple isolated logical resources. For example, a physical memory can be used to construct a virtual memory, a physical core can be used to simulate virtual cores, a physical machine can run multiple virtual machines, and a local area network (LAN) can be partitioned into multiple VLANs. There are several motivations for virtualization:

1. **Resource sharing**: virtualization allows resources to be shared in such a way that one only needs to pay for usage;
2. **Isolation**: virtual machines provide better isolation than processes (mostly from a security point of view), although performance isolation remains difficult;
3. **Portability**: virtual machines can run on any of the provider's physical machines; the allocation of virtual machines to physical machines, as well as the connectivity of the virtual machines can be managed by the provider.

Computer system interfaces connect 4 different layers:
- the application layer;
- the library layer;
- the operating system layer;
- the hardware layer.

There are four different methods of interactions:
I. Library functions, which connect the application to the library;
II. System calls, which connect the library to the operating system;
III. Privileged instructions, which connect the operating system to the hardware;
IV. General instructions, which connect all layers to the hardware.

The combination of I and IV is known as the **application programming interface (API)**. The combination of II and IV is known as the **application binary interface (ABI)**. The combination of III and IV is known as the **instruction set architecture (ISA)**.

The system virtual machine provides the ISA interface, and is realized by a **hypervisor** or a **virtual machine monitor (VMM)** on top of a physical machine and on top of an OS. A **virtual machine** is any environment created by the virtual machine monitor. Each virtual machine runs its own guest OS.

There are three types of system virtual machine:
1. A **process VM** provides a separate set of instruction, and functions as an interpreter/emulator running on top of an OS.
2. A **native VMM** provides low-level instructions, along with a bare-bones minimal OS;
3. A **hosted VMM** provides low-level instructions, but delegates most work to a full-fledged OS.

There are several types of sensitive instructions:
- **Control-sensitive instructions** may affect the configuration of a machine, e.g. memory resources;
- **Behavior-sensitive instructions** have an effect which is partially determined by context. For example, an instruction may set a flag only when it is executed in system mode.

Sensitive instructions need to be executed in kernel mode. The **Popek-Goldberg theorem** states that, *'for any conventional computer, a VMM may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions'*. **Paravirtualization** is a technique which modifies a guest OS such that all sensitive instructions are replaced by hypervisor calls.

When provisioning, there are several bounds/restrictions to take into account. A lower bound is given by the idea that we need `total reservation of admitted VMs + new VMs reservation <= host capacity`. An upper bound can be obtained by considering that we need to have `request + current allocation <= limit`. Finally, we have that

$$allocation\left(VM_i\right) = \frac{share\left(VM_i\right)}{\sum_j share\left(VM_j\right)}.$$

VMs can be migrated to different physical machines in two ways:
1. **Cold migration** requires the virtual machine to be shut down first. Then, it can be migrated to another physical machine and restarted on that machine. The main disadvantage of cold migration is that it can require a large amount of downtime.
2. **Live migration** migrates a machine while it keeps running, which significantly decreases the amount of downtime. This makes it the preferred choice in cloud computing. However, live

migration is significantly more complicated, as it requires transferring the CPU state, network connections, memory content and storage content while avoiding other problems (e.g. consistency).

Oversubscription occurs when the total contractually agreed provision of resource R to all VMs on a physical machine P exceeds its capacity. Oversubscription can be agreed upon in SLAs and can make the use of physical machines more efficient. The degree of oversubscription on a given physical machine may vary by resource (CPU, memory, storage, and network).
Overload, on the other hand, occurs when the total demand for resource R by all VMs on a physical machine P at time t exceeds its capacity.

Containers are a form of OS virtualization where each container shares kernel functionality, but has its own libraries. Containers typically run a single (micro)service or application, and are executed in a runtime environment, known as the **container engine**. Containers are typically lightweight compared to VMs, and run on top of a host OS, where they are visible as processes.

Docker Swarm provides an engine for orchestrating microservice applications in a secure cluster of Docker hosts. A cluster consists of master and worker nodes, where the workers execute containerized applications and the masters manage the clusters and are replicated to ensure availability. The cluster state is maintained in a distributed memory database known as `etcd`. Security is provided in Docker Swarm by default, by authenticating nodes and encryption management data and network communication.

7. Scalability and elasticity

Scalability roughly describes how to address the 'growth' of a service. There is no consensus on the definition of scalability in the literature, and it is difficult to capture in scenarios and tactics. Two qualitative definitions of scalability are as follows:
1. 'Scalability is the ability to handle increased workload (without adding resources to a system)';
2. 'Scalability is the ability to handle increased workload by repeatedly applying a cost-effective strategy for extending a system's capacity'.

