Uni Distributed Systems Summary

Summary for the distributed systems course at HdM Stuttgart

Felicitas Pojtinger

2023-02-05

Meta

Contributing

These study materials are heavily based on professor Kriha's "Verteilte Systeme" lecture at HdM Stuttgart and prior work of fellow students.

Found an error or have a suggestion? Please open an issue on GitHub (github.com/pojntfx/uni-distributedsystems-notes):

Figure 1: QR code to source repository

If you like the study materials, a GitHub star is always appreciated :)

License

Figure 2: AGPL-3.0 license badge

Course Timeline

This course on distributed systems covers a range of topics related to the design, implementation, and management of distributed systems. The course is divided into several sections, including:

1. **Introduction to distributed systems**: This section provides an overview of distributed systems and their key characteristics.
2. **Theoretical models of distributed systems**: This section covers the use of theoretical models, such as queuing theory and process and I/O models, to understand and analyze distributed systems.
3. **Message protocols**: This section covers the use of message protocols, including delivery guarantees, causality, and reliable broadcast, to facilitate communication between components in a distributed system.
4. **Remote procedure calls**: This section covers the use of remote procedure calls (RPCs) to invoke functions on a remote machine, as well as different RPC mechanisms such as marshaling, thrift, and

Introduction to Distributed Systems

Overview

1. Definition of a distributed system (DS)
2. Challenges for developers working with DS
3. Reasons for using a DS
4. Examples of DS
5. Characteristics of DS
6. Middleware for DS
7. Concepts and architectures in DS, including scale, parallelism, and latency
8. Resources related to DS

Definition of a Distributed System

A system that is made up of independent agents that interact with each other and produce a cohesive behavior. The interactions and events in this system happen concurrently and in parallel, meaning that they occur simultaneously and independently of one another. This type of system is often used to perform tasks that are too complex or large for a single agent to handle, and the concurrent and parallel nature of the interactions allows for efficient and effective processing of the tasks.

Emergence

Four types of emergence: strong emergence, weak emergence, evolutionary emergence, and constructed emergence.

- **Strong emergence** refers to situations where it is not possible to predict what will emerge from the interactions of the system.
- **Weak emergence** refers to situations where simple principles combine to produce surprising results.
- **Evolutionary emergence** refers to the complex but robust development of a system over time, such as the transformation of an egg into a human being.
- **Constructed emergence** refers to the complex but often not robust emergence of a system that has been intentionally designed, such as a distributed system.

Emergent failure modes: Instances of cascading failures in constructed emergence.

Why are Distributed Systems complicated?

- One reason is **emergence**, which refers to the complex and often unexpected behaviors that can arise when multiple components of a system interact with each other.
- Another reason is the **single machine view**, which can make it difficult to understand and debug issues that arise in a distributed system.
- **Errors** are also an inherent part of distributed systems, and developers must be prepared to handle and troubleshoot these errors.
- Additionally, there is no **“free lunch”** in distributed systems, meaning that there is always some trade-off or cost associated with every design decision.
- Finally, developing a distributed system involves **total end-to-end system engineering**, which can be a complex and time-consuming process.
- All distributed systems algorithms are also **based on the failures that**

Why Distribute a System?

There are several reasons why an organization might choose to use a distributed system:

- **Robustness/resilience**: Distributed systems are designed to be resistant to single points of failure, which makes them more resilient and able to handle unexpected issues. This can be achieved through techniques such as replication, which allows data to be stored on multiple servers to ensure that it is still available even if one server fails.
- **Performance**: Distributed systems can be designed to split processing into independent parts, which can improve overall performance by allowing different parts of the system to operate in parallel.
- **Scalability/throughput**: Distributed systems can be designed to handle large numbers of requests per second, making them well-suited for applications that need to scale to handle high levels

Examples of Distributed Systems

There are many examples of distributed systems, including:

- **Energy grids and telecom networks**: These systems distribute electricity and communication signals across large geographic areas, often using complex networks of wires and other infrastructure.
- **Villages, towns, and cities**: These systems are examples of distributed systems that are made up of many interconnected parts, including homes, businesses, roads, and other infrastructure.
- **IT infrastructure of large companies**: Many large companies rely on distributed systems to manage their IT infrastructure, including servers, storage, and networks.
- **High-performance clusters**: These systems are used to perform complex calculations and simulations, often in scientific or technical fields.
- **Google, Facebook, and other internet companies**: These companies rely on distributed systems to power their online platforms and

Overview

- **Influence of distribution topology and remoteness**: The physical layout of a distributed system and the distance between its components can have a significant impact on its performance and behavior.
- **Emergent behaviors and concurrent events**: Distributed systems can exhibit complex and often unexpected behaviors as a result of the interactions between their components. Concurrent events, in which multiple parts of the system are executing at the same time, can also contribute to this complexity.
- **Few analytic solutions and few model-based approaches**: There are often few analytic solutions or model-based approaches available for understanding and predicting the behavior of distributed systems, which can make them challenging to design and debug.
- **Heterogeneous components**: Distributed systems often consist of a wide variety of components, each with its own hardware, software,

Eight Fallacies of Distributed Computing

- **The network is reliable**: This fallacy assumes that the network is always available and that communication between components will always be successful. In reality, networks can experience outages or other problems that can disrupt communication.
- **Latency is zero**: This fallacy assumes that there is no delay in communication between components, which is not always the case. Latency, or the time it takes for a message to travel from one component to another, can vary depending on the distance between components and other factors.
- **Bandwidth is infinite**: This fallacy assumes that there is an unlimited amount of capacity available for transmitting data, which is not always the case. Network bandwidth is a finite resource that can become congested, leading to slower communication speeds.
- **The network is secure**: This fallacy assumes that the network is invulnerable to security threats, such as hacking or data breaches. In

Programming Languages and Distributed Systems

There are two main approaches to programming languages and distributed systems: the transparency camp and the message camp.

The **transparency camp** focuses on hiding the complexity of a distributed system from the programmer. This can be achieved through techniques such as creating type-safe interfaces and calls, and hiding security, storage, and transactions behind frameworks such as .NET or Enterprise Java Beans (EJBs). This approach treats the distributed system as a programming model, rather than something that requires special handling.

The **message camp**, on the other hand, takes a more direct approach to programming distributed systems. This approach typically involves using a simple create, read, update, delete (CRUD) interface and using message content as the interface. Messages are often coarse-grained, meaning that they carry a large amount of data in a single message, often in the form of

History of Distributed Systems

The history of distributed systems can be divided into several distinct periods:

- **1950s-1980s**: During this period, basic research was conducted on topics such as time, consensus, and computability, which laid the foundations for the development of distributed systems.
- **1990s**: In the 1990s, distributed systems were used to connect Intranet applications using technologies such as Common Object Request Broker Architecture (CORBA), Remote Procedure Calls (RPC), and Distributed Component Object Model (DCOM). Client-server web servers also became popular during this period. Programming models dominated the design and development of distributed systems.
- **2000s**: In the 2000s, distributed systems were used to power peer-to-peer software for file sharing and large social sites emerged, which posed new challenges in terms of scalability and performance.

Metcalfe's Law

Metcalfe's law is a principle that states that the value or utility of a network increases as the number of users in the network increases. This is because the more people who are using the network, the more useful it becomes as a platform for communication, collaboration, and the exchange of information and resources. The adoption rate of a network also tends to increase in proportion to the utility provided by the network, which is why companies often give away software or other products for free in order to increase the size of their user base and the value of their network.

Metcalfe's law is often cited as a factor that can contribute to the emergence of scale-free, or power law, distributions in networks. This type of distribution is characterized by a few nodes (or users) with many connections, and many nodes with only a few connections. The existence of network effects, in which the value of a network increases with the number of users, can help to explain why we don't see many Facebooks or

Security Topics for Distributed Systems

Security is an important concern in distributed systems, as they often handle sensitive data or perform critical functions. Some key security topics that are relevant to distributed systems include:

- **Authentication**: This refers to the process of verifying the identity of a user or device. In a distributed system, authentication may be used to ensure that only authorized users can access certain resources or perform certain actions.
- **Authorization**: This refers to the process of granting or denying access to specific resources or actions based on the identity of a user or device. In a distributed system, authorization controls may be used to ensure that users can only perform actions that are appropriate for their role or level of access.
- **Confidentiality**: This refers to the protection of information from unauthorized access or disclosure. In a distributed system, confidentiality may be achieved through techniques such as

Theoretical Foundations of Distributed Systems

The theoretical foundations of distributed systems are a set of concepts and principles that form the basis for the design and analysis of these systems. Some of the key theoretical foundations of distributed systems include:

- **No global time**: In a distributed system, it is not possible to rely on a single, global clock to coordinate events. Instead, techniques such as logical clocks and vector clocks are used to provide a partial ordering of events within the system.
- **FLP theorem of asynchronous systems**: The FLP (Fischer, Lynch, and Patterson) theorem states that it is impossible to design an asynchronous distributed system that is both safe and live (that is, capable of making progress). This theorem highlights the challenges of building distributed systems that can handle failures or delays in communication.
- **Failure detection and timeout**: One of the challenges of distributed

Distributed Systems Design Fields

The design of a distributed system involves addressing a number of common problems and considering various architectural factors in order to create a system that is scalable, reliable, and secure. Some key considerations in distributed system design include:

- **Common problems**: When designing a distributed system, it is important to consider a number of common problems that can impact the performance, reliability, and security of the system. These include issues such as fail-over, maintenance, policies, and security integration.
- **Information architecture**: The information architecture of a distributed system refers to the way in which information is organized and structured within the system. This includes defining and qualifying the various information fragments and flows that make up the system.
- **Distribution architecture**: The distribution architecture of a

An Introduction to Middleware

Middleware is software that sits between the operating system and the application layer of a distributed system, providing a layer of abstraction that enables communication and resource sharing among the various components of the system. Some key characteristics of middleware include:

- It is used to facilitate the creation of distributed applications.
- Provides glue code and generators that allow different programming languages and systems to interoperate.
- Controls messages and enforces delivery guarantees, such as at-least-once delivery.
- Reorders requests from participants to create a causal or total ordering.
- Takes over responsibility for messages and may store them temporarily.
- Creates groups of nodes that process events together and controls

Distribution Transparencies

Distribution transparencies are features that are designed to hide the complexities of working with distributed systems from the user or developer. Some key distribution transparencies include:

- **Access**: This transparency masks differences in languages and data representation, allowing different systems to communicate and exchange data with each other.
- **Failure**: This transparency masks failures and enables fault tolerance through automated fail-over to other servers.
- **Scalability**: This transparency provides intelligent load balancing of requests to ensure that the system can handle a large number of requests without becoming overloaded.
- **Redundancy**: This transparency transparently replicates data to ensure that it is available even if one or more servers fail.
- **Location**: This transparency allows users to access services using logical, rather than physical, names. This enables services to be

Classification of Middleware

Middleware can be classified into several categories based on the type of service it provides and the way it communicates with other components in a distributed system. Some common types of middleware include:

- **Socket-based services**: These are middleware systems that use sockets to communicate with other components in a distributed system. Sockets are a low-level communication mechanism that allows programs to send and receive data over a network.
- **Remote procedure calls (RPCs)**: These are middleware systems that allow programs to make calls to procedures or functions that are located on a remote machine, as if they were located on the local machine.
- **Object request brokers (ORBs)**: These are middleware systems that enable communication between objects that are running on different machines. ORBs provide an interface that allows objects to communicate with each other using a common protocol. Examples

Theoretical Models of Distributed Systems

Overview

1. Message passing theoretical model
2. Distributed computing topologies
3. Client-server systems, including critical points, architectures, processing and I/O models

Synchronous vs. Asynchronous Systems

Synchronous and asynchronous systems are two types of distributed systems that differ in the way that they handle communication and the passage of time.

In a **synchronous system**, events are assumed to be delivered in a lockstep manner, with a fixed delay between the occurrence of an event and its delivery. This means that events are delivered at predetermined intervals, and the system can be designed to operate on the assumption that events will be delivered at these intervals.

Asynchronous systems do not have a fixed delay between the occurrence of an event and its delivery. Events may be delivered at any time, and the system must be able to handle this uncertainty. Asynchronous systems typically require more complex distributed algorithms to ensure correct operation, but they are generally easier to build and maintain than synchronous systems.

Properties of Message Protocols

Message protocol properties are characteristics that describe the desired behavior of a message passing protocol in a distributed system. These properties are used to ensure that the protocol operates correctly and achieves its intended goals.

Some common message protocol properties include:

- **Correctness**: This property refers to the invariant properties of the protocol, which are properties that are expected to hold throughout all possible executions of the protocol. Ensuring the correctness of a protocol is important for ensuring that the protocol achieves its intended goals.
- **Liveness/termination**: This property refers to the ability of the protocol to make progress in the context of certain failures and within a bounded number of rounds. A protocol that satisfies this property is said to be “lively” or “live”, while a protocol that does not

Complexity of Distributed Algorithms

- **Time complexity** refers to the amount of time it takes for an algorithm to complete. This is often measured in terms of the time of the last event before all processes finish.
- **Message complexity** refers to the number of messages that need to be sent in order for the algorithm to complete. This includes both the number of messages sent and the size of the messages. The number of rounds needed for termination is also an important factor in the message complexity of an algorithm, as it can have a significant impact on the overall scalability of the protocol.

Failure Types

In distributed systems, there are several types of failures that can occur. These failures can have different impacts on the system and can require different approaches to handling them.

Some common types of failures in distributed systems include:

- **Crash failure**: This type of failure occurs when a process stops working and remains down. This can be caused by a variety of issues, such as hardware or software problems.
- **Connectivity failures**: This type of failure occurs when there is a problem with the network that connects the nodes in the system. This can cause "split brain" situations, where the system becomes divided into two separate networks, or node isolation, where a node becomes disconnected from the rest of the system.
- **Message loss**: This type of failure occurs when individual messages are lost during transmission. This can be caused by a variety of

Distributed Computing Topologies

There are several types of distributed computing topologies that can be used to design distributed systems. These topologies can have different characteristics and can be used to achieve different goals, depending on the needs of the system.

Some common types of distributed computing topologies include:

- **Client/server systems**: In this type of topology, clients initiate communication with servers, which process the requests and send a response back to the client. This is the most common type of distributed system and is often used for applications where clients need to request specific information or services from servers.
- **Hierarchical systems**: In this type of topology, every node can act as both a client and a server, but some nodes may play a special role, such as a domain name system (DNS) server. This type of topology can reduce communication overhead and provide options for central

Queuing Theory: Kendall Notation M/M/m/ß/N/Q

The Kendall notation, also known as the Kendall notation for Markov chains, is a way of describing the behavior of a queuing system. It is often used in the field of operations research to analyze the performance of systems that have a finite number of servers and a finite queue size.

- ß: Population Size (limited or infinite)
- M,D,G: Probability distribution for arrivals
- N: Wait queue size (can be unlimited)
- Q: Service policy type (Fifo, shortest remaining time first etc)
- M,D,G: Probability distribution for service time
- m: Number of service channels

Generalized Queuing Theory Terms (Henry Liu)

- **Server/Node**: A combination of a wait queue and a processing element
- **Initiator**: The entity that initiates a service request
- **Wait time**: The time a request or initiator spends waiting in line for service
- **Service time**: The time it takes for the processing element to complete a request
- **Arrival rate**: The rate at which requests arrive for service
- **Utilization**: The percentage of time the processing element spends servicing requests, as opposed to being idle
- **Queue length**: The total number of requests waiting and being serviced
- **Response time**: The sum of the wait time and service time for a single visit to the processing element
- **Residence time**: The total time spent by the processing element on a

Little's Law

- Little's Law states that in a stable system, the long-term average number of customers (L) is equal to the long-term average effective arrival rate (λ) multiplied by the average time a customer spends in the system (W).
- This can be expressed algebraically as $L = \lambda W$.
- Little's Law is used to analyze and understand the behavior of systems that involve waiting, such as queues or lines. It can help to predict the average number of customers in a system, as well as the average time they will spend waiting, given a certain arrival rate.