Another piece of literature distinguishes between the following 4 types of scalability:
1. **Load scalability** describes how a system deals with increased load;
2. **Space scalability** describes how a system deals with increased demand for memory;
3. **Space-time scalability** describes how a system deals with increased demand for number of objects;
4. **Structural scalability** deals with expanding the system without major modifications to its architecture.

Scaling the size of a system has no effect on the sequential part, and a linear effect on the parallel part. **Amdahl's law** computes the speedup of a problem under the assumption that the problem size is kept fixed. This gives the following equation:

$$speedup = \frac{1}{(1-P)+\frac{P}{N}},$$

where P is the parallelizable portion of the program and N is the number of processors.
Gustafson's law considers a slightly different kind of problem: it considers problems where the *absolute* amount of parallelism increases with the problem size. Gustafson's law effectively describes the speedup as the extra factor of time needed when a parallelized program is ran sequentially. This gives the following equation:

$$speedup = S + P \cdot N,$$

where S is the serial portion of the program, P is the parallelizable portion of the program and N is the number of processors[7].
Amdahl's law suggests that, when problems of a fixed size are being solved and *'if tuning a variable has a negligible impact on the performance,'* it might be better to *'move your attention somewhere else (to another system variable)'*.
From Gustafson's law, it follows that, when the problem size increases, one should *'utilize more resources to solve large problems, not fixed size ones'*.

There are several tactics and/or rules of thumb to achieve scalability:

- dependencies should be limited;
- no machine should need/have complete information;
- decisions should be based on local information;
- failure of one machine should not ruin the results;
- there should not be an assumption of a global, shared clock.

Techniques and mechanisms to achieve scalability include:

- hiding latency *(by doing something useful while waiting)*;
- introducing concurrency;
- partitioning, by replicating/relocating data, or by dividing work into smaller tasks;
- limiting communication.

Elasticity is defined as the *'ability of a system to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner'*. Elasticity requires one or more adaptation mechanisms, such as auto-scalers. Scalability is a pre-requisite for elasticity, but not sufficient; scalability does not consider temporal aspects such as how fast, how often, and with what granularity a system should scale.
Several aspects are relevant when trying to achieve elasticity:

- *Timing*: this includes aspects such as the percentage of time during which a system is in an under-provisioned, over-provisioned, or perfect state, as well as the frequency of adaptations. These aspects depend on the adaptation policy.
- *Latency*, which describes the time needed to bring up or drop a resource. This depends on the adaptation mechanism.
- *Accuracy*, which describes the relative deviation of the amount of allocated resources from the actual resource demand *(i.e. compares supply to demand)*, expressed as an average over time. The accuracy depends on the allocation granularity and the adaptation policy.

Elasticity can be measured by expressing the underprovisioning *(and overprovisioning)* of resources in the following quantities:

$$Accuracy_U = \frac{\sum_i U_i}{T},$$
$$Timeshare_U = \frac{\sum_i A_i}{T},$$

where U_i is the area between the resource supply and demand curves when the supply is too little *(effectively, this is the amount of resources we lack, multiplied by the amount of time during which we lack them)*, A_i is the amount of time during which we lack resources, and T is the total amount of time. A similar definition holds for overprovisioning, where we consider the surplus of resources we have *(i.e. where supply is too large)*, and the time during which we have too many resources.
Another way to measure elasticity is through the so-called **jitter**, which describes the amount of changes in the number of resources. The jitter J is given by

$$J = \frac{E_S - E_D}{T},$$

[7] Remark: the example in the lecture defines S as the execution time of the serial portion of the program, and P as the portion of the program which can be parallelized multiplied by the execution time of the parallel part.

where E_S is the number of supply changes and E_D is the number of demand changes during interval T.

8. Quality attributes: scenarios & tactics

Quality attributes of a system can describe how well the system adheres to certain extra-functional requirements. We define the following quality attributes:

- **availability** expresses the readiness for correct service as a probability;
- **reliability** expresses the continuity of correct service as a length of time;
- **modifiability** expresses how difficult it is to introduce desired changes;
- **performance** describes whether the service timely responds to service request events, and also is related to throughput and jitter;
- **testability** describes how difficult it is to verify the correctness of the system;
- **usability** describes how user-friendly the system is;
- **portability** describes how difficult it is to make the system run on another platform.

Note: it was suggested in the lecture that these definitions do not need to be remembered.

Dependability is defined as the ability to deliver service that can justifiably be trusted. It consists of attributes such as availability, reliability, safety, integrity, and maintainability. **Security** is the ability to resist unauthorized attempts to access data and services. It encompasses attributes such as availability, integrity, and confidentiality.

Quality attributes can refer to a system, its architecture, the development process or business qualities. We focus on the qualities of the developed system.

A **scenario** defines the desired response of a system in a particular situation. A scenario consists of 6 elements, which are divided into 3 categories:

I. Inputs:
   1. **stimulus**;
   2. **source of stimulus**;

II. Preconditions:
   3. **environment** of the scenario *(i.e. state of the system)*;
   4. **resources** (artifacts) that are *subject of the QA*;

III. Required outputs:
   5. **response**;
   6. **response measure**, *where metrics are used to specify the desired level/intensity of the response. Ideally, these metrics are as quantitative as possible*.

Tactics describe *how to enforce required quality*, and become a part of the designed system and its architecture. Every step in the development process further constrains the space to take quality attributes into account. As the architecture is at the top level of the process, it is most critical; however, the architecture by itself cannot provide quality attributes.

Several quality attributes conflict with each other. **Brewer's CAP theorem** states that any networked shared-data system can have *at most two* of the following three desirable properties:

1. **Consistency**, i.e. having a single up-to-date copy of the data;
2. High **availability** of the data (for updates);
3. **Tolerance** to network **partitions**.

Assumptions about the stimuli in a scenario can be captured by models. For example, one can use failure models, attack models, and load/usage models.

Reliability can be expressed as a length of time, measured by $MTTF$. **Availability** is expressed as a probability, measured by $\frac{MTTF}{MTTF + MTTR}$, where $MTTF$ is the mean time to failure, and $MTTR$ is the mean time to repair.

Tactics are defined in a very similar way to scenarios: they use the same 6 elements, but describe the categories differently:

I. The inputs are now described as 'inputs that originate from the stimulus';
II. The preconditions are now described as 'inputs that originate from the system';
III. The required outputs should now be 'guaranteed outputs'.

One can thus say that tactics guarantee required outputs *for given inputs*. The tactic effectively comes in between elements (I) and (II) on the one hand and (III) on the other hand, and describes what to do to achieve the guaranteed output.

Modifiability is a quality attribute concerned with the *cost of system change*. Cost factors describe the extent of (a single) responsibility; more complex responsibilities imply larger modules which have a larger modification cost. **Coupling** is the degree to which the responsibilities of two components are related. Stronger coupling implies a *higher cost* to modify components. A single change may affect multiple modules if they depend on each other or have overlapping responsibilities. **Cohesion** is the degree to which the responsibilities of a single module form a meaningful unit. Stronger cohesion implies a *lower cost* to modify components. Several types of cohesion can be considered good: functional, sequential, and data cohesion. Other types of cohesion, such as procedural, temporal, logical and coincidental cohesion are typically bad, due to them forming arbitrary grouping criteria.
Several tactics can be used to achieve modifiability. Some of them are based on the idea of *'localizing modification'*:

- Maintain semantic coherence;
- Anticipate and isolate expected changes;
- Abstract common services;
- Generalize the module and/or its data representations;
- Limit options.

Other tactics are based on *'preventing ripple effects'*. These include:

- Hide information/encapsulate data;
- Maintain existing interfaces;
- Insert an intermediary between modules.

Yet another set of tactics is based on *'deferring binding time'*. These tactics include:

- Runtime registration;
- Using configuration files;
- Polymorphism (i.e. runtime binding of method calls);
- Component replacement;
- Runtime binding of independent services.

Modifiability is addressed both in the development view *(with respect to the distribution of responsibilities over modules, the interfaces that define access to data and services, the application of layering to reduce 'use'-dependencies, and by making coherence and coupling explicit)* and in the process view *(with respect to binding and configuration)*. Styles and frameworks typically address modifiability as well by defining a particular structure of building blocks (each of which has their own responsibilities) and interaction patterns.

9. Quality attributes: fault tolerance

Fault tolerance is a quality attribute of a system which describes the system's ability to deal with faults. In practice, this means a fault-tolerant system provides a graceful degradation of service.

We distinguish between failures, errors and faults:

- A **failure** occurs when *a system does not meet its specification*. This means that the externally observable behavior deviates from the expected correct behavior.
- An **error** is *a system state that may lead to a failure*. Errors need not be externally observable, and the system may still recover before it displays a failure.
- A **fault** is the cause of an error. It should be noted that the failure of another system can be a fault.

Fault tolerance is concerned with preventing faults from becoming system failures.