Hejunka

Hejunka is a Japanese term that refers to the practice of leveling the production process by smoothing out fluctuations in demand and task sizes. It is often used in lean manufacturing and just-in-time (JIT) production systems to improve the efficiency and flow of work through a system.

The goal of Hejunka is to create a steady, predictable flow of work through the system by reducing variability in task sizes and demand. This can be achieved through a variety of methods, such as:

- **Setting limits** on the number of tasks or requests that can be processed at any given time
- **Balancing the workload** across different servers or processing elements
- **Prioritizing tasks** based on their importance or impact on the overall system

Lessons Learned from Queuing Theory

- **Request numbers**: Caching can be used to reduce the number of requests that need to be processed by storing frequently accessed data in memory, so that it can be quickly retrieved without the need to fetch it from a slower storage medium.
- **Batching**: The use of a multi-get API can help to reduce the number of requests that need to be processed by allowing multiple requests to be bundled together and processed as a single unit.
- **Task sizes and variability**: Service level agreements (SLAs) can be used to define the acceptable level of variability in task sizes and completion times, and Hejunka is a technique that involves leveling the production process by smoothing out fluctuations in demand and task sizes. This can help to reduce variability and improve the efficiency of the system.

Request Problem in Multi-Tier Networks

In a multi-tier network, the request problem refers to the fact that **requests must travel through multiple layers or tiers of servers** in order to be processed, and each layer adds its own processing time and potential delays to the overall response time.

The average response time in a multi-tier network is therefore the sum of the trip average (the time it takes for a request to travel from one server to the next) multiplied by the wait time (the time a request spends waiting for a server to become available) at each layer, plus the sum of the service demand (the time it takes for a server to process a request) at each layer.

It is important to note that in a multi-tier network, all requests are synchronous and may be in contention with each other, which means that wait times can occur due to contention for server resources. This can impact the overall efficiency of the system and may require the use of techniques such as caching or batching to reduce the number of requests

Task Size Problem in Multi-Tier Networks

In a multi-tier network, the task size problem refers to the fact that differences in task size can cause delays and inefficiencies in the processing of requests.

In the case of pipeline stalls between nodes, large differences in task size can cause requests to be held up at one node while waiting for the next node to become available, leading to delays in the overall response time.

Theories & Model vs. Reality

When applying queuing theory models to real-world systems, there are several factors that can impact the accuracy and usefulness of the model. These include:

- **Latency**: Latency refers to the time it takes for a request to travel from one server to another or for a task to be completed. Latency can vary based on a variety of factors, such as network speed, server load, and the distance between servers, and it can impact the accuracy of queuing theory models.
- **Blocking/locking/serialization in service units**: In real-world systems, servers may block or lock requests while they are being processed, or may process requests serially rather than in parallel. This can impact the accuracy of queuing theory models that assume parallel processing.
- **Non-random distributions and feedback effects**: In real-world systems, request and task arrival rates may not always follow a

Critical Points in Client/Server Systems

On the **client side**:

- **Locating the server**: The client must be able to locate the server in order to establish a connection. This process can be impacted by factors such as network speed and latency.
- **Authentication**: The client may need to authenticate itself in order to access the server and its resources.
- **Sync/async**: The client may be able to send requests synchronously or asynchronously, which can impact the overall performance of the system.
- **Speed up/down**: The client may be able to adjust the speed at which it sends requests in order to optimize the performance of the system.
- **Load balancing**: The client may need to use load balancing techniques to distribute requests evenly across multiple servers or processing elements.
- **Queues**: The client may need to use queues to manage requests and

Stateful vs Stateless Systems (The Stateful Server Problem)

The stateful server problem refers to the trade-offs that must be considered when designing and implementing a server-based system that maintains state information.

On the one hand, stateful servers have several **advantages**:

- **Data locality**: Stateful servers can store data locally, which can improve the performance of the system by reducing the need to fetch data from external storage.
- **Consistency**: Stateful servers can ensure that data is consistent and up-to-date, which can be important in certain applications.

However, stateful servers also have some **disadvantages**:

- **Availability**: Stateful servers may be less available than stateless servers, as they may be more vulnerable to failures or downtime.
- **Load balancing**: Stateful servers may be more difficult to load

Terminology for Client/Server Systems

- **Host**: A physical machine with n CPUs.
- **Server**: A process running on a host that receives messages, performs computations, and sends messages (not necessarily responses).
- **Thread**: An independent computation context within a process, which can be pre-empted by the kernel (kernel-thread) or yield voluntarily (application-level scheduling).
- **Multi-threading**: The use of multiple threads within a single process context. This can be achieved using kernel-level threading (where the kernel switches between threads) or using multiple kernel threads running in parallel on a multi-core system.
- **Multi-channel**: A thread that is able to watch multiple channels using a single system call, such as a `select()` call.
- **Synchronous processing**: A caller calls a function and waits for its results, doing nothing while waiting.
- **Asynchronous processing**: A caller calls a function and immediately

Overarching Client/Server Architectures

- **Multi-Tier System**: This type of architecture involves splitting up the system into multiple layers or tiers, each of which performs a specific set of functions. The tiers may include a presentation layer, a business logic layer, and a data storage layer, among others.
- **Large fan-out Architectures**: This type of architecture involves a central component that receives requests from many clients and distributes them to multiple servers or other resources. This can allow the system to scale more easily and handle a large volume of requests.
- **Pipeline (SEDA)**: This type of architecture involves breaking up the processing of a request into multiple stages, each of which is handled by a separate component. The stages are connected in a pipeline, with each stage processing the request and passing it on to the next stage.
- **Offline Processing**: This type of architecture involves processing

Different Process Models

- **Single Thread/Single Core**: This type of process model involves a single thread of execution running on a single core. This can be efficient for certain types of workloads, but may not be able to take full advantage of multiple cores or processors.
- **Multi-Thread/Single Core**: This type of process model involves multiple threads of execution running on a single core. This can allow the system to perform multiple tasks concurrently, but may not be able to fully utilize the processing power of multiple cores or processors.
- **Multi-Thread/Multi-Core**: This type of process model involves multiple threads of execution running on multiple cores or processors. This can allow the system to fully utilize the processing power of multiple cores or processors, and can be more efficient for certain types of workloads.
- **Single Thread/Multi-Process**: This type of process model involves a

Questions for Process Models

- Can it use available cores/CPUs?
- What is the ideal number of threads?
- How does it deal with delays/(b)locking?
- How does it deal with slow requests/uploads?
- Is there observable non-determinism aka race conditions?
- Is locking/synchronization needed?
- What is the overhead of context switches and memory?

Amdahl's Law

According to Amdahl's Law, the maximum improvement in overall system performance that can be achieved by improving a particular part of the system is limited by the fraction of time that the improved part of the system is used. In other words, if only a small portion of the system's workload is affected by the improvement, the overall improvement in performance will also be small.

$$speedup = \frac{1}{(1 - parallel fraction + \frac{parallel fraction}{number of processors}}$$

For example, if a particular part of a system is improved so that it runs twice as fast, but that part of the system is only used 10% of the time, the overall improvement in system performance will be limited to a 10% increase. On the other hand, if the improved part of the system is used 50% of the time, the overall improvement in performance will be much larger, at 50%.

Amdahl's Law is often used to understand the potential benefits and

Different I/O Models

- **Synchronous Blocking (Java before NIO/AIO)**: Prior to the introduction of the Java New I/O (NIO) and Asynchronous I/O (AIO) APIs, Java had a different model for handling input/output (I/O) operations. This model involved using threads to block and wait for I/O operations to complete, which could be inefficient and consume a lot of system resources.
- **Synchronous Non-Blocking (Polling pattern)**: The polling pattern is a way of handling I/O operations in which a central component periodically checks for the completion of I/O operations. This can be done by repeatedly calling a function that checks the status of the operation, or by using a timer to trigger the check at regular intervals.
- **Asynchronous Blocking (Reactor pattern)**: The Reactor pattern is a way of handling I/O operations in which a central component is notified when an I/O operation is completed, rather than periodically checking for its completion. This can be more efficient than the

Questions for I/O Models

- Can it deal with all kinds of input/output?
- How are synchronous channels integrated?
- How hard is programming?
- Can it be combined with multi-cores?
- Scalability through multi-processes?
- Race conditions possible?

Message Protocols

Overview

1. Message protocols
    1.1 Delivery guarantees in point-to-point communication
    1.2 Reliable broadcast
    1.3 Request ordering and causality
2. Programming client-server systems using sockets

The Role of Delivery Guarantees

Shop order: The scenario described involves an online shop in which orders are placed and processed. The goal is to ensure that orders are delivered correctly and efficiently, regardless of any potential issues that may arise.

- **TCP Communication properties**: TCP (Transmission Control Protocol) is a networking protocol that is used to establish and maintain communication between devices over a network. It has several key properties that are relevant to the scenario described, including reliability, flow control, and congestion control.
- **At-least-once**: The “at-least-once” delivery guarantee means that a message may be delivered more than once, but it will always be delivered at least once. This can be useful in situations where it is more important to ensure that a message is delivered, even if it may be duplicated, than it is to prevent duplicates from occurring.
- **At-most-once**: The “at-most-once” delivery guarantee means that a

Why is TCP not Enough?

While TCP (Transmission Control Protocol) is a widely used networking protocol that provides a reliable communication channel between devices, it is not always sufficient on its own to ensure proper behavior in all situations. Here are some reasons why TCP may not be enough:

- **Lost messages retransmitted**: TCP includes mechanisms for retransmitting lost messages, which can help to improve the reliability of communication. However, if messages are frequently lost or the network is particularly unreliable, the overhead of retransmitting lost messages may become a burden on the system.
- **Re-sequencing of out of order messages**: TCP includes mechanisms for reordering out-of-order messages, which can help to ensure that messages are delivered in the correct order. However, if messages are frequently delivered out of order, this can be inefficient and may cause issues with the overall communication process.
- **Sender choke back (flow control)**: TCP includes flow control

Different Levels of Timeouts

- **Business-Process-Timeout**: This timeout is set at the business process level and is used to ensure that a business process does not get stuck or take too long to complete. This timeout may be triggered if a particular task or operation within the process takes longer than expected to complete, or if the process as a whole takes too long to finish.
- **RPC-Timeout (order progress)**: This timeout is set at the level of remote procedure calls (RPCs) and is used to ensure that RPCs do not get stuck or take too long to complete. This timeout may be triggered if an RPC takes longer than expected to complete, or if the progress of an RPC is not being monitored properly.
- **TCP-Timeout (reliable channel)**: This timeout is set at the TCP (Transmission Control Protocol) level and is used to ensure that the reliable communication channel provided by TCP is functioning properly. This timeout may be triggered if a connection is lost or if

Delivery Guarantees for RPCs

- **Best effort**: The “best effort” delivery guarantee means that no specific guarantees are made about the delivery of requests or responses. This means that requests may be lost or responses may not be received, and there is no mechanism in place to ensure that this does not happen.
- **At least once**: The “at least once” delivery guarantee means that a request may be delivered more than once, but it will always be delivered at least once. This can be useful in situations where it is more important to ensure that a request is delivered, even if it may be duplicated, than it is to prevent duplicates from occurring.
- **At most once**: The “at most once” delivery guarantee means that a request will be delivered at most once. This can be useful in situations where it is more important to prevent duplicates from occurring than it is to ensure that a request is always delivered.
- **Once and only once/exactly once**: The “once and only once” or

Overview

Idempotency is a property of operations or requests that ensures that they can be safely repeated without changing the result of the operation. In other words, if an operation is idempotent, it will have the same result whether it is performed once or multiple times.

- The first request needs to be idempotent: In a sequence of requests, it is important that the first request is idempotent. This ensures that the first request can be safely repeated if it fails or is lost, without affecting the overall result of the operation.
- The last request can be only best effort
- Messages may be reordered

Server State and Idempotency

Idempotency is an important property to consider when designing operations or requests that may be repeated or delivered multiple times, as it can help to ensure that the operation or request is able to be safely repeated without affecting the overall result. Here are some additional considerations related to idempotency and server state:

- **No need to remember a request and its result**: If an operation or request is idempotent, the server does not need to remember the request or its result. This can be useful in situations where the server's storage is limited or unreliable, as it means that the server does not need to maintain a record of all previous requests and their results.
- **Server can lose its storage**: If the server's storage is lost or becomes unavailable, it should not affect the overall result of the operation or request, as long as the operation or request is idempotent. This can help to ensure that the operation or request is able to be safely

Implementing Delivery Guarantees for Idempotent Requests

- **“At least once”** implementation for idempotent requests: For idempotent requests, the “at least once” delivery guarantee can be implemented by simply sending an acknowledgement (ack) to the client after the request has been received. This approach does not require any updates to the server state, and is suitable for requests that do not have any critical side effects.
- **“At most once”** implementation for nonidempotent requests: For nonidempotent requests, the “at most once” delivery guarantee can be implemented by storing a response on the server until the client confirms that it has been received. This approach requires the server to maintain state for each response, and may involve adding a request number to each request to help the server detect and discard duplicate requests.
- **“Exactly once”** implementation: The “exactly once” delivery guarantee is not possible to achieve in asynchronous systems with

Repeating Non-Idempotent Operations

If an operation is not idempotent, it means that it cannot be safely repeated multiple times and is likely to have unintended side effects. In this case, there are several measures that can be taken to ensure reliable communication:

- **Use a message ID** to filter for duplicate sends: By including a unique message ID in each request, the server can filter out duplicates and only execute the request once.
- **Keep a history list of request execution results** on the server: If the reply to a request is lost, the server can retransmit the result from its history list. This helps to ensure that the client receives the correct result even if the initial reply was lost.
- **Lease resources on the server**: In some cases, it may be necessary to keep state on the server in order to facilitate communication. For example, a client may “lease” resources on the server for a specific period of time. This can help to ensure that resources are used

Request Order in Multi-Point-Protocols

- **No request order from one sender**: In a multi-point protocol, there is no guaranteed order for requests sent by a single sender. This means that requests may be received and processed in a different order than they were sent, and the sender should be prepared to handle this possibility.
- **No request order between different senders**: In a multi-point protocol, there is no guaranteed order for requests sent by different senders. This means that requests from different senders may be received and processed in a different order than they were sent, and the senders should be prepared to handle this possibility.
- **No request order between independent requests of different senders**: In a multi-point protocol, there is no guaranteed order for independent requests sent by different senders. This means that independent requests from different senders may be received and processed in a different order than they were sent, and the senders

Request Ordering with Multiple Nodes

In a multi-node system, it may be necessary to use a reliable broadcast protocol to ensure that requests are processed in the desired order. Here are some examples of protocols that can be used for request ordering with multiple nodes:

- **Reliable Broadcast**: Reliable broadcast is a protocol that ensures that a message is delivered to all nodes in the system, and that it is delivered in the same order to all nodes. This can help to ensure that requests are processed in the correct order, even if they are sent from different nodes or if there are delays or other issues with the network.
- **FIFO Cast**: FIFO cast is a protocol that ensures that messages are delivered in the order in which they were sent, with the first message sent being the first one to be delivered. This can help to ensure that requests are processed in the correct order, even if they are sent from different nodes or if there are delays or other issues with the

Implementing Causal Ordered Broadcasts

- **Piggybacking previous messages**: One solution for implementing causal ordered broadcasts is to piggyback every message sent with the previous messages. This means that when a message is sent, it is accompanied by the previous messages that it depends on. This can help to ensure that processes that may have missed a message can learn about it with the next incoming message and then deliver it correctly.
- **Sending event history with every message**: Another solution for implementing causal ordered broadcasts is to send the event history with every message. This can be done using techniques such as vector clocks, which are used to track the dependencies between events in a distributed system. With this approach, messages are not delivered until the order is correct. This can help to ensure that messages are delivered in the correct order, even if there are delays or other issues with the network.