A fault description can consist of information on the origin of a fault, the reasons for its occurrence, and the duration of the fault. A **fault**/failure **model** identifies the system components where faults may occur, classifies the type of faults that can occur, and defines the notion of 'correct behavior'. Resilience of the system can be expressed using fault metrics, which measure the level of fault tolerance. One important metric is the concept of t-resilience of a system: a system is said to be **t-resilient** if it can tolerate up to and including t faulty components.

We distinguish 5 types of failures:

- In a **crash failure**, a server halts, but is working correctly until it halts. Furthermore, the halting is observable by other processes/clients.
- In an **omission** failure, a server fails to respond to incoming requests. This type can be further subdivided into receive omissions, where the server fails to receive an incoming message, and send omissions, where a server fails to send messages.
- In a **timing failure**, a server's response lies outside a specified time interval. In other words, the server responds either too fast or too slow.
- In a **response failure**, the server's response is incorrect. This type can be further subdivided into value failures, in which the value of the response is wrong, and state transition failures, in which the server deviates from the correct flow of control.
- In an **arbitrary failure**, also known as a **Byzantine failure**, a server may produce arbitrary responses at arbitrary times.

In the context where a process P attempts to detect the failure of a process Q, we distinguish 5 types of halting:

- **Fail-stop halting** occurs in case of a crash failure which is reliably detectable.
- **Fail-noisy halting** occurs in case of a crash failure which is *eventually* reliably detectable.
- **Fail-silent halting** can occur in cases of omission or crash failures. Here, the client cannot tell what went wrong.
- **Fail-safe halting** can occur in case of arbitrary failures, where the failures are benign in the sense that they cannot do any harm.
- **Fail-arbitrary halting** occurs when arbitrary failures occur in malicious cases.

In the context of messaging, several types of failures can occur:

1. Messages can be *lost*;
2. Messages can be *duplicated*;
3. Messages can be *created*.

In the context of message loss, the following requirements can be set:

- **Fair-loss delivery (FLD)** states that, 'if a correct process p infinitely often sends a message m to a correct process q, then q delivers m an infinite number of times'.
- **Reliable delivery (RD)** states that, 'if a correct process p sends a message m to a correct process q, then q eventually delivers m'.

In the context of message duplication, the following requirements can be set:

- **Finite duplication (FD)** states that, 'if a correct process p sends a message m a finite number of times to a correct process q, then m cannot be delivered an infinite number of times by q'. In other words, this means that the network does not repeatedly perform more retransmissions.
- **Stubborn delivery (SD)** states that, 'if a correct process p sends a message m once to a correct process q, then q delivers m an infinite number of times'.
- **No duplication (ND)** states that, 'no message m is delivered by a process more than once'.

In the context of message creation, the following requirements can be set:

- **No creation (NC)** states that, 'if some process q delivers a message m with sender p, then m was previously sent to q by p'.
- **Authenticity (AU)** states that, 'if some correct process q delivers a message m with sender p and process p is correct, then m was previously sent to q by p'.

The requirements described above can be combined to form link models. The following four link models have been mentioned in the lecture:

- **Fair-loss links** provide fair-loss delivery, under finite duplication and with no creation (FLD, FD, NC). Fair-loss links assume a crash-stop process model.
- **Stubborn links** do not state a loss requirement, and have stubborn delivery with no creation (SD, NC). Stubborn links also assume a crash-stop process model.
- **Perfect links** provide reliable delivery, under no duplication and with no creation (RD, ND, NC). Perfect links assume a crash-stop model as well.
- **Authenticated perfect links** are similar to perfect links, with the exception that they replace the no-creation requirement by a requirement for authenticity. This gives the combination of requirements RD, ND, AU. Authenticated perfect links work reasonably well in a Byzantine process model.

It should be noted that UDP guarantees fair-loss delivery (FLD) and that TCP guarantees reliable delivery (RD).

A one-to-one communication (unicast) service is **reliable** if it satisfies the requirements for a perfect link (RD, ND, NC). In this context, the reliable delivery (RD) requirement is referred to as **validity**. The combination of the no-duplication (ND) and no-creation (NC) requirements is referred to as **integrity**.

In a **fault-tolerant process group**, each correct process executes the same commands in the same order as every other correct process. This can be achieved using reliable multicast communication. A group communication (multicast) is **reliable** when it satisfies the following three requirements:

1. **Validity**, which states that 'if a correct process multicasts message m, then it will eventually deliver m (to itself)';
2. **Integrity**, which states that 'a correct process delivers message m at most once and if it delivers a message with sender s, then it belongs to the group of m, and m was previously sent in a multicast operation by s';
3. **Agreement**, which states that 'if some correct process delivers m, then all other correct processes in the group will eventually deliver m'.

A **consensus problem** requires a set of correct processes to make the same decision (i.e. to reach agreement). Typical applications of consensus problems include leader selection, deciding whether to commit/abort a transaction and, *as we consider here, establishing a common total order of operations*. A valid solution to a consensus problem should provide us with the following **consensus guarantees**:

- **termination**: every correct process eventually decides *some* value;
- **validity**: if a process decides v, then v was proposed by some process;
- **integrity**: no process decides twice;
- and one of the following two guarantees:
  - **agreement**: no two <u>*correct*</u> processes decide differently;
  - **uniform agreement**: no two processes decide differently.

Several tactics can be used to improve fault tolerance:

- **Fault masking**, which consists of error detection, error diagnosis, error/fault containment, and finally error recovery. In other words, fault masking attempts to prevent a fault from turning into a failure.
  - Error detection can be performed in several different ways, including performing acceptance tests, raising exceptions, setting a watchdog timer (which gives an error when the result is not available within the specified time) and by using pings/echos. An alternative way uses heartbeat messages to detect errors.
  - Error recovery can be performed either in a forward manner or in a backward manner. Forward error recovery finds a new error-free state and continues the computation from there (such as state can e.g. be the initial state). Backward error recovery restores a previous error-free state, which requires storing checkpoints.

- **Redundancy**, which can be achieved in several different ways:
  - By having **spares**. There are three types of spares: hot spares, where updates are synchronous with the primary; warm spares, which are updated periodically and thus run behind within certain bounds; and cold spares, which are off-line, and do not get updated without being brought online first.
  - Redundancy can also be used to mask failures, e.g. by adding error correction codes (which is a form of **information redundancy**), by repeating failing operations (under the assumption that the fault was temporary, this is a form of **time redundancy**), or by having multiple hardware/software components (e.g. as in a RAID configuration, which gives a form of **physical redundancy**).
  - There are two different types of redundancy:
    - **Active redundancy**, in which all replicas perform calculations/updates in parallel and determine which outcome to deliver. This requires synchronization to achieve consistency.
    - **Passive redundancy**, where a replica is activated only when a previous replica fails.
  - In **triple modular redundancy**, signals are replicated three times and then pass through a set of so-called voters; even if one of the incoming signals fails, the voters can still determine from the remaining two incoming signal replicas what the correct value was.
- **Fault prevention** centers around using sound development methodologies, properly working with transactions (i.e. in an ACID[8] manner), and perhaps temporarily removing nodes from service (e.g. restarting a component to clean up accumulating memory leaks).

Generally, tactics for improving availability and reliability center around extending the uptime of the system or restricting the downtime of the system. Together, these techniques are referred to as **fault tolerance techniques**, which aim to prevent faults from becoming failures.

10. Consistency and replication

Partitioning, also known as sharding, *splits the data* into smaller subsets (called partitions or shards), and distributes them over multiple nodes. **Replication**, on the other hand, distributes *multiple copies* (replicas) of the same data set over distinct nodes. Replication can be used to achieve redundancy. In practice, partitioning and replication are often combined, where multiple replicas exist for each partition.

There are multiple **quality drivers for partitioning**:

1. Partitioning can help when the size of the dataset (e.g. a database) becomes *too large* for a single node;
2. Partitioning can improve *scalability*, as it allows for load balancing of data. This works best for key-value data stores (e.g. NOSQL);
3. Partitioning can improve *performance*, as throughput can be improved when distinct partitions can be accessed concurrently.

Key-value data can be partitioned in two different ways:

- By *key range*. This works for totally ordered keys, and is efficient for range queries. A downside is that hits may lead to so-called '**hot ranges**'; that is, if a given range of keys is queried disproportionately often, then the node responsible for this range will easily get overloaded (while other nodes will not be doing nearly as much work);
- By *key hash*. This works by hashing keys to achieve a uniform distribution over nodes. Using a hash function to distribute keys is inefficient when range queries are used (since elements in the range are distributed uniformly over all nodes). Furthermore, so-called '**hot keys**', i.e. keys with a high volume of read or write requests, remain a problem.