Implementing Absolutely Ordered Casts

- **All nodes send messages to every other node**: One solution for implementing atomic broadcasts is for all nodes to send their messages to every other node in the system. This ensures that all nodes have a complete set of messages, which can be used to determine the total order of the messages.
- **All nodes receive messages, but wait with delivery**: After receiving all of the messages, all nodes can wait with delivery until the total order of the messages has been determined.
- **One node is selected to organize the total order**: To determine the total order of the messages, one node can be selected to organize the messages into a total order. This node can use a variety of techniques, such as vector clocks or Lamport timestamps, to determine the order of the messages.
- **The node sends the total order to all nodes**: Once the total order has been determined, the node can send the total order to all other

Properties of Sockets

Sockets are a programming interface that enables communication between networked computers. They are used to send and receive data over a network connection using either TCP (Transmission Control Protocol) or UDP (User Datagram Protocol) connections.

Socket properties include:

- **Using either TCP or UDP connections**: Sockets can use either TCP or UDP connections to communicate with other computers over a network. TCP provides a reliable, connection-oriented communication channel, while UDP provides a connectionless, unreliable communication channel.
- **Serving as a programming interface**: Sockets provide a programming interface that enables applications to send and receive data over a network connection. They are typically used in conjunction with other programming constructs, such as threads or event loops, to

Berkeley Sockets

Berkeley sockets provide a set of primitives or functions that can be used to create, manage, and manipulate network connections and communication channels.

Here is a brief description of the meaning of each of the Berkeley Sockets primitives:

- **Socket**: The socket primitive creates a new communication endpoint, or socket, that can be used to send and receive data over a network connection.
- **Bind**: The bind primitive attaches a local address to a socket, specifying the address and port on which the socket will listen for incoming connections or data.
- **Listen**: The listen primitive announces the willingness of a socket to accept incoming connections from other hosts.
- **Accept**: The accept primitive blocks the caller until a connection

Server Side Processing using Processes

Server-side processing using processes is a model for handling incoming requests in a networked system. It involves the following steps:

1. Server: The server listens on a specified port, waiting for incoming connection requests.
2. Client: A client connects to the server on an arbitrary port and establishes a connection between the client and the server.
3. Server: The server accepts the connection request and spawns a new process to handle the request. The process is assigned to the same port as the original connection, and the server goes back to listening for new connection requests.

This model allows the server to scale to some degree, as it can handle multiple requests concurrently by spawning new processes to handle each request. However, process creation can be expensive, and there may be limits on the number of processes that can be created on a given system.

Server Side Processing using Threads

Server-side processing using threads is a model for handling incoming requests in a networked system that is similar to the process-based model, but uses threads instead of processes to handle requests. It involves the following steps:

1. Server: The server listens on a specified port, waiting for incoming connection requests.
2. Client: A client connects to the server on an arbitrary port and establishes a connection between the client and the server.
3. Server: The server accepts the connection request and spawns a new thread to handle the request. The thread is assigned to the same port as the original connection, and the server goes back to listening for new connection requests.

This model allows the server to scale well, as it can handle multiple requests concurrently by spawning new threads to handle each request.

Design Considerations for Socket-Based Services

- **Message formats**: The message formats that will be exchanged between clients and servers should be carefully designed to ensure that data can be transmitted and received correctly. This includes ensuring that data is represented consistently on different hardware platforms and avoiding problems caused by different data representations.
- **Protocol design**: The protocol that will be used by clients and servers to communicate should also be carefully designed. This includes deciding whether clients will wait for answers from the server (synchronous communication) or whether communication will be asynchronous, whether the server can call back to the client, whether the connection will be permanent or closed after each request, whether the server will hold client-related state (e.g. a session), and whether the server will allow concurrent requests.

Stateless vs. Stateful Socket-Based Services

A stateless service is one in which the server does not store any information about previous requests or maintain any state between requests. This can have several advantages, including:

- **Scaling extremely well**: Because the server does not maintain any state between requests, it can handle a large number of requests concurrently without running out of resources or becoming overloaded.
- **Making denial of service attacks harder**: Because the server does not maintain any state between requests, it is more resistant to denial of service attacks, as attackers cannot exhaust resources by sending a large number of requests that require the server to store state.
- **Forcing new authentication and authorization per request**: A stateless service can require clients to authenticate and authorize each request separately, improving security by requiring clients to prove their identity and permissions for each request; this can

TCP SYN Flooding

TCP SYN flooding is a type of denial of service (DoS) attack that exploits a weakness in the TCP connection establishment process. In a normal TCP connection, the client sends a SYN (synchronize) packet to the server to initiate the connection, and the server responds with a SYN-ACK (synchronize-acknowledge) packet to confirm that the connection can be established. The client then **sends an ACK** (acknowledge) packet to complete the three-way handshake and establish the connection.

In a TCP SYN flooding attack, the attacker sends a large number of SYN packets to the server, either from a single machine or from a network of compromised machines. The server responds to each SYN packet with a SYN-ACK packet, **but the client never completes the three-way handshake** by sending an ACK packet. As a result, the server is left waiting for the ACK packet, and the resources used to track the half-open connections can be exhausted, preventing the server from accepting any new connections.

Writing a Socket Client & Server

For the server:

1. **Define the port number of the service**: The server should specify the port number on which it will listen for incoming connections. For example, the HTTP server typically listens on port 80.
2. **Allocate a server socket**: The server creates a server socket and binds it to the specified port number. The server socket then listens for new connections.
3. **Accept an incoming connection**: When a client attempts to connect to the server, the server socket accepts the connection and creates a new socket for the client connection.
4. **Get the input channel**: The server reads the input channel from the socket to receive messages from the client.
5. **Parse the client message**: The server parses the client message to understand the request being made.
6. **Get the output channel**: The server gets the output channel from the

Distribution Transparency with Sockets

- **Invocation**: Server-side functions cannot be called directly from the client side, and messages must be defined and sent using socket operations
- **Location/relocation/migration**: If the service moves, the client connection will be broken
- **Replication/concurrency**: Not supported
- **Failure**: Not supported
- **Persistence**: Not supported

Infrastructure of Client-Server Systems

1. **Directory**: Helps locate the server
2. **Proxy**: Checks client authorization and routes requests through the firewall
3. **Firewall**: Allows outgoing calls only
4. **Reverse proxy**: Caches results, ends SSL sessions, and authenticates clients
5. **Authentication server**: Stores client data and authorizes clients
6. **Load balancer**: Distributes requests across servers

Remote Procedure Calls

Overview

1. Call versions: local, inter-process, and remote
2. Mechanics of remote calls:
    2.1 Marshaling/serialization
    2.2 Data representation
    2.3 Message structure and schema evolution
    2.4 Interface definition language
    2.5 Tooling: generators
3. Cross-language call infrastructures: Thrift and gRPC

Call Versions

1. **Local calls**: made within a single process and do not require network communication
2. **Inter-process calls**: Made between processes on the same machine and can be implemented using a variety of mechanisms, such as shared memory or pipes
3. **Remote calls**: Made between processes on different machines and typically require network communication

Remote Calls vs. Remote Messages

- **Remote calls**: Hide the remote service calls behind a programming language call and are tightly coupled with synchronous processing
- **Remote messages**: Use a message-based middleware and create a new concept of a message and its delivery semantics; a message system can simulate a call-based system, but not vice versa

In-Process (Local) Calls

- **No special middleware is required** for calls made within the same programming language
- Calls into the OS are not inter-process calls
- Special attention is required for cross-language calls made within a single process, such as calls to native code in Java
- In-process calls **are fast** because they do not require network communication or context switches between processes.
- Have **“exactly once”** semantics, meaning that they will be executed only once and will not be retried if they fail.
- Are **type-safe and link-safe**, meaning that they are checked for type compatibility at compile time and will not result in linker errors. However, they may have issues with dynamic loading if the called code is not available at runtime.
- Can be **either sequential or concurrent**, depending on the design of the application. Concurrent in-process calls can be implemented

Inter-Process Calls (IPC)

- Local inter-process calls are faster than remote calls, but **not as fast as in-process calls**
- Do **not have "exactly once"** semantics
- Are **type-safe and link-safe** if both processes use the same static libraries, but may have issues with dynamic loading
- Can be **either sequential or concurrent**, but the caller no longer controls the execution; the receiver must protect itself from concurrent access
- **Cannot assume a single name and address space** and require additional addressing or mapping
- Are still **independent of byte ordering**
- **Require cross-process garbage collection** to manage memory usage
- **Can only use value parameters**, as the target process cannot access memory in the calling process
- Are **not real programming language "calls"** and require the creation

Remote Procedure Calls (RPC)

- Remote procedure calls (RPCs) are **much slower than local calls** and **do not have delivery guarantees** without a protocol
- May have **version mismatches** that show up at runtime
- Are **concurrent** and the caller no longer controls the execution; the callee must protect itself from concurrent access
- **Cannot assume a single name and address space** and require additional addressing or mapping
- Are **affected by byte ordering**
- May **require network garbage collection** if they are stateful
- May **involve cross-language calls**
- **Can only use value parameters**, as the target process cannot access memory in the calling process
- **Are not real programming language "calls"** and require the creation of messages to simulate the missing features
- Often stateless

Overview

- **Marshaling/Serialization**: Process of converting program data into a format that can be transmitted over a network
- **External Data-Representation**: Standardized format for representing binary data
- **Interface Definition**: Defines a service, a set of related functions that can be accessed remotely
- **Message Structure and Evolution**: Way in which data is organized and transmitted between systems
- **Compilers**: Generate code (stub/skeleton or proxy) to facilitate communication between systems
- **Request/Reply Protocol**: Handles errors that may occur during the remote call process
- **Process/I/O Layer**: Responsible for managing threads and input/output operations

Marshaling/Serialization

Marshaling/serialization is the process of converting parameters (basic types or objects) into a common transfer format (message) that can be transmitted over a network. At the target site, the message is transformed back into the original types or objects. There are several different approaches to marshaling/serialization, each with its own trade-offs:

- **Language-dependent output format**: This approach uses a proprietary format that is specific to a particular programming language. It may be slower and have limitations in expressiveness, but it can be more efficient in terms of the size of the message.
- **Language-independent output format**: This approach uses a standardized format that is not tied to any particular programming language. It may be more verbose and result in larger messages, but it is more flexible and can be used with a wider range of languages.
- **Binary schema-based**: In this approach, the sender and receiver both have a shared understanding of the structure of the message.

Schema Evolution

When data or functions change, it is important to consider how different versions of a system will coexist and communicate with each other.

Forward compatibility means that older receivers should be able to understand messages from newer senders. This allows newer versions of a system to be deployed without causing problems for existing clients.

Backward compatibility means that newer receivers should be able to understand messages from older senders. This allows older versions of a system to continue functioning even after newer versions have been introduced.

For JSON

Forward compatibility means that older versions of a system should be able to understand messages from newer versions. Examples of changes that can be made to a system in a way that maintains forward compatibility:

- **Adding a new required field**: Older readers will simply ignore the new field, so they will be compatible with newer versions.
- **Narrowing a numerical type (e.g. float to int)**: Older readers will assume ints, which are a subset of floats, so they will be compatible with the newer int type.

Backward compatibility means that newer versions of a system should be able to understand messages from older versions. Examples of changes that can be made to a system in a way that maintains backward compatibility:

Stubs and Skeletons

Stubs and skeletons are code that is used to facilitate communication between different systems, typically in the context of remote procedure calls (RPCs). Stubs are used by clients to initiate a remote call, while skeletons are used by servers to receive and process the remote call.

There are several ways to generate stubs and skeletons, including:

- - **Generating them in advance from an interface definition language (IDL) file**: In this approach, the stubs and skeletons are generated from an IDL file, which defines the interfaces and data structures that are used for communication. The IDL file is used as a blueprint for generating the code.

- **Generating them on demand from a class file**: In this approach, the stubs and skeletons are generated from a class file, which defines the classes and methods that are used for communication. The class file is used as a blueprint for generating the code.

Finding an RPC Server

In a remote procedure call (RPC) system, a client needs to be able to locate the server in order to initiate a remote call.

Service:

1. **Start listening at port X**: The server starts listening for incoming requests on a specific port.
2. **Tell portmapper about program, version, and port**: The server registers itself with the portmapper, which is a service that keeps track of the programs and ports that are available on the system. The portmapper is typically a separate daemon that runs on the system.

Client:

1. **Ask portmapper for program, version**: The client sends a request to the portmapper asking for the port number of the desired program and version.

Factors to Consider when Choosing an IDL

- **Are data types easily expressed using the IDL?** It is important to choose an IDL that can easily represent the types of data that will be used in the RPC system.
- **Is hard or soft versioning used?** Hard versioning means that the IDL is strictly enforced, and any changes to the IDL will break compatibility. Soft versioning means that the IDL is more flexible and can be changed without breaking compatibility.
- **Are structures self-describing?** Self-describing structures contain enough information to be understood by any system, even if it is not familiar with the specific structure. This can be useful for maintaining compatibility between different versions of a system.
- **Is it possible to change the structures later and keep backward compatibility?** It is important to choose an IDL that allows for changes to be made to the structures in a way that maintains backward compatibility with older versions of the system.

Popular IDLs for RPCs

- **CORBA (Common Object Request Broker Architecture)**: CORBA is a standard for interoperating between different programming languages and platforms. It defines an interface definition language (IDL) that can be used to describe the interfaces and data structures that are used for communication, as well as a runtime system for executing the RPCs.
- **Microsoft CLR (Common Language Runtime)**: The CLR is a runtime environment for executing .NET programs. It includes a cross-language call infrastructure that allows programs written in different .NET languages to communicate with each other.
- **Thrift**: Thrift is a cross-platform RPC framework that allows different programming languages to communicate with each other. It includes an IDL for defining the interfaces and data structures that are used for communication, as well as code generators for generating the stubs and skeletons that are used to initiate and process the RPCs.

Distributed Objects

Overview

1. Fundamental Properties of Objects
2. Local and remote object references
3. Parameter passing
4. Object invocation types
5. Distributed Object Services
6. Object Request Broker Architectures
7. Interface Design
8. Java RMI

Objects vs. Abstract Data Types

- Objects are data structures that combine state (data) and behavior (methods or functions) in a single entity. They are a key concept in object-oriented programming (OOP).
- Abstract data types (ADTs) are data structures that are defined by the operations that can be performed on them, rather than by their implementation. ADTs are often used to model real-world concepts or objects.

There are some differences between the two:

- **Objects have an identity**, which is a unique identifier that distinguishes the object from other objects in the system. ADTs do not have an identity in the same way that objects do.
- **Objects may store state** (data) within themselves, while ADTs do not store state. Instead, they define the operations that can be performed on data stored elsewhere.