Sometimes, **rebalancing** partitions is necessary. To give some examples, rebalancing may be needed when the workload per node has increased, the partition size should be increased, or when nodes fail and the node set needs to be adjusted in size. It is noted that re-scaling the node set (or the set of possible keys) leads to a different mapping between keys and nodes, and thus requires re-distributing records over partitions. It can be useful to have multiple partitions per

[8] ACID stands for atomic, consistent, isolated and durable. Consistent refers to all system invariants being maintained at all times. Isolated refers to no transient states being observable.

node; this introduces two mappings (one from keys to partitions and one from partitions to nodes), and may be useful to allow load balancing by moving[9] partitions over nodes in such a way that the load on each node is balanced.

There are multiple **quality drivers for replication**:
1. *Reliability*: replication removes a single point of failure and allows the use of consensus protocols to deal with corrupted data;
2. *Availability*: the probability of n servers being unavailable is only $1 - p^n$ if each server has an (independent) probability of p of being unavailable. In other words, the probability of all replicas being unavailable is much smaller than that of one server/replica being unavailable.
3. *Performance*: throughput can be improved by allowing concurrent access of distinct replicas. Furthermore, latency can be reduced by accessing a nearby replica, and the network traffic can be reduced due to increased data proximity.
4. *Scalability*: replication allows load balancing.

A system provides **replication transparency** when users of the system are unaware of the fact that several replicas exists. The concept mainly refers to data values or services being replicated. Note that, when replication transparency applies for a client, the client can only identify a single (logical) data object, and the client also has location transparency.

In an architecture, several concerns apply with respect to replication. In particular, since replication comes with costs, it is useful to consider how many replicas to use, where to place them, whether and how to maintain consistency, what architectural elements to use for storage and management, and what protocols to use for accessing replicas.

A basic model for replication has multiple clients and one or more servers. The server is a distributed data store with two operations: update (i.e. write) and query (i.e. read). In this model, each server has a special entity, known as the **replica manager (RM)**, which manages the local part of the data store. Clients send requests to front ends, which then contact the service (consisting of the replica managers) to retrieve data. The behavior of the system can be described using 5 phases, which *do **not** necessarily have to be executed in the given order*:
1. **Request phase**, in which requests are accepted by the frontend and then either communicated to a fixed RM *(in a passive replication scheme)* or multicasted to all RMs *(in an active replication scheme)*.
2. **Coordination phase**, in which the RMs determine the execution order of the requests.
3. **Execution phase**, in which the RMs *tentatively* execute the requested operation.
4. **Agreement phase**, in which the RMs reach consensus on the effect of the operation and commit the operation.
5. **Response phase**, in which some RMs respond to the frontend, which in turn replies to the client.

When multiple replicas are used, and one copy of the data is updated, that change should be made at all copies. Keeping the content of all replicas the same is known as **consistency**. **Consistency models** are contracts between a data store and clients, and specify the unit[10] of consistency. Consistency models also determine the outcome of a sequence of read/write operations performed by one or more clients, and can be classified into two broad categories:
1. **Data-centric models**, which aim to achieve consistency from the perspective of the data store;
2. **Client-centric models**, which aim to achieve consistency from the perspective of the client.

The key idea behind consistency is the **single-server paradigm**, which states that operations should *'appear as if they were performed as indivisible actions by a single server with requires'*. This means that queries/updates should have the same effect, and be executed in the same order

[9] Note that nodes can have different numbers of partitions in this scheme. For example, a frequently-used partition may be put on a node all by itself, while many rarely-used partitions are put together on a single node. This makes the total load on each node closer to the average.

[10] In this context, 'unit' refers to the set of items which are updated together.

across all replicas. Two operations are said to be **conflicting** *if the outcome of executing them as a sequence of two atomic actions may potentially differ for the two possible execution orderings*. Conflicting operations come in two flavors:

- **read-write conflicts** occur when different values are returned for two execution orderings;
- **write-write conflicts** occur when the store is left in different states for two execution orderings.

Consistency places constraints on the interleaving of operations allowed to the single server. These constraints are described according to the following rules:

1. The interleaved sequence of operations meets the specification of a (single) correct copy of the objects;
2. The order of operations in the interleaving originating from a single client is consistent with the order in which that client issued them (program order);
3. The order of operations in the interleaving is consistent with the global (real-time) ordering of the operations.

These constraints can also be summarized informally as follows:

1. Only one party can hold a lock on an object;
2. The order of operations follows the original order of the program;
3. The order of operations follows the global (real-time) ordering.