Properties of Objects

- Local objects are created with the new() operator and are **only accessible within the scope of their creator**.
- An **object reference** is a unique identifier that is used to locate and access an object. It is returned when the object is created and can be used to call the object's methods.
- Objects have a **lifecycle** that is tied to their creator. They exist as long as the virtual machine (VM) is alive and the objects are in use, and they are typically destroyed when their creator is destroyed.
- Objects have **fine-granular interfaces and methods**, which means that they expose a large number of individual functions or operations that can be performed on them.
- Objects can be **small and numerous**, which means that there may be many objects in a system, each with its own state and behavior.

Objects in an object-oriented (OO) programming language do have many properties that can make them challenging to implement in a concurrent

Challenges for Remote Objects

Remote objects (ROs) are objects that are accessed and managed remotely, typically over a network. They are a key concept in distributed object systems, which are systems that enable objects on different machines to communicate and interact with each other.

- Object **identity is usually only valid locally**, meaning that it is only meaningful within the scope of the machine on which the object is stored. This can make it difficult to identify and access ROs from a remote machine.
- ROs **must be created and managed by a server**, rather than by a client. This is because the client does not have direct access to the objects and cannot create them using the new() operator.
- Clients **must have a way to find and access ROs**, which may involve using a registry or other lookup service.
- **Concurrent access to ROs must be controlled** to prevent conflicts and ensure the consistency of the objects' state. This may involve

What is a Remote Object?

- A remote object is an object that is accessed and managed remotely, typically over a network.
- It is a combination of a **unique identity**, an **interface**, and an **implementation**.
- The unique identity is used to locate and access the object from a remote machine.
- The interface defines the operations that can be performed on the object.
- The implementation is the code that defines how the object behaves and carries out these operations.
- **Clients** know the interface and use the identity of the remote object, but **do not know about the implementation**.
- To clients, the interface of the remote object represents the entire object.
- Achieving complete transparency of remote calls behind object

CORBA:

- Defines its **own basic types**, including sequence, string, array, record, enumerated, and union.
- Uses **value objects (data)** to represent data that is passed between objects.
- Uses **remote object references** (reference semantics) to locate and access objects on remote machines.

CORBA is designed to provide language independence, so it defines its own types and **does not allow user-defined classes** to be used in the interfaces of remote objects.

Java RMI:

- Uses the **basic types of the Java language**, such as int, byte, etc.
- Allows **serializable non-remote objects** (value semantics) to be used

Accessing Remote Objects

- A **naming service** is a service that acts like a directory and allows clients to look up the location of a remote object based on its name or other identifier. The client can then use the location information to access the object.
- A **web server** can be used to host serialized versions of remote objects. The client can request the object from the server over the web and deserialize it to access its methods and state.
- Remote objects can be accessed through other means, such as via mail or a piece of paper. For example, a client might send a request for a remote object to another machine via mail or other physical means, and the server could return the object or a reference to the object in the same way.
- Another **remote object can serve as a "factory"** that creates and manages other remote objects. The client can access the factory object and use its methods to create and access other remote

The Broker Pattern

- Is a design pattern that involves using a separate component (the broker) to mediate communication between two other components (the client and the service).
- Serves as an **intermediary** between the client and the service and is responsible for routing requests and responses between them.
- **Decouples the client and the service**, allowing them to evolve independently and communicate through the broker.

Remote Object Reference

- **Object implementation (servant)**: The object implementation is the code that defines how the object behaves and carries out its operations. This is also known as the object's "servant," as it serves the object and carries out its requests.
- **Object adapter**: The object adapter is a component that manages the communication between the object implementation and the object request handler (ORB). It is responsible for routing requests from the ORB to the object implementation and returning responses to the ORB.
- **Active object map (remote object table)**: The active object map is a data structure that maps object identifiers to object implementations. It is used to locate the object implementation for a given object identifier.
- **Object request handler (ORB)**: The ORB is the component that handles requests to and from remote objects. It is responsible for

Static vs Dynamic Invocation

Static Remote Method Invocation

- Involves using **pre-generated stubs** that are linked to the client program in advance.
- The client program uses the stubs to invoke the remote method in the same way that it would invoke a local method.
- The **stubs handle the details of marshalling and demarshalling** the request and response, as well as routing the request to the appropriate object on the remote machine.

Dynamic Invocation:

- Involves **building a request object at runtime**, based on the meta-information of the remote object.
- The request object is sent to a dispatcher on the servant host, which handles the request as if it were a normal method invocation.
- The dynamic invocation approach is **similar to the reflection pattern**.

Asynchronous Invocations

- **One-way calls**: One-way calls are calls that do not expect a response from the server. They are called “one-way” because they involve only a single message being sent from the client to the server. One-way calls cannot have return values or out parameters, and their delivery is best-effort (meaning that there is no guarantee that the message will be delivered).
- **Deferred synchronous**: Deferred synchronous invocations involve making a call to a remote method and continuing to execute while the method is being executed. The client can later check for the results of the method (blocking), but the delivery is at-most-once (meaning that the message will be delivered at most one time).
- **True asynchronous with server callbacks**: True asynchronous invocations involve making a call to a remote method and continuing to execute while the method is being executed. The server can differentiate between synchronous and asynchronous calls and can

Main Distributed Object Services

- Finding objects:
  - **Naming service**: Maps names to object references
  - **Trading service**: Allows objects to offer their services and allows clients to search for objects by constraint
- Preserving object state
  - **Persistence service**: Stores object state transparently and allows it to be loaded on demand
  - **Transaction service**: Preserves object consistency across changes to multiple objects in a distributed, nested, or flat context
  - **Concurrency service**: Provides locks for shared objects
  - **Security service**: Checks the roles of principals
- Grouping of objects: Collections

Portable Interceptors

Interceptors can be used to **transparently add additional (context) information** to calls and transport it between object request brokers (ORBs).

Remote Interface Design

- Respect the **possibility of concurrent calls** in your interface design: Avoid keeping inconsistent state across method calls.
- Avoid the **“half-baked object” anti-pattern**: Do not perform staged initialization of an object.
- Avoid using complicated or **unclear orders of calls**: Design the interface in a clear and straightforward manner.

Problems with Remote Objects

- **Interfaces**: Remote object interfaces can be too granular, with many small methods that perform simple tasks. This can lead to slow performance, as each call to a remote object involves a significant amount of overhead.
- **No direct support for state handling on servers**: Distributed object systems do not generally provide direct support for managing the state of remote objects on the server side. This can make it difficult to maintain consistent state across multiple remote objects or to persist the state of an object across different contexts.
- **Bad for "data schlepping" applications**: Remote objects are often too expensive to use in applications that involve "data schlepping," or the transfer of large amounts of data between the client and server. This is because each call to a remote object involves significant overhead, which can make it inefficient to transfer large amounts of data.
- **Cross-language calls expensive to build**: Making calls between

Java RMI Request/Reply Protocols

JRMP (Java Remote Method Protocol) is the first protocol for RMI. It has the following characteristics:

- **Bandwidth problems** due to distributed garbage collection with short term leases and permanent reference counting.
- Allows for the **dynamic download of code**.

RMI-IIOP (RMI over CORBA's Internet Inter-Orb Protocol) has the following characteristics:

- Uses **Java Naming and Directory Interfaces (JNDI)** to lookup object references and is persistent.
- Requires code changes and the use of PortableRemoteObject.
- **Requires the generation/definition of code** and IDL files for CORBA systems.
- Allows for the movement of IDL files for Java Remote Object Interfaces to CORBA systems, and the generation of CORBA stubs

Java RMI Classes & Tools

- **Remote**: Tag interface that Remote Object Interfaces extend.
- **RemoteException**: Class that is thrown by all Remote Object methods.
- **Naming**: Class that is used by clients to find remote object references, and by servers to register their objects.
- **UnicastRemoteObject**: Class that Remote Object Implementations extend.
- **rmic**: Tool that generates stub/skeleton/IDL files.
- **registry**: Simple name server for Java objects.

Activation in Java RMI

Activation is an important feature in Java RMI (Remote Method Invocation) because it allows servers to transparently store servant state on persistent storage and recreate servants on demand. This helps the server **control its resources** against memory exhaustion and performance degradation.

Security in Java RMI

- Specify the quality of service (QOS) of sockets used by RMI, such as **using an SSL channel**.
- Use an **RMISecurityManager** to prevent or control local access from downloaded implementations.

Distributed Business Components

Overview

- **Part One**: General Component Technology covers the following topics:
  - The limitations of traditional object-oriented programming.
  - Distributed components.
  - Mapping business concepts to programming constructs.
- **Part Two**: Enterprise Java Beans Example covers the following topics:
  - Object model of Enterprise Java Beans (EJBs).
  - Basic mechanisms of EJBs.
  - Separation of concerns in EJBs, including persistence, transactions, and security.
  - Separation of context in EJBs, including the environment.
  - Evolution and lessons learned from EJBs.

Problems With Object Interfaces

- **Object interfaces** are tightly intertwined networks of references that share state and promises, and changes to these interfaces can have **ripple effects**.
- A **component framework** can simplify the interface for calling objects by **encapsulating** them.
- A **messaging system** can be stateless, or it can include all necessary state in the message itself, which is known as **context-complete communication**. This approach is used in webservices and REST architectures and is known as Service Oriented Architecture (SOA).

Component Based Processing

- Components are **self-contained software packages** with runtime interfaces and automatic deployment capabilities that are designed to fit into a component framework.
- A **component framework** allows components to be plugged in and supports the **composition and collaboration** between them.
- There are **roles** for development, composition, and installation of components in the component model.

Enterprise components build on this:

- **Integration with existing infrastructure**, such as transactions, security, and legacy systems
- **Network addressable** interfaces
- **Medium to large granularity**, potentially representing 10-20 tables or more
- Representation of a business concept in an **isomorphic** manner

From Objects to Components

1. **Object-oriented design** involves isolated, monolithic applications that are not distributed.
2. **Distributed objects** involve calls between applications, but can have management and performance issues with large numbers of small remote objects.
3. **Distributed systems** involve multi-tier systems with point-to-point connectivity, but can be expensive and difficult to develop.
4. **Distributed components** use a framework for pluggable business components and create a market for interoperable components. This approach addresses modeling, development, and deployment and aims to achieve re-use and lower development costs.

Components **go beyond distributed systems** and were designed to address some of the same issues as object-oriented development.

Business Components

- Business concepts, such as entities and processes, can be translated into software artifacts like EC1 and PC1.
- **Business components** are designed to **directly represent concepts** from the business and may be represented using the UML “package” construct.

Alternatives to Isomorphic Mapping

- Isomorphic mapping, or the idea that software artifacts directly correspond to business concepts, **may not always be the best approach**.
- Alternatives to isomorphic mapping include **domain analysis, generative computing, and aspect-oriented development**.
- In generative computing, different models are used to capture the transformation process, including computation-independent models, platform-independent models, and platform-dependent models.

Generative Computing

- Involves the use of **domain-specific languages and code generators** to automate software development
- Reduces the need for manual coding and allows developers to **focus on design and functionality**
- Uses a **high-level, abstract description** of the desired software, which is then converted into actual code by a code generator
- **Can save time** and reduce the risk of errors by eliminating the need to manually write and debug code
- Often **used in domains with repetitive or boilerplate code** or where the complexity of the code makes manual coding impractical
- Can be used to **create customizable software** for different users or environments

Monolithic Software vs. Components

- Components are **clusters of software, configuration, and other elements** that form a unit for deployment and maintenance within a component framework.
- Components are made up of collaborating elements and can be **adjusted without requiring source code changes**. This is an alternative to traditional applications, which are monolithic and cannot be easily modified.

Objects vs. Components

- A regular object **combines business logic** with specific mechanisms such as persistence and hides the internal interfaces behind the external interface.
- Objects **may make assumptions about the environment**, such as which database to use, and these are hidden in the code.
- **Customizing objects requires code changes**, which can be time-consuming and risky.

Separation of Concerns and Context

- The internal interface of a component is described in **meta-information**, which allows deployers to connect the component to the appropriate framework services.
- Concerns such as persistence and transactions are separated from the business logic of the component and are typically **implemented by the framework**.
- **Context information**, such as which database to use, is contained in **meta-information** rather than in the code of the component.
- This separation of concerns and context **allows components to be customized after development**.

Overview

- Enterprise Java Beans (EJBs) allow the construction of distributed applications by combining components from different vendors.
- Developers do **not need to understand low-level** distributed mechanisms such as transactions when using EJBs.
- Can run in **EJB containers** from different vendors without modification.
- Provide enterprise **lifecycle support**, including development, deployment, and runtime support.
- Provide **enterprise data** support.

EJB Component Model

The EJB component model includes four types of EJBs: stateless session beans, stateful session beans, entity beans, and stateless message-driven beans.

- **Stateless session beans** provide stateless services that are shared among clients.
- **Stateful session beans** hold conversational state and are not shared among clients.
- **Entity beans** map to rows in a database and are shared among clients. They are used to store and access business data in a persistent manner.
- **Stateless message-driven beans** receive messages asynchronously and can connect to an enterprise MOM.

These different EJB types allow for scalability, client code on the server side, asynchronous processing, and the representation of business data.

Session Beans (Stateful and Stateless)

- Session beans are **per-client objects**, except for stateful session beans which represent client code on the server side.
- Both stateful and stateless session beans can **participate in transactions** if the session-synchronization interface is used.
- Session beans **may access databases**, but do not directly represent persistent objects.
- **Short-lived** and are removed when the container crashes.
- Stateless session beans are the most scalable and EJB servers should be able to support large numbers of them.

Entity Beans (Deprecated Since 3.0)

- Entity beans are **shared objects** that are protected through transactions.
- They have a **longer lifetime than session beans** and represent important business data.
- Entity beans **can be persisted** through the container mechanism (Container Managed Persistence, or CMP) or they can handle their own persistence (Bean Managed Persistence, or BMP).
- Entity beans have a **unique identity**, which is visible to clients through a primary key, and clients can request a “handle” which is a persistent pointer to the entity object.
- In EJB 3.0, persistence is no longer a concern for EJBs and is **instead covered by the Java Persistence API**.

Message-Driven Beans

- Message-driven beans are **invoked asynchronously** and do not have client context available during processing.
- They can be transaction aware, meaning that message receipt and processing can be enclosed in a single transaction.
- Message-driven beans are **short-lived and stateless**, and are removed when the container crashes.
- They **do not directly map** to business data.
- Message-driven beans have been integrated into the general EJB framework to **reuse EJB container services** such as transactions, security, concurrency, and deployment description.
- If a message-driven bean crashes during processing, the message is marked as “unread” and **can be processed after the container is restarted**.

Client View of EJBs

- Clients **never access the bean class** directly.
- EJBs can offer a **remote or local interface** to clients (clients of the local interface must be in the same Java virtual machine as the bean container).
- The business logic is contained in the bean class.

Local vs. Remote Interfaces

- The remote interface of an EJB uses remote object calling conventions, while the local interface uses local Java calling conventions.
- When calling the **local interface** of an EJB, value objects are **passed by reference**, meaning the client and EJB will share the same objects.
- The **remote interface**, on the other hand, requires that **value objects be copied**.
- EJB implementers who want to provide both a local and a remote interface must be aware of these different calling conventions and design their beans accordingly.

Separation of Concerns and Context in EJB

Separation of concerns is done through the EJB framework, separation of context is done through deployment:

- The **automatic transaction management** feature maps to system management transaction modes.
- **Persistence** maps to system management definitions of data sources and pool sizes.
- Automatic, method-level **security** maps to system management definitions of role/user binding.
- The roles involved in component development, application assembly, and deployment map to the deployment descriptor and JNDI interface.

1. A client invokes an **entity bean interface.**
2. The container delegates the request to the **entity bean business logic.**
3. The entity bean business logic **delegates** tasks such as transactions, persistence, and security to resources accessed through JDNI and a database.

At the point of interception, the container provides resource management, lifecycle, state management, transactions, and security services to the bean.

- Containers are increasingly **taking over roles that were previously the responsibility of operating systems**, such as isolating different applications from each other.
- In the J2EE environment, class loaders are used for this purpose.
- However, it is believed that **a better concept is needed** that does not

Containers and Threads

- Containers manage resources across applications and store context and session information in threadlocal storage.
- As a result, **container-managed applications are not allowed to create their own threads**, as these threads would not have the necessary metadata and context information.
- EJB 3.0 offers a managed service for connectors to create threads, allowing applications to use threads without having to manage resources themselves.
- Applications should not assume responsibility for resource management, as the **container is responsible for choosing the appropriate resource management policy**.

Entity Bean Container Contract

The Entity Bean-Container Contract defines a set of methods that the container can call on an entity bean in order to manage its lifecycle and access resources.

These methods include:

- setEntityContext: Bean stores context as an interface to the environment
- PrimaryKeyClass ejbCreate: Actions related to bean instance construction
- ejbPostCreate: Bean identity is now available
- ejbActivate: Bean can acquire necessary resources
- ejbPassivate: Bean releases resources, expecting to be put back into the pool
- ejbRemove: Last chance for the bean before destruction
- ejbStore: Bean should update its internal state, expecting it to be

Bean Managed vs. Container Managed Persistence

There are two main approaches to persistence in EJBs: bean-managed and container-managed.

- In **bean-managed persistence**, the bean is responsible for performing its own persistence, with the timing of these actions controlled by the container.
- In **container-managed persistence**, the bean's state is completely stored and loaded by the container.

It is generally believed that **container-managed persistence is the preferred approach in the future**, as it is more portable and does not require adjustments to different datastores. Bean-managed persistence, on the other hand, is less portable and requires more effort to adapt to different datastores.

JNDI Naming Context

- EJBs locate all their resources through JNDI (Java Naming and Directory Interface) calls, which allows deployers to **specify the proper services** for each EJB **through the deployment descriptor**.
- This allows the **deployer to manipulate all JNDI lookups** and customize the resources that each EJB has access to.

Deployment Descriptor

- The deployment descriptor is a **file that contains meta-information** about an EJB in XML format.
- This meta-information is **used by different components**, including the bean itself (to declare the names and interfaces used), the deployer (to adjust values for specific environments), and generators (to generate queries from the meta-information).

Security Support

Principal delegation:

- In this mode, the EJB container **passes along the identity** of the original caller (client) to the EJB, allowing the EJB to operate under the same identity as the client.
- This allows the EJB to access resources and perform actions **based on the permissions of the client**.

“Run as” identity:

- In this mode, the EJB container **assigns a specific identity to the EJB**, regardless of the identity of the original caller.
- This allows the EJB to operate under a specific identity that is different from the client’s identity, and allows the EJB to access resources and perform actions based on the permissions of the assigned identity.
- This can be useful in situations where the EJB needs to **perform**

Transaction Modes

EJBs support several transaction modes, which allow developers to specify the level of transaction support required by the EJB.

The available transaction modes are:

- **Not supported**: The EJB does not require or support transactions.
- **Required**: The EJB requires that a transaction be active when it is called, and will participate in the existing transaction if one is already active. If no transaction is active, a new one will be started.
- **Supports**: The EJB will participate in an existing transaction if one is active, but it does not require a transaction to be active when it is called. If no transaction is active, the EJB will execute without a transaction.
- **RequiresNew**: The EJB requires a new transaction to be started when it is called, regardless of whether a transaction is already active.
- **Mandatory**: The EJB requires that a transaction be active when it is

Best Practices for EJBs

- **Use local interfaces if possible** to reduce network overhead
- **Use transfer objects** to reduce network traffic
- Avoid excessive use of entity beans and **use local session beans** instead
- Use message driven beans for asynchronous processing
- **Use container managed transactions** and security to simplify development
- **Use container managed persistence** and avoid bean managed persistence if possible
- **Use the Java Persistence API (JPA)** for persistence management instead of entity beans in EJB 3.0 and later
- Properly design your data model and access patterns to optimize performance
- **Use performance monitoring tools** to identify and fix bottlenecks in your application

Shortcomings of EJBs

- **Large number of artifacts** for the programmer to control
- **Meta-data separated** in deployment descriptor instead of code
- Home interfaces and **finding of remote objects tedious**
- **Performance problems** in the O/R mapping due to abstract schema approach
- **No rapid prototyping** possible
- Entity beans **overloaded with security, transactions and persistence**

New Ways for Object-Relational Mapping

- The trend towards using generic, abstract schema mappings for O/R mapping appears to be over, and **SQL is becoming more prevalent**.
- Developers have historically had difficulties with the highly abstract entity beans used in Enterprise Java Beans (EJBs) for persistence. The use of **plain old Java objects (POJO)** for persistence has been seen as a more straightforward and effective solution.

Lessons Learned from EJBs

- It **takes a long time to develop** and scale a complex framework for components like EJBs.
- It is important to get the interfaces right in order to achieve good performance.
- Developers may **not fully understand the implications of abstractions**.
- Frameworks can sometimes require developers to do **unnecessary code duplication**.
- It is important to **avoid coupling too many concerns** (such as transactions, persistence, and security) in a single framework.
- **Code generation can be helpful**, but it requires tooling and customization.
- It is **best to wait for "best practice patterns"** to emerge before implementing complex new technology on a large scale.

Services

Overview

- A recap of CORBA
- Web services
- SOA
- Microservices
- Conway's Law
- Serverless computing

Timeline of Distributed Service Architectures

- In the early 1990s, distributed service architectures for use in intranets (private networks) emerged, including **Unix RPC and DCE**.
- In 1998, **CORBA** (Common Object Request Broker Architecture) was introduced as a distributed service architecture for use in intranets.
- In the early 2000s, the **REST** (Representational State Transfer) architectural style was introduced for building web services that could be accessed over the internet.
- In 2004, **SOA** (Service-Oriented Architecture) emerged as a way to design and build web services that could be easily reused and composed to create larger, more complex systems.
- In 2010, the **Microservices** design pattern became popular as a way to build large, complex applications by composing small, independent services that communicate with each other through APIs.
- In 2016, the concept of **Serverless Computing**, emerged as a way to build and run applications and services without the need to manage

CORBA Security Model

- In the CORBA security model, the concept of **secure delegation** is used to ensure that communication between systems is secure.
- When **systems** communicate with each other, they **authenticate themselves** to ensure that they are authorized to exchange information.
- Tokens are used to flow client information between systems, but **no secrets are shared**.
- **Defined routes** are used to prevent token abuse and ensure that later tiers can verify the original requestor and route of the request.

CORBA Core Properties

- CORBA was primarily designed as an **intranet technology** for use within private networks.
- CORBA is **language-independent**, with a focus on defining interfaces between systems.
- The IIOP (Internet Inter-ORB Protocol) is the **base protocol** used in CORBA to ensure interoperability and handle cross-cutting concerns.
- IIOP also provides **delivery guarantees** for communication between systems.
- CORBA was primarily used to **connect heterogeneous (legacy) software** in large corporations.
- The **standardization** process for CORBA was **difficult and tedious**.
- The use of “boilerplate code” in CORBA often led to **extensive code generation and model-driven development**.

Overview

- A Web Service is a software component that **represents a business function or service**, and can be accessed by other applications over **public networks using standard protocols** and transports (such as SOAP over HTTP).
- Web Services **use XML** to create requests and responses and send them using HTTP, allowing machines to communicate with each other for various purposes, such as supply chain management or business-to-business processing.
- **XML-RPC**, proposed by David Winer, was one of the earliest standards for Web Services.
- Web Services **have been used internally** by companies for some time.

Web Services Core Properties

- Web Services use **“simple” requests** that can be sent over public networks/the internet using HTTP transport for firewall reasons.
- XML is used as the message format for Web Services, making them **language-independent**.
- Features such as reliability, security, and transactions were added to Web Services to improve their functionality.
- Many Web Services were **re-writes of CORBA interfaces using XML syntax** and expressing a business function.
- Web Services were heavily promoted as a solution for automatic interoperability based on self-describing services and ontologies, but this was largely overhyped.
- The technical foundation for Web Services was provided by forms of **XML-RPC**, although the acronym “SOAP” (Simple Object Access Protocol) did not actually have anything to do with distributed objects.

UDDI Functionality

UDDI (Universal Description, Discovery, and Integration) is a registry that provides a “find and publish” API for distributed services. It works like this:

1. Providers **publish their services** in a registry.
2. Requesters **search for the desired service** in the UDDI (Universal Description, Discovery, and Integration) registry.
3. The UDDI registry **retrieves the provider location and WSDL** (Web Service Description Language) service description for the requester.
4. The requester **creates a request** based on the WSDL description.
5. The requester **sends the request** to the provider using a specified transport protocol (such as SOAP over HTTP).

This type of architecture is called **“service-oriented”** because it uses a broker for service advertisement and lookup, and requester and provider bind dynamically with respect to the transport protocol used.

UDDI Content and Categories

- All content in UDDI is expressed in XML.
- The UDDI registry includes information about companies and services, as well as meta-information elements such as tModel.
- A key feature of UDDI is the expectation that requester and provider will do a **dynamic bind**, agreeing on service and transport characteristics.

The UDDI registry has three main categories of information:

- **White pages**: information about companies, such as location and contact details.
- **Yellow pages**: business categorization and classification by type and industry.
- **Green pages**: meta information about services and their qualities.

WSDL Overview

- WSDL (Web Service Description Language) is the **metadata language** used by Web Services.
- WSDL defines how service providers and requesters understand Web Services.
- When exposing back-end systems as Web Services, WSDL **defines and exposes the components and lists all the data types, operations, and parameters** used by the service.
- WSDL **provides all the information that a client application needs** to construct a valid SOAP invocation, which is then mapped onto back-end enterprise logic by the Web Services platform.

WSDL Elements

- WSDL documents **define services as collections of network endpoints** or ports.
- The abstract definitions of endpoints and messages in WSDL are **separated from their concrete network deployment** or data format bindings.
- Separation of abstract and concrete definitions **allows for the reuse** of abstract definitions

It includes the following elements:

- **Types**: A container for data type definitions using a specified type system (such as XSD).
- **Message**: An abstract, typed definition of the data being communicated.
- **Operation**: an abstract description of an action supported by the service.

SOAP

- SOAP is an RPC (Remote Procedure Call) protocol that **uses XML**. It includes elements for type marshalling and RPC semantics.
- The header element in SOAP **can contain meta-information**, but it is optional.

There are several aspects that define it's performance:

- Marshaling time
- Internet transport time
- Effect of size on transport
- Demarshaling time

It has been found that internet **transport time**, especially in the absence of Quality of Service (QoS) measures, **has a greater impact on overall request time** than the size and interpretation effort of a textual format.

Security and Web Services

- **SOAP, WSDL, and UDDI**: Message Envelope, Interfaces Definition, and Registry.
- **WS-Security**: Secure Messaging Definitions.
- **WS-Trust**: How to Get Security Tokens (issuing, validation, etc.).
- **WS-Federation**: How to Make Security Interoperable Between Trust Domains.
- **WS-Policy**: How to Express Security Requirements.
- **SAML**: A Language to Express Security-Related Statements.
- **WS-Reli**: Rights Management.
- **WS-Util**: Helper Elements.
- **WS-Authorization**: Expression of Access Rights.

Reliable Messaging

Reliable B2B (Business-to-Business) messages require the following qualities:

- **Guaranteed delivery** (acknowledgement enforced)
- **Duplicate removal** (using message ID)
- **Message ordering** (using sequence numbers)

SOAP and HTTP partially achieve this like so:

1. The first application layer exchanges persistent messages with the requester.
2. The requester sends a SOAP message with a message ID, sequence number, and QoS (Quality of Service) tag to the receiver.
3. The receiver must send an acknowledgement.
4. The receiver exchanges persistent messages with the second application layer.

Secure Messages

- Digital **signatures** with XMLDsig
- Digital **encryption** with XMLEnc
- WS-Security moves from channel-based security to message **(object)-based security**, allowing individual messages to be signed and encrypted.
- WSDL can **advertise the QoS** expected/provided by a receiver.
- **End-to-end security** is possible across intermediaries.
- Today, **federation** (expressed in the WS-Federation standard) is more important, as seen in the use of OAuth2.
- **Intermediaries can add signatures or encryption** to the SOAP envelope to create a chain of trust.
- New signatures or encryption information is **always prepended** to existing information.
- No encryption of the envelope, header, or body tag is allowed.
- Signatures must respect the right of **intermediaries to change the**

SAML

- SAML allows **externalization of policies and mechanisms** related to authentication, authorization, and attribute assertion.
- The access control point **only needs to check assertions** and does not have to implement these mechanisms.
- SAML allows **interchangeability of statements between different services** because the format of the assertions is fixed.

Transaction Models

Atomic transactions:

- Are not nested (**standalone**)
- Are **short**
- Involve a **tightly coupled** business task
- Can be **rolled back** in case of error
- Can be disrupted by system crashes

Activity transactions:

- Involve **nested tasks**
- Are **long-running**
- Involve a **loosely coupled** business activity
- Include **compensating tasks and activities** to address errors
- Can be disrupted by errors such as order cancellations

Stateful Web Services

- Stateful architectures, such as computational grids, require the concept of a **resource**.
- WS-Resource is a protocol that adds resource information to web services through **metadata descriptions** in the WSDL and WS-Addressing schemas.
- An **identifier** is used to communicate state information between requestors and endpoints.
- Advanced **notification requests** can be built on top of WS-Resource.

Scaling Web Services

- **Avoid** using XML messaging for **fine-grained RPC**, such as requesting the square root of a number or a stock quote.
- **Use course-grained RPC** instead, with web services that "do a lot of work, and return a lot of information".
- **Consider an asynchronous messaging model** when the transport may be slow or unreliable, or the processing is complex or long-running.
- Take the overall system performance into account and **don't assume** that XML's "bloat" or HTTP's **limitations are a problem until they are demonstrated in your application**.
- **Consider the frequency of messaging** when designing your web service. A high rate of requests may suggest that you load (replicate) some data and processing back to the client.
- For web services that aggregate data from other web services, **consider performing the aggregation on demand** or during off-hours in one large, course-grained transaction.

Why UDDI Failed

UDDI relied on the following assumptions:

- **Central registries** of service descriptions
- **Independent automatic agents** searching for services
- Machines **understanding** service descriptions
- Machines **deciding** on service use
- Machines being able to use a service properly
- Machines being able to **construct advanced workflows** from different services

These assumptions were problematic because:

- There is often **ambiguity**, undefined aspects, and incompatible terms in service descriptions and interfaces.
- It is difficult for machines to properly use and construct advanced workflows from different services.

Lessons Learned from Web Services and CORBA

- Web services and CORBA are low-level concepts that **lack semantics**.
- Workflow has not been effectively addressed by web services and CORBA.
- Service-Oriented Architecture (SOA) is not only about interfaces and interface design, but **also about hosting services**.
- The **availability of a service on the web** is more valuable than many specifications and interfaces.

SOA Core Properties

- Services offer **high-level interfaces** that relate to business functions.
- Service choreography (i.e. the coordination of services to achieve a larger business process) is performed outside the individual services.
- **Semantic standards and technologies** (such as OWL, SAML, and the Semantic Web) are used to enable agents to understand services and their interfaces.
- Legacy applications can be made available to other companies through the use of a service interface.
- SOA **relies on Web Service technology** as its foundation.