Two operations which (at least partially) overlap in time are said to be **concurrent**. Two operations which are (fully) separated in time are said to be **serial**.
Actual executions of operations are **sequentially consistent** if they satisfy the first two rules. A data store with replicated objects is **sequentially consistent** when *all* its executions are sequentially consistent. It can be noted that sequential consistency effectively *allows swapping the order of pairs of consecutive operations originating from distinct clients to obtain a single server execution (even when they are conflicting)*. A different definition of sequential consistency (from the book) is as follows:

> *"A data store is said to be sequentially consistent when it satisfies the following condition: the result of any execution is the same as if the (read and write) operations by all processes on the data store were executed in some sequential order, and the operations of each individual process appear in this sequence in the order specified by its program."*

It should be noted that sequential consistency is *not* compositional; that is, when having data items that are each kept sequentially consistent, their composition as a set need not be so. This can be solved by making the assumption of linearizability. According to the book, **linearizability** states that *"each operation should appear to take effect instantaneously at some moment between its start and completion"*. A different definition is based on the rules above: actual executions that satisfy the first and third rule are called **linearizable**. Similarly, a data store with replicated objects is linearizable when all its executions are linearizable. Linearizability effectively places the constraint that *swapping operations originating from distinct clients is only allowed if they are concurrent operations or concern independent variables*. In principle, linearizability means that, at the time of completion of a write operation, the results should be propagated to the other data stores/replicas.

Causal consistency is a different and weaker notion of data-centric consistency. Causal consistency requires that *"writes that are potentially causally related must be seen by all processes in the same order. Concurrent[11] writes may be seen in a different order on different machines."* Sequential consistency is stronger than causal consistency in the sense that it requires *all* writes to be seen in the same order by all clients. Causal consistency can be enforced by the delivery mechanism in the coordination phase, and can be determined by using timestamps and vector clocks.

Eventual consistency is an even weaker data-centric consistency notion. Eventual consistency requires that *"in the absence of further updates and system failures all replicas eventually have the same state."* Eventual consistency can be considered as a bare minimum notion of consistency.

There are 4 **client-centric consistency models**:

[11] In this context, the book seems to define 'concurrent' as 'not being causally related'.

1. **Monotonic read consistency** states that *"if a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more recent one"*. It can be achieved by storing updates to the item along with it (so that clients can determine whether they have the most recent version or not).
2. **Monotonic write consistency** states that *"a write operation on a data item x is completed before any successive write operation on x by the same process"*. It can be achieved by attaching version numbers to objects, or by making writes atomic (i.e. processing them on all servers before returning).
3. **Read your writes consistency** states that *"the effect of a write operation by a process on data item x will always be seen by a successive read operation on x by the same process"*. It can be achieved by making writes atomic (i.e. processing them on all servers before returning), or by making the replica notify all other replicas that the old value has become stale before returning.
4. **Write follow reads consistency** states that *"a write operation by a process on a data item x following a previous read operation on x by the same process is guaranteed to take place on the same or a more recent value of x than was read"*.

Replica placement involves several aspects, such as where to place the servers (i.e. ***replica managers***), where to place the replicated content (also known as *the* ***replicas***), and whether to assign the replicas to replica managers statically or dynamically. There are 3 types of replica placement:
1. **Permanent replicas**, of which there generally are a limited, fixed, and statically determined number. The use of permanent replicas presumes a stable and a-priori known load distribution, and can lead to increased reliability and performance.
2. **Server-initiated replicas**, of which there are a dynamically changing number. They are typically used to cope with peak loads on the service, and can increase availability.
3. **Client-initiated replicas** *(i.e. client caches)*, which can improve response time by improving data proximity and also reduce the network load. Client-initiated replicas have no persistent storage; that is, the data has limited lifetime. Furthermore, the client is responsible for maintaining consistency, although the server may assist in this.

Replication protocols determine what happens upon an update operation. Replication protocols can be classified according to the following aspects:
- The *information being disseminated*: this could be the operation itself (resulting in active replication), the resulting state (resulting in passive replication), or a notification.
- The *party which takes the initiative* for the dissemination: a distinction can be made between push-based (where the server initiates the dissemination) and pull-based (where the client initiates the dissemination) protocols.
- The *communication primitives* which are used for the dissemination: this could be unicast, multicast, or gossiping.
- The *moment at which completion is reported* to the client: this can be upon receipt of the operation, after atomically executing the operation, or using a quorum-based strategy.

Together, these aspects determine the supported consistency model.

There are several distinctions between **push-based** and **pull-based protocols**. These distinctions are summarized in the following table:

Issue	Push-based protocol	Pull-based protocol
State of server	List of client replicas and caches	None
Messages sent	Update (and possibly fetch update later)	Poll and update
Response time at client	Immediate (or fetch-update time)	Fetch-update time

A mix of push- and pull-based protocols can be obtained using a mechanism based on **leases**. In such a mechanism, updates are pushed only when the client has an outstanding lease (on an object). Clients without a lease either pull data or request a lease. Client state at the server-side can, in this manner, be limited to maintaining a list of outstanding leases, which is typically more efficient than maintaining a list of all clients. The server can influence the size of its state by modifying the lease time.