SOA Interface Design

- **Object interfaces** can be conversational, accept transactions, and are fast. They use object references.
- **Component interfaces** use value objects, have a transaction border, and are generally stateless. They are relatively fast.
- **Service interfaces** are used for long-running transactions with state stored in a database. They include compensation functions and have short process times but long business task execution times. Service interfaces are isolated, independent, and can be composed with larger services (choreography) or made up of smaller services (orchestration). They are stateless.

SOA Blueprint Service Types

- **Component Services**: These are atomic operations on simple objects, such as database access.
- **Composite Services**: These are atomic operations that use multiple simple services (orchestration) and are stateless for the caller.
- **Workflow Services**: These are stateful services with defined state changes, with the state kept in a persistent store.
- **Data Services**: These provide information integration through a message-based request and response mechanism.
- **Pub/Sub Services**: These are event-based services with callbacks and registration.
- **Service Broker**: These are intermediate services that manipulate and forward messages based on rules.
- **Compensation Services**: These revert actions, but do not roll back like traditional transactions.

SOA vs. Microservices

SOA:

- Services are usually **larger** and more transactional.
- **Prefer orchestration** using higher level middleware components.
- **Enterprise-oriented architecture** with middleware (messaging), service layers, and ownership concepts.
- Uses metadata and messaging middleware for **contract decoupling**.
- Follows a **“share as much as possible”** approach.
- Big services **are transactional**.

Microservices:

- **Smaller and more specialized** than SOA services.
- Use **eventual consistency technologies**, which are not ACID.
- **Prefer choreography**, which can lead to highly connected and dependent systems.
- Have a simple **API gateway** and teams own their infrastructure and

RPC vs. REST

- The web is based on **representing resources using URIs**, while web services create private, non-standard ways of accessing information.
- The envelope paradigm used in **RPC does not offer any benefits over the generic HTTP methods** (GET, PUT, POST).
- **RPC** mechanisms are **not suitable for the web**. Some extensions to the HTTP methods might be necessary to support certain features, such as tuple-space systems.

The Web's Architecture

- Follows a **client-server model**, where clients request resources from servers.
- Has a **uniform interface**, with resources identified using URIs and manipulated through representations.
- **Messages are self-descriptive**, with metadata, headers, and other information that allows them to be understood and processed by clients and servers.
- Uses hypermedia as the engine of application state (**HATEOAS**), meaning that responses to requests include actionable links that allow clients to navigate the system and access related resources.
- Is a **layered system**, with intermediaries such as caching servers, security systems, and load balancers playing important roles.
- Is **designed to be cacheable**, with responses indicating whether they can be cached and for how long.
- Is **stateless**, meaning that clients need to provide context and state

REST Maturity Model

- **REST Level 0 (RPC)**: This level resembles regular RPC, with function calls being made to a single endpoint. Resources are not accessible and do not have an identity.
- **REST Level 1 (Resources)**: Function names and parameters are turned into resources, and appointments now have an identity that can be accessed through GET and POST requests.
- **REST Level 2 (HTTP verbs)**: It is important to use the correct HTTP verb (GET, POST, etc.) to indicate the intended action. GET is idempotent and creates cachable resources. Response codes are used to indicate the status of the request, such as the creation of a new resource or a conflict.
- **REST Level 3 (HATEOAS)**: Responses include encoded actions that can be invoked by the client. Services can change their URIs without breaking clients, and the “rel” attribute is used to describe the semantics behind the URI link.

REST Resource Archetypes

- **Document**: Fields and links representing a base resource. Created using POST.
- **Collections**: Containers maintained by the server with URI generation. Created using POST.
- **Stores**: Container elements maintained by the client with “put” and without URI generation on the server side. Inserted or updated using PUT.
- **Controllers**: Procedures accessed using POST.
- **URI path design**: Reflects the resource model, with variable path segments and query terms.

CRUD with REST

In RESTful web services, requests are made by a **requestor to a representation of a resource**. The HTTP methods (GET, POST, PUT, DELETE) are used to perform different actions on the resource:

- **GET**: Reads the resource and does not change the server state (idempotent).
- **POST**: Creates a new resource on the server.
- **PUT**: Updates an existing resource on the server.
- **DELETE**: Deletes a resource on the server.

The separation of updates and reads is a principle of good software design that has been around for a long time. It is known as the "command-query separation principle" and was made a requirement in the Eiffel programming language.

RESTful Web Features

RESTful web services have four key characteristics:

- Use the HTTP protocol in a **CRUD-like** manner, with HTTP methods corresponding to different actions on resources.
- Are **stateless**, meaning that the client and server do not maintain a persistent connection or state information between requests.
- **Use meaningful URIs** to represent objects and their relationships in the form of directory entries, with relationships typically being parent/child or general/specific entity relations.
- **Use XML or JSON** as a transfer format and content negotiation with mime types.

Critical Points with REST

- **Delivery of requests**: How to handle at-least-once or at-most-once delivery and transactions.
- **Security**: How to secure requests and delegate security to backend systems (e.g. bearer tokens).
- **Performance**: How to optimize performance over HTTP, especially with large amounts of data or many round-trips.
- **Single responsibility**: How to avoid the service becoming heavy, kludgy, and serve more than a single responsibility over time.

GraphQL as a REST Alternative

- **Avoids over- and under-fetching** of data by allowing the client to specify exactly what data it needs in a single request.
- **Reduces the number of requests** needed by allowing the client to request all the necessary data in a single request.
- **Uses a single endpoint** with resolvers to handle all requests.
- **Uses a uniform syntax** for both data and queries, making it easy to use and understand.
- **Is typed** to prevent mistakes and ensure that the correct data is returned.
- **Supports federated servers**, allowing multiple servers to be combined and accessed through a single endpoint.
- **Has the potential for huge queries**, which can be a danger if not carefully managed.

The Reasons for Microservice Adoption

- Ultra large-scale sites require **efficient horizontal scaling**.
- Unicorn companies (successful startups) **need to develop new features quickly** with independent teams.
- Unicorn companies need to **deploy new features quickly** due to competition and the need for experimentation.
- Unicorn companies **need to offer an API for network effects**.

Scalability Problems of Monolithic Applications

- Monolithic applications **can only be deployed as a whole**, making it difficult to scale specific components.
- The **central database** of a monolithic application can be hard to scale.
- The **API** of a monolithic application can be **hard to scale**.
- Developers are **dependent on the general release plan** for the entire application, which can be inflexible and slow.

Scalability Benefits of Microservices

- Individual functions can be **deployed independently**, making it easy to perform A/B testing.
- **Independent teams and releases are possible**, allowing for more flexibility and faster development.
- **Databases are often independent** due to sharding, making it easy to scale them.
- The **API** can be **quickly scaled** to meet demand.

The Microservice Ecosystem

- **Continuous integration/deployment** allows for rapid experimentation.
- **Fully automated build and deploy processes** ensure efficient development and deployment.
- A **variety of programming languages** can be used.
- **Continuous monitoring** tools, such as ELK, help identify and troubleshoot issues.
- **Both REST APIs and RPC tools** can be used to communicate between microservices.
- **Containers** are preferred over virtual machines (VMs) because they are more lightweight and efficient.
- **DevOps** teams are responsible for operations.
- **Site-reliability engineers (SREs)** oversee the performance and reliability of the system.
- **Distributed transactions** are avoided.

Microservice Design Patterns

- **API gateway**: Facade to fine-granular services
- **Client-side discovery**: Provided by MS chassis (e.g. spring boot)
- **Server-side discovery**: Services register themselves during startup, or self registration by chassis
- **Service instance per container**: Scales better than VM per service
- **Serverless deployment**: Use of functions as a service (FaaS) to run small pieces of code in response to specific events
- **Database per service**: Each microservice has its own database, with no direct access to the databases of other microservices
- **Event-driven architecture**: Programming without a stack, with changes triggered by events
- **Event sourcing**: Record change events in an event store
- **CQRS**: Separate update and idempotent read operations
- **Transaction log tailing**: Follow transaction log for changes
- **Database triggers**: Put events in events table after changes

Critical Points with Microservices

- **Cross-concerns**, such as transactions, security, performance, and scalability, **can be challenging** to manage in a microservice system. Site-reliability engineers (SREs) may be responsible for addressing these issues.
- Virtual machines (VMs) may be too large for small services with low throughput.
- The **maintenance** of many microservices **can be time-consuming and complex**.
- **Monitoring can be challenging** due to the large number of independent services and the difficulty in identifying correlations between them.
- **Different languages and technologies** may cause confusion and fragmentation
- There is a **risk of creating new central bottlenecks**, such as an event store.

Serverless Definition

- Cloud-native platform for **short-running, stateless computation and event-driven** applications
- **Scales** up and down instantly and automatically
- **Charges for actual usage** at a millisecond granularity
- **Decouples computation and storage**, scales and priced separately
- **Pays for code execution** instead of allocated resources
- Allows developers to **focus on writing code**, not infrastructure management

Stateless Applications

- Stateless system or component **does not maintain state or context between requests**
- Each request is treated as an **independent action**, not influenced by previous requests
- Configuration information may be stored in classpath or persistent storage
- Function has **no memory**
- **Change management may be a challenge** in stateless systems
- **Hypercomposition** (breaking a system down into smaller, independent components) **may be difficult** in stateless systems

Issues with Serverless

- Countless small **IAM rules**
- Coupling with less scaleable components
- Slow **cold starts**
- Unclear **bug handling**
- Stateful functions
- **Limits** everywhere
- New testing concepts needed
- **High costs when waiting** for something
- **Unpredictable latency** due to the dynamic nature of the environment
- **Lack of direct addressability** of functions
- **Limited support for common software patterns**, such as batch processing
- **Difficulty debugging**, tracing, and monitoring functions
- **Inefficient storage systems** for small objects or frequent access

- **Scalable Webhook**: Triggers based on external events
- **Gatekeeper**: Controls access to internal resources
- **Internal API**: Provides an interface to internal resources
- **Internal Handoff**: Communicates between serverless functions
- **Aggregator**: Gathers and processes data from multiple sources
- **Notifier**: Sends notifications to external systems or users
- **FIFOer**: Ensures first-in, first-out processing of events
- **Streamer**: Processes data from a stream in real-time
- **Router**: Routes events to the appropriate destination
- **State Machine**: Executes a series of steps based on the current state of an event

Theoretical Foundations of Distributed Systems

Overview

- Basic Concepts
  - Distributed Systems Fallacies
  - Latency
  - Correctness and Liveness
  - Time, Ordering and Failures
  - The Impossibility of Consensus in Async. DS (FLP)
  - The CAP Theorem
- Failures, Failure Types and Failure Detectors
- Time in Distributed Systems
- Ordering and Causality
- Algorithms for Consensus in DS
- Optimistic Replication and Eventual Consistency

The Eight Fallacies of Distributed Computing

- **The network is reliable**: This fallacy assumes that the network will always be available and free of errors or failures, which is not always the case.
- **Latency is zero**: This fallacy assumes that communication between nodes in a distributed system will be instantaneous, but in reality, there is always some latency due to the time it takes for a request to be processed and a response to be received.
- **Bandwidth is infinite**: This fallacy assumes that there is unlimited bandwidth available for communication between nodes, but in reality, bandwidth can be limited by factors such as network congestion or hardware limitations.
- **The network is secure**: This fallacy assumes that the network is completely secure and free from threats such as hackers or malicious software, but this is not always the case.
- **Topology doesn't change**: This fallacy assumes that the topology of

Analyzing Latency

- **Know the long-term trends in hardware**: Latency can be influenced by the performance of the hardware being used, so it's important to be aware of trends in hardware development and how they may impact latency.
- **Understand the problem of deep queuing networks and the solutions**: Deep queuing occurs when there are many requests waiting to be processed, leading to longer latency. Understanding this problem and implementing solutions such as load balancing can help reduce latency.
- **Know your numbers with respect to switching times, router delays, round-trip times, IOPS per device, and perform "back of the envelope" calculations**: It's important to have a good understanding of the specific numbers and metrics related to latency in your system, such as switching times and router delays, and to perform calculations to estimate the impact of these factors on latency.

Characteristics of Distributed Systems

- **High complexity**: Distributed systems involve a large number of interacting agents, such as servers, clients, and network devices, which can make them complex to design, build, and maintain.
- **Partial knowledge**: In distributed systems, each node or agent typically has only partial knowledge about the state of the system, the actions of other nodes, and the current time. This can make it difficult to coordinate actions and ensure consistency across the system.
- **Uncertainty**: Distributed systems are prone to uncertainty due to factors such as node failures, network delays, and changes in the system's environment. This uncertainty can make it challenging to predict the behavior of the system and ensure its reliability.

Liveness vs. Correctness

Correctness and liveness are two important properties of distributed systems that ensure they function as intended and make progress.

- **Correctness** refers to the property that ensures that a system will not exhibit undesirable behaviors, such as incorrect results or incorrect behavior. It can be thought of as the **absence of bad things** happening in the system.
- **Liveness**, on the other hand, refers to the property that ensures that a system will eventually make progress and achieve its intended goals. It can be thought of as the **presence of good things** happening in the system.

Both correctness and liveness are **based on assumptions** about failures and other conditions in the system, such as fairness and the presence of Byzantine errors. Ensuring that a distributed system has both correctness and liveness is critical for its success.

Liveness and Correctness in Practice

Here is an example of how correctness and liveness can be defined for an event-based system:

Correctness:

- **Receive notifications only if subscribed to them**: This ensures that a node only receives notifications for events it is interested in.
- **Received notifications must have been published before**: This ensures that notifications are not received before they have been published, which would lead to incorrect behavior.
- **Receive a notification only at most once**: This ensures that a node does not receive the same notification multiple times, which could lead to incorrect behavior.

Liveness:

- **Start receiving notifications some time after a subscription was**

Timing Models

In distributed systems, timing models refer to the way that events and communication between nodes are synchronized. There are three main types of timing models: synchronous, asynchronous, and partial synchronous.

- **Synchronous timing models**: In synchronous timing models, transmit times are strictly defined and events happen at specific moments. Nodes in a synchronous system can immediately detect when another node has crashed because the system relies on a clock to synchronize events. Examples of synchronous systems include CPUs and other types of hardware.
- **Asynchronous timing models**: In asynchronous timing models, there is no exact time between sending and receiving messages. Messages will "eventually" arrive, but there is no guarantee about when. Because there are no strict timing constraints, a node in an asynchronous system cannot tell whether another node has crashed

The Fischer, Lynch and Patterson Result

- The FLP (Fischer, Lynch, and Patterson) result is a theoretical result that shows it is **impossible to reach consensus in asynchronous distributed systems** in certain circumstances.
- The result has **significant implications** for distributed algorithms that rely on consensus, such as leader election, agreement, replication, locking, and atomic broadcast.
- The problem is **caused by the need for a unique leader** to make a decision, but the asynchronous nature of the system can lead to delays and the re-election of new leaders, which can **indefinitely delay** the decision-making process.
- This problem affects most consensus-based distributed algorithms and can result in non-terminating runs where no decision is reached.

Overview

States that in the presence of network partitions, a client must choose either consistency or availability, but not both.

- **Choosing consistency**: If the client chooses consistency, they may not get an answer at all.
- **Choosing availability**: If the client chooses availability, they may receive a potentially incorrect answer.

Preconditions for the CAP Theorem

- To be considered consistent, the system must ensure that **all operations are atomic and linearizable**, meaning that they can be thought of as occurring at a single instant in time and have a total order.
- For a system to be considered available, it must **ensure that every request received** by a non-failing node **is met with a response**, and that every request terminates.
- Partition tolerance: The network will be **allowed to lose arbitrarily many messages** sent from one node to another.

Common Misconceptions of the CAP Theorem

- **Consistency**: Many systems do not achieve a total order of requests due to the costs (latency, partial results) involved.
- **Availability**: Even an isolated node with a working quorum on the other side must answer requests, breaking consistency. The node does not know that a quorum exists.
- **Partition Tolerance**: You cannot un-choose partition tolerance, as it is always present. CA systems are therefore not possible.

The Modern View of the CAP Theorem

- There are **more failure types than just partition tolerance**, such as host-crash and client-server disconnect. These failures cannot be completely avoided.
- Many systems **do not need linearizability**, and it is important to carefully consider the type of consistency that is needed.
- Most systems **prioritize latency over consistency**, with availability coming in second.
- A **fully consistent system** in an asynchronous network **is impossible** (in the sense of FLP). FLP is much stronger than CAP.
- The **architecture of the system** (replication, sharding) and the **abilities of the client** (failover) also have an impact on the system's behavior.

PACELC

PACELC: A **more complete portrayal** of the space of potential consistency tradeoffs for distributed database systems.

- In the presence of a partition (P), the system must trade off availability and consistency (A and C).
- In the absence of a partition (E), the system can trade off latency (L) and consistency (C) when running normally.

Technical Failures

- **Network failures**, such as partitioning, which can occur when a network is divided into smaller, separate networks that are unable to communicate with each other.
- **CPU/Hardware failures**, such as instruction failures or RAM failures, which can occur when the hardware components of a system malfunction or fail.
- **Operating system failures**, such as crashes or reduced function, which can occur when the operating system experiences an error or malfunction.
- **Application failures**, such as crashes, stopped states, or partially functioning states, which can occur when an application experiences an error or malfunction.

Unfortunately, in most cases there is no failure detection service that can identify when these failures occur and allow others to take appropriate action. However, it is possible to develop a **failure detection service** that

Failure Types

- Bohr-Bug:
  - **Shows up consistently** and can be reproduced. Easy to recognize and fix.
- Heisenbug:
  - **Shows up intermittently**, depending on the order of execution.
  - High degree of **non-determinism** and context dependency.
  - Due to complex IT environments, they are both more frequent and **harder to solve**.
  - They are **only symptoms** of a deeper problem.
  - Changes to software may make them disappear temporarily, but more changes can cause them to reappear.
  - Example: Deadlock “solving” through delays instead of resource order management.

Failure Models

- **Crash-stop**: A process crashes atomically and stays down.
- **Crash-stop with recovery**: A process crashes and is down until it begins recovery, and is up again until the next crash occurs. For consensus, at least $2f+1$ machines are needed (quorum).
- **Crash-amnesia**: A process crashes and restarts without recollection of previous events or data.
- **Failstop**: A machine fails completely, and the failure is reported to other machines reliably.
- **Omission errors**: Processes fail to send or receive messages even though they are alive.
- **Byzantine errors**: Machines or parts of machines, networks, or applications fail in unpredictable ways and may recover partially. For consensus, at least $3f+1$ machines are needed.

Many protocols for achieving consistency and availability **make assumptions about failure models**. For example, transaction protocols

Failures and Timeouts

- **Timeouts are not a reliable way to detect failures** in distributed systems because they can be caused by various factors, such as short interruptions on the network, overload conditions, and routing changes.
- Timeouts **cannot distinguish between different types** and locations of failures.
- Timeouts **cannot be used in** protocols that require **failstop behavior** of its participants.
- Most distributed systems only offer timeouts for applications to notice problems, so they do not provide detailed information about the state of participants or membership.
- Using timeouts **can result in "split-brain" conditions**, where a system behaves as if it is functioning properly but is actually experiencing a failure or malfunction.

Failure Detectors

A failure detector (FD) is a mechanism used in distributed systems to detect failures of processes or machines. It is not required to be correct all the time, but it should provide the following **quality of service (QoS)**:

- **Safety**: The FD should be safe all the time, meaning it should not falsely suspect processes of being faulty during "better" failure periods.
- **Liveness**: The FD should be live during "better" failure periods, meaning no process should block forever waiting for a message from a dead coordinator.
- **Accuracy**: Eventually, some process x should not be falsely suspected of being faulty. When x becomes the coordinator, every process should receive x's estimate and make a decision based on it.
- **Low overhead**: The FD should not cause a lot of overhead, meaning it should not consume too many resources or slow down the system.

Time in Distributed Systems

In distributed systems, there is **no global time** that is shared across all processes. Instead, different approaches are used to model time in these systems. These approaches include:

- **Event clock time**: This is a logical model of time that represents the order of events within a single process.
- **Vector clock time**: This is a logical model of time that represents the order of events between multiple processes.
- **TrueTime**: This is a physical model of time that represents the interval of time between events.
- **Augmented time**: This is a combination of physical and logical models of time that takes into account both the interval of time between events and the order of events.

Logical time is modeled as partially ordered events within a process or between processes. It is used to **represent the relative order of events** in

Consistent vs. Inconsistent Cuts

- **Consistent cuts**: These cuts produce causally possible events, meaning that events occur in a **logical and possible order**.
- **Inconsistent cuts**: These cuts produce events that arrive before they have been sent, resulting in an **illogical or impossible order**.

Event Clocks (Logical Clocks)

- Event clocks, also known as logical clocks, are **systems for ordering events** within processes according to a chosen causal model and granularity.
- Events are partially **ordered based on the order in which they occur**. The time between events is a logical unit of time that has no physical extension.
- Events delivered through messages can be used to relate the processes and their times to the events. The external order of these events is also a partial order of events between processes (for example, the event “send(p1,m)” occurs before the event “recv(p2,m)”).
- The value of the logical clock is **updated to the maximum of the current value plus one or the received value**.

Lamport Logical Clock

- The Lamport logical clock **counts events and creates an ordering relation between them**. These counters can be used as timestamps on events.
- The ordering relation captures all causally related events, but it also includes many unrelated (concurrent) events, which **can create false dependencies**.

Vector Clocks

- Vector clocks are **transmitted with messages** and compared at the receiving end.
- If, for all positions in two vector clocks A and B, the values in A are larger than or the same as the values from B, we say that **Vector Clock A dominates B**.
- This **can be interpreted** as potential causality to detect conflicts, as missed messages to order propagation, etc.

Physical Interval Time (TrueTime)

- TrueTime works by **using time servers** to check for rogue clocks, which are clocks that are not synchronized with the correct time.
- The error in TrueTime is **typically in the range of 6 milliseconds**.

Hybrid Clocks

Hybrid clocks are systems that **combine elements of both logical and physical models** of time in order to address the limitations of each approach. There are several reasons why hybrid clocks may be used in distributed systems:

- In large distributed systems, such as those spanning multiple data centers across the world, **vector clocks can become too large** to maintain efficiently. Hybrid clocks can be used to reduce the size of the clocks while still maintaining an accurate ordering of events.
- Physical interval time, such as **TrueTime**, requires that reads and writes **wait until the interval time is over on all machines**. This can be inefficient in some cases, and hybrid clocks can be used to allow for more flexibility in terms of when reads and writes can occur.

Ordering in Distributed Event Systems

- **FIFO (first-in, first-out) ordering**: This refers to the requirement that a component must receive notifications in the order in which they were published by the publisher.
- **Causal ordering**: This refers to the requirement that events must be ordered based on their causal relationships, as defined by the system.
- **Total ordering**: This refers to the requirement that events must be ordered in a specific way, such that no other component in the system is allowed to receive an event before a preceding event has been received. One component may be responsible for deciding the global order of events in this case.

Overview

Consensus is a **process used by a group** of processes to **reach agreement on a specific value**, based on their individual inputs. The objective of consensus is for all processes to decide on a value v that is present in the input set.

- **Termination**: Every correct process eventually decides on some value.
- **Validity**: If a process decides on a value v, then v was proposed by some process.
- **Integrity**: No process decides on a value more than once.
- **Agreement**: No two correct processes decide on different values.

Algorithms for Consensus

These protocols offer **trade-offs** in terms of correctness, liveness (availability and progress), and performance:

- **Two-Phase Commit (2PC)**: This algorithm is used to ensure that a group of processes all commit to the same decision. It involves two phases: a prepare phase, in which the processes prepare to commit to a decision, and a commit phase, in which they actually commit to the decision.
- **Static Membership Quorum**: This algorithm is based on the concept of quorum, which refers to a minimum number of processes that must be present in order to reach a decision. The static membership quorum algorithm is used to achieve consensus in systems with a fixed number of processes.
- **Paxos**: This algorithm is used to achieve consensus in distributed systems with a dynamic membership. It involves multiple rounds of voting in order to reach a decision.

Two-Phase Commit (2PC)

Steps:

1. **Preparation phase**: In this phase, the processes prepare to commit to a decision. Each process sends a “prepare to commit” message to a coordinator process, which is responsible for coordinating the decision-making process.
2. **Decision phase**: Once all the processes have prepared to commit, the coordinator process sends a “commit” message to all the processes. This message instructs the processes to commit to the decision.
3. **Confirmation phase**: Each process sends a “commit confirmation” message to the coordinator process, indicating that it has successfully committed to the decision.
4. **Finalization phase**: Once all the processes have confirmed that they have committed to the decision, the coordinator process sends a “commit complete” message to all the processes, indicating that the decision has been successfully made.

Quorum-Based Consensus

Steps:

1. A **decision is proposed** by one of the processes in the distributed system.
2. **Each process** in the system **votes** on the proposed decision.
3. The **votes are counted and checked** against the quorum requirement. The quorum requirement is the minimum number of votes that must be received in favor of the decision in order for it to be approved.
4. If the **quorum requirement has been met**, the decision is considered to have been approved.
5. The processes move forward with **implementing the decision**.

Example:

1. A group of five processes in a distributed system (A, B, C, D, and E) are **deciding whether to commit** to a new software update.
2. **Process A proposes the decision** to update the software.

Paxos

Steps:

1. **Prepare** (Phase 1a):

- The **Proposer** (also known as the leader) selects a proposal number N and **sends a Prepare message to** a quorum of **Acceptors**.

2. **Promise** (Phase 1b):

- If the proposal number N is larger than any previous proposal, the **Acceptors promise not to accept proposals less than N** and **send the value they last accepted for this instance to the Proposer**.
- If the proposal number is not larger, a **denial** is sent.

3. **Accept!** (Phase 2a):

- If the **Proposer** receives responses from a quorum of Acceptors, it **may choose a value to be agreed upon**.

Raft

RAFT is a distributed consensus protocol that allows a group of processes (called “replicas”) to agree on a value (“decide”) in the presence of failures. RAFT is divided into three distinct roles: **Leader, Follower, and Candidate**.

The protocol consists of the following **steps**:

1. **Leader Election**:

- When a replica starts up or its leader fails, it becomes a Candidate and **initiates an election** by sending RequestVote messages to all other replicas.
- If a Follower receives a RequestVote message from a Candidate with a higher term, it responds with its vote and updates its term to match the Candidate’s term.
- **If a Candidate receives a quorum of votes** (more than half of the replicas), **it becomes the Leader** and sends AppendEntries messages to all other replicas to replicate its log.

Atomic Broadcast Conditions

A distributed algorithm that **guarantees correct transmission** of a message from a primary process to all other processes in a network or broadcast domain, including the primary.

It satisfies the following conditions:

- **Validity**: If a correct process broadcasts a message, then all correct processes will eventually deliver it
- **Uniform Agreement**: If a process delivers a message, then all correct processes eventually deliver that message
- **Uniform Integrity**: For any message m, every process delivers m at most once, and only if m was previously broadcast by the sender of m
- **Uniform Total Order**: If processes p and q both deliver messages m and m0, then p delivers m before m0 if and only if q delivers m before m0

Atomic Broadcast Protocol

Data:

- **Epoch (e)**: The duration of a specific leadership
- **View (v)**: Defined membership set that lasts until an existing member leaves or comes back
- **Transaction counter (tc)**: Counts rounds of execution, such as updates to replicas

Phases:

1. **Leader election/discovery**: Members **decide on a new leader** and form a consistent view of the group.
2. **Synchronization/recovery**: Leader gathers outstanding, uncommitted requests recorded at members and **updates members missing certain data until all share the same state**.
3. **Working**: Leader **proposes new transactions** to the group, collects confirmations, and sends out commits.

Gossip Protocols

Gossip protocols are a class of distributed algorithms that **rely on randomly chosen pairs of nodes** in a network to exchange information about the state of the system. They are typically used for group membership, failure detection, and dissemination of information.

There are several key characteristics of gossip protocols:

- **Randomized**: Gossip protocols rely on randomly chosen pairs of nodes to exchange information, which helps to reduce the risk of overloading any particular node.
- **Scalability**: Gossip protocols scale well in large, distributed systems because they only require communication with a few nodes at a time.
- **Fault tolerance**: Gossip protocols are designed to tolerate failures and can continue to operate even if some nodes go down.
- **Asynchronous**: Gossip protocols do not rely on a central authority or global clock, so they can operate asynchronously in a distributed

DWAL

A DWAL (Distributed Write-Ahead-Log) is a data structure that is used to ensure that updates to a distributed system are stored in a way that allows them to be recovered in case of system failure. It is a type of write-ahead log, which means that updates are written to the log before they are applied to the system's state. This allows the updates to be replayed in the correct order after a system failure.

Design Components of DWALs

- **Global visibility**: Replicated **state should be visible to all processes in the system**. This can be achieved through the use of atomic broadcast or other consensus protocols to ensure that all processes have a consistent view of the system state.
- **Consensus protocol**: A consensus protocol such as Paxos or Raft is used to ensure that all processes agree on the order of updates to the replicated state. This ensures that all processes have a consistent view of the system state and reduces the risk of conflicts or data loss.
- **Majority decisions**: In a consensus protocol-based system, majority decisions are used to ensure that the system can make progress even in the presence of failures. This means that a majority of processes must agree on the order of updates to the replicated state before they can be applied.
- **Group membership**: In order to ensure that the DWAL can function

Properties of Replication Models

- **Who is responsible for updates**: Single master or multiple masters?
- **What is being updated**: State transfer or operation transfer?
- How updates are **ordered**
- **Conflict detection** and resolution
- **Method for updating** replica nodes
- **Guarantees for divergence**

Single-Leader Replication

Steps:

1. Client sends x=5 to Node1 (master)
2. Master updates node 2 and node 3 (followers)
3. Client receives changed value (or old value; due to replication lag)

Advantages:

- **Ordered** updates
- **Efficient** caching
- Highly **available reads**

Disadvantages:

- **Replicas may be out of sync** with the master
- **Leader crash** may cause problems
- **Followers may take a while to take over** in case of leader failure
- **Not suitable for critical resources** such as primary keys

Eventually Consistent Reads

Eventual consistency model: **Allows for a certain level of lag** between updates to be propagated to all replicas

Steps:

1. Client updates value on Master-Replica node
2. Master-Replica eventually propagates update to Slave replica
3. Client performs a stale read from client node, potentially returning outdated value

Multi-Master Replication

Multi-Master Replication (MMR) is a type of replication in which **multiple servers can accept write requests**, allowing any server to act as a master. This means that updates can be made to any server, and the changes will be replicated to all other servers in the network. MMR can be used to **improve the availability and scalability** of asystem, as it allows updates to be made to any server and allows multiple servers to handle write requests.

It also introduces the **possibility of conflicts**, as multiple servers may receive updates to the same data simultaneously. To resolve these conflicts, MMR systems typically use conflict resolution algorithms (last writer wins, keeping different versions, anti-entropy background merge/resolve) or allow the user to manually resolve conflicts.