Protocols can be classified as:
- **Passive replication protocols**, which propagate values;
- **Active replication protocols**, which propagate operations;
- **Gossip-based protocols**, which propagate updates randomly.

Primary-based protocols are protocols in which each data item x is associated with a primary, which is responsible for coordinating write operations on x. A distinction can be made based on whether the primary *is always on a fixed remote server*, or whether the primary *can be transferred to the local process* which performs a write operation. This gives rise to the following distinguishable types of primary-based protocols:
- In **primary-read, primary-write protocols**, all reads and writes go through the primary. This type of protocol supports linearizability, as clients perceive atomic reads and writes. However, all reads and writes are blocking, which means that there is no concurrency and limited availability. As a result, this type of protocol is seldomly used.
- In **local-read, primary-write**[12] (or **remote-write**) **protocols**[13], sequential consistency is supported by having blocking writes, and non-blocking concurrent reads. This type of protocol bounds the staleness of values, as at most one write will be missed by a reading client. This type of protocol effectively blocks the writing client until updates have propagated to all replicas.
- A **local-read, local-write** (also known as **local-write**) **protocol** supports eventual consistency and has no blocking operations (and therefore has 100% availability[14]). This type of protocol works by migrating the primary to the local process, and then performing the write locally (on what is now the primary).

In passive replication, the five phases of the replica manager behavior are executed in the following order (which is equivalent to the order given in the definition earlier in this document):
1. During the *request phase*, the frontend attaches a UID to the request and forwards it to the primary;
2. During the *coordination phase*, the primary handles the request atomically and in order of receipt, using the UID to prevent executing an update more than once;
3. During the *execution phase*, the primary executes the request and stores the response;
4. During the *agreement phase*, in case of the request is for an update, the primary sends the updated state, the UID, and the response to the backups and waits for acknowledgments;
5. During the *response phase*, the primary responds to the client's frontend, which hands the response to the client.

Several recovery procedures apply in this context:
- Upon failure of the primary replica manager (RM), linearizability can be retained by replacing the primary with a unique backup, which can be determined by a leader election;
- When the backup RM takes over, all backups agree on the set of operations that have been performed by sending the write request to the backups using atomic multicast.

In active replication, the frontend multicasts requests to all replica managers (RMs), which all behave the same/symmetrically. This achieves sequential consistency, due to a total ordering being used to deliver updates. The five phases of replica manager behavior now take the following form:

[12] Note that the slides erroneously refer to this as a 'write' protocol; the first three words have accidentally been moved several lines up in the slides.

[13] The book explains one type of write/remote-write protocols: a **primary-backup protocol**.

[14] The availability is only 100% from the perspective of the protocol, not from the perspective of failures.

1. During the *request phase*, the frontend attaches a UID to client operations and forwards it to all RMs using totally ordered reliable multicast;
2. During the *coordination phase*, group communication delivers the request to every correct RM in the same total order;
3. During the *execution phase*, every RM executes the operation on its replica, resulting in the same local state and response;
4. *There is no agreement phase in active replication;*
5. During the *response phase*, the RMs send their responses labeled with the request identifier to the frontend, which forwards it to the client after a certain number of responses have been received; the exact number of responses depends on the desired level of fault resilience.

In **quorum-based protocols**, there exists a general scheme. In this scheme, there are N replicas of a file. To read this file, a client needs to assemble a **read quorum**, which is an arbitrary collection of N_R nodes. To modify a file, a client needs to assemble a write quorum, which is an arbitrary collection of N_W nodes. Then, to prevent conflicts, the values of N_R and N_W must satisfy the following constraints:

1. $N_R + N_W > N$, which guarantees the absence of **read-write conflicts**, as any read operation will involve at least one of the N_W nodes on which the last write was performed.
2. $N_W > \frac{N}{2}$, which guarantees the absence of **write-write conflicts**, as the latest version's number can be determined through a majority vote. More specifically, any write operation will involve at least one of the N_W nodes on which the last write operation was performed (and which therefore have correct knowledge of the number of the most recent version).

The scheme in which $N_R = 1$ and $N_W = N$ is known as **Read-One, Write-All (ROWA)**. In this scheme, any replica is able to respond to read operations without involving any other replicas *(which hence provides good read performance)*, but at the cost of requiring all nodes to be involved in write operations.