Steps:

1. Client 1 writes x=5 to Node 1 (master)

Leaderless Quorum Replication

Write:

- In a leader-less (quorum) replication system, the **client decides how many machines to write to or read from** using the formula W+R>N, where N is the number of machines in the replication group.
- Without a designated leader, quorum systems may suffer from **long tail effects**.
- If a quorum is not available, the **client can choose to write to a "sloppy quorum"** and risk the write being lost.
- Without anti-entropy, there is a **high risk of partial writes** in the system, which can lead to inconsistencies and can be difficult to clean up.

Read:

- Some **systems may detect inconsistencies** during a read operation.
- These systems can **either automatically perform a cleanup** (e.g. using

Session Modes of Asynchronous Replication

The following guarantees seem to enable **“sequential consistency”** for a specific client, meaning that the program order of updates from this client is respected by the system. Clients can track these guarantees using vector clocks:

- **“Read your writes”** (RYW) ensures that the contents read from a replica include previous writes by the same user.
- **“Monotonic reads”** (MR) ensures that successive reads by the same user return increasingly up-to-date contents.
- **“Writes follow reads”** (WFR) ensures that a write operation is accepted only after writes observed by previous reads by the same user are incorporated in the same replica.
- **“Monotonic writes”** (MW) ensures that a write operation is accepted only after all write operations made by the same user are incorporated in the same replica.

Global Modes of Replication

There are multiple different modes to choose from:

- **Strong consistency** ensures that **all previous writes are visible**, and is characterized by the following properties:
  - Ordered: Writes are accepted in the order that they were made.
  - Real: All writes are visible.
  - Monotonic: Write operations are accepted only after all previous write operations made by the same user are incorporated in the same replica.
  - Complete: All writes are included.
- **Consistent prefix** ensures that an **ordered sequence of writes is visible,** and is characterized by the following properties:
  - Ordered: Writes are accepted in the order that they were made.
  - Real: All writes are visible.
  - X latest missing: Some of the latest writes may be missing.
  - Snapshot isolation-like: The system behaves like snapshot isolation, where writes made by a transaction are not visible to other

Distributed Services and Algorithms I

Overview

What is a Distributed Service?

Function provided by a **distributed middleware** with:

- High scalability
- High availabilit

Services and Instances

- Distributed systems:
  - Comprised of **services**, such as applications, databases, caches, etc.
  - **Services are** made up of instances or nodes, which are **individually addressable hosts** (physical or virtual)
- Key observation:
  - Unit of **interaction is at the service level, not the instance level**
  - Concerned with **logical groups of nodes, not specific instances**
  - Example: Interacting with a database server, rather than a specific database instance.

Core Distributed Services

- **Finding** Things
  - Name Service
  - Registry
  - Search
- **Storing** Things
  - Various databases
  - Data grids
  - Block storage, etc.
- **Events** Handling and asynchronous processing: Queues
- Load **Balancing** and Failover
- **Caching** Service
- **Locking** Things and preventing concurrent access: Lock service
- **Request scheduling** and control: Request multiplexing
- **Time** handling
- Providing **atomic transactions**: Consistency and persistence
- **Replicating** Things: NoSQL DBs

Availability

Is defined as:

$$Availability = \frac{Uptime_{agreed\ upon} - Downtime_{planned\ and\ unplanned}}{Uptime_{agreed\ upon}}$$

Continuous availability **does not allow for planned downtime**.

Typical Hardware Causes for Downtime

- Overheating
- PDU failure
- Rack-move
- Network rewiring
- Rack failures
- Racks go wonky
- Network maintenances
- Router reloads
- Router failures
- Individual machine failures
- Hard drive failures

Availability through Redundancy

Across groups of resources:

- Multi-site data center
- Disaster recovery
- Scalability

Within a group of resources:

- High availability (HA)
- Clustering
- Centralized administration (CA)
- Automatic failover (CO)
- Scalability
- Data replication
- Quorum algorithms (require multiple machines)

Between two resources:

3 Copy Disaster Recover Solution

- Maintains 3 copies of data/resources with at least 2 in different locations
- Enables quick switchover in case of disaster for business continuity
- Provides high availability and protects against data loss.

Serial vs. Redundant Availability

- **Serial chain** of components:
  - **Availability decreases with more members** in the chain
  - Individual components need higher availability
- **Redundant, parallel** components:
  - Unavailability of each component is multiplied and subtracted from 1 to determine overall availability
  - **Only one component needs to be up** to maintain availability.

Global Server Load Balancing

- DNS Round Robin:
  - Simple load balancing technique that distributes traffic to multiple servers based on the client's DNS query
  - Little mitigation in case of problems like overload, failure, etc.
  - Clients may disregard TTL settings
  - Takes approximately 15 minutes to drain traffic from troubled servers.
- BGP Anycast:
  - Uses BGP routing to direct clients to the nearest available server
  - BGP does not consider link latency, throughput, packet loss, etc. in selecting the best route
  - With multiple routes to the destination, BGP simply selects the one with the least number of hops
  - Troubleshooting can be demanding.
- Geo-DNS:
  - Uses the client's geographical location to determine the closest server for traffic distribution
  - Relies on the accuracy of the DNS provider's IP and location

Failover with Virtual IPs

Failover with one virtual IP:

- DNS points only to **one Virtual IP (VIP)**
- In case of a server failure, **client sessions are lost** but they can establish a new session on reconnect
- **No changes in DNS are required**, avoiding the potential issues of flushes and timeouts

Multi-site failover:

- A **combination** of geo-aware DNS and a Load Balancer/Fail-over front-server
- **Requests can be re-routed** to different locations in case of a server failure
- May still have the limitations and **issues associated with geo-aware DNS**.

Failover, Load Balancing and Session State

- **Sticky Sessions**: Keeps session state on a single server, offers advantages with a non-replicated system of records but limited in terms of fail-over and load-balancing options.
- Session Storage **in DB**: Session state is stored in a database, offers better scalability and fail-over options compared to sticky sessions.
- Session Storage **in Distributed Cache**: Session state is stored in a distributed cache, provides better performance and scalability compared to database storage, but still with fail-over options.

Today: Stateless servers with state in DB are the norm, but sticky sessions are still useful because records need to be replicated.

Compromise: Replicate sessions between pairs of servers, then enable switching between them as failovers

P2P Load Balancing

- **Evaluator Functions**: Access server stats in shared memory and determine the outcome of a request, whether it is handled by its own server, redirected, or proxied.
- **Server Stats**: Various metrics such as CPU usage, number of requests, memory usage, etc., are replicated in shared memory and used by evaluator functions to make load-balancing decisions.
- **Server Stat Replication**: The replication of server stats is done through multicast.

Requirements of Distributed Name/Directory Systems

Functional Requirements:

- **Re-bind/resolve methods** to store name/value pairs
- **Query Interface**
- **Name Aliases** for multiple logical hierarchies (**DAG-structured name space**)
- **Composite Names** (path names)
- **Location Independence** by separating address and location of objects
- Sound **security system** to ensure access to objects based on names.

Non-Functional Requirements:

Definition: Non-Functional Requirements are necessary functions of a system that **ensure speed, reliability, availability, security, etc**. Many systems fulfill functional requirements but fail to meet non-functional requirements.

Design of Distributed Name/Directory Systems

Name Space:

- The company name space organization is a **design and architecture concern**.
- **System Management** enforces rules, policies, and controls changes.
- **Different machines** host parts of the name space (zones).

Naming Service:

- **Interfaces** (naming service, factoryfinder, factory, account) allow administrators to **hide versions/implementations** from clients.
- Finders in a remote environment support **object migration and copying**.

Examples of Naming Services

- Domain Name System (DNS)
- X.500 Directory
- Lightweight Directory Access Protocol (LDAP)
- CORBA Naming Service
- Java Registry
- J2EE JNDI (mapped to CORBA Naming Service)

Google vs. Amazon

Amazon:

Services

- Storage Services:
  - S3 (huge storage capacity)
  - RDS
  - Aurora
  - Mongo
- Computation Services: EC2 (Virtual Machines)
- Queuing Services
- Load-balancing services
- Elastic map reduce
- Cloudfront (fast cache)
- Lambda
- Stepfunctions
- AI framework services

Best Practices for Designing Services

- Keep services **independent**
- **Measure** services
- Define **SLAs and QoS** for services
- Allow **agile development** of services
- Allow hundreds of services, **aggregate** them on special servers
- **Avoid middleware and frameworks** that force patterns
- Keep **teams small and organized around services**
- Manage **dependencies carefully**
- **Create APIs** for customer access to services

CQRS (Command Query Responsibility Segregation)

- **Separates the responsibilities of reading data** (queries) **and modifying data** (commands) into separate objects or services.
- **Improves scalability and performance** by allowing reads and writes to be **optimized separately**.
- **Promotes event-driven architecture** by allowing commands to trigger domain events.
- Simplifies domain modeling by **reducing the complexity of aggregates**.
- **Increases consistency** by using separate models for reads and writes.
- **Reduces the coupling** between the read and write sides of the system.

Requirements of Caching Services

- **Scalable** with ability to add machines
- **Avoid “thundering herds”** due to placement changes
- **Supports replication** of cache entries
- **High performance** required
- Optional **disk backup** support
- Supports **various storage mediums**, from RAM to SSD
- Supports **different cache replacement policies** with caution.

Handling Changing Machine Counts in Caching Services

- Problem: Changing Machine Count
- Solution: **Consistent Hashing (Ring)**
- **Machines are mapped into a ring** and their **position determines the key-space they are responsible for**.
- Machines can be assigned multiple virtual positions.

Consistent Hashing Algorithms

Simple Consistent Hashing Algorithm:

- URLs and caches are **mapped to points on a circle** using a standard hash function.
- URL assigned to **closest cache in clockwise direction**.
- Adding a new cache only reassigns the closest URLs to it, **items don't move between existing caches**.

Dynamo Consistent Hashing Algorithm:

- **Separates placement and partitioning.**
- Uses **virtual nodes** assigned to real machines for more flexibility.
- Virtual node is responsible for multiple real nodes.
- Improved load balancing due to **additional indirection**.

Cache Patterns

Pull:

- Occurs **during request time**
- Concurrent misses and client crashes **can result in outdated caches**
- **Complicated handling** of concurrent misses and updates
- Can be **slow and dangerous for backends**

Push:

- Automated **push updates cached values**
- Should **only** be used **for values that are always needed**

Pre-warmed:

- The system **loads the cache before the application starts** serving clients
- Used **for big applications with pull caches** to avoid boot issues

Cache Design Considerations

- **Kinds** of information fragments
- **Lifecycle** of fragments
- **Validity** of fragments
- **Effects** of fragment invalidation
- **Dependencies** between fragments, pages, etc.

Local vs. Distributed Events

Local:

- Observer updates sent on one thread
- If observer doesn't return, mechanism stops
- If observer calls back to observed during update call, deadlock can occur
- Solution doesn't scale and is not reliable (e.g. observer crashes result in lost registrations)
- Does not work for remote communication

Distributed:

- Various combinations of **push and pull models** possible
- Receivers can install **filters** using a constraint language to filter content (reduces unwanted notifications)

Asynchronous Event-Processing

- Publisher and subscriber communicate through **interaction middleware**
- Used to **decouple components** and asynchronous sub-requests from synchronous main requests (so that multiple fast tasks can run parallel to a slow main task)
- Implemented as **Message-Oriented-Middleware (MOM)** or socket-based communication library
- Can be implemented in **broker-less or brokered mode**.

Features of Event-Driven Interaction

- **Basic Event**: Any entity can send and receive events without restrictions or filtering.
- **Subscription**: A receiver can subscribe to specific events, making event delivery more efficient.
- **Advertisement**: The sender informs receivers about possible events, reducing the need for broadcasting.
- **Content-Based Filtering**: The sender, middleware, or receiver can apply filtering based on event content.
- **Scoping**: Administrative components can manipulate event routes, enabling invisible communication between components.

Types of Message Oriented Middleware (MOMs)

Centralized Message-Oriented-Middleware:

- Collects all notifications and subscriptions in one central place, enabling **easy event matching and filtering**
- Has a **high degree of control** and no security/reliability issues on clients
- Can create **scalability and single-point-of-failure problems**

Clustered Message-Oriented-Middleware:

- Provides **scalability** at **higher communication costs**
- Has lots of routing/filter-tables at cluster nodes, making **filtering and routing of notifications expensive**

Simple P2P Event Libraries:

- Local libraries are aware of each other, but **components are de-coupled**

ZeroMQ

- **Brokerless** socket library for messaging, with message filtering
- Connection patterns include **pipeline, pub/sub, and multi-worker**
- **Various transports**, including in process, across local process, across machines, and multicast groups
- **Message-passing process model** without the need for synchronization
- **Multi-platform and multi-language** support
- “Suicidal snail” fail-fast mechanism to **kill slow subscribers**

Horizontal vs. Vertical Sharding

Horizontal: Per (database) row, e.g. first 100 users are in shard 1, 200 in shard 2 etc.

Vertical: Per (database) column, e.g. profile and email is in shard 1, photos and messages in shard 2 etc.

Sharding Strategies

- Allow adding heterogenous hardware in the future
- Sharding should not make app code unstable
- Sharding should be transparent to the app
- Sharding and placement strategies should be separate

Horizontal Sharding Functions

Algorithms applied to the key (often: user ID) to create different groups (shards):

- **Numerical range**: users 0-100000, 100001-200000, etc.
- **Time range**: 1970-80, 81-90, 91-2000, etc.
- **Hash and modulo** calculation
- **Directory-based mapping** using a meta-data table for arbitrary mapping from key to shard

Consequences of Sharding

- **No more SQL JOINs**, leading to lots of copied data
- Increased need for **partial requests** for data aggregation
- **Expensive distributed transactions** required for consistency (if needed)
- Vertical sharding distributes related data types from one user, while horizontal sharding distributes related users from each other (**bad for social graph processing**)
- **SQL limitations** due to mostly key/value queries and problems with automatic DB-Sequences
- Every change **requires corresponding application changes**

Overview

- Within the database, **referential integrity** rules protect containment relationships
- **No equivalent in object space**
- No protection in distributed systems

For example when an employee leaves:

- All rights are cancelled
- Disc-space is archived and erased
- Databases for authentication and application-specific DBs are updated
- Badge no longer works
- All equipment has been returned

Functional Requirements of Relationship Services

- **Definition of relations between objects** without modifying those objects
- Support for **different types of relations**
- Ability to **create graphs of relations**
- Ability to **traverse relationship graphs**
- Support for **reference and containment relations**

Why Relationship Services Failed

The good:

- Powerful **modeling tool**
- Helps with creation, migration, copy, and deletion of composite objects
- Maintains referential integrity

The bad:

- Tends to create **many small server objects**
- **Performance impact**
- Not supported by many CORBA vendors for a long time
- EJB only supported with local objects in the same container.

Distributed Services and Algorithms II

Overview

- Classic (ACID) distributed consistency includes:
  - Distributed 2P locking
  - Distributed 2PC consensus
- ACID 2.0 eventual (coordination-free) consistency includes:
  - CAP and its children, CALM, CRDTs, etc.
  - Distributed replication (e.g. Cassandra)
  - CALM (Bloom) consistency
  - CRDTs (Conflict-free Replicated Data Types)
- Distributed Coordination (e.g. Chubby, ZooKeeper) includes:
  - Distributed consensus protocols
  - Cluster scheduler (e.g. Borg)

Why Truth is Expensive

- Strong consistency is discouraged.
- Coordination and distributed transactions slow down the process and affect availability.
- The cost of knowing the truth is high for many applications.
- The truth might only be a partial or outdated version.
- Availability is prioritized over consistency by making local decisions with available information.
- Improves the user experience by making this trade-off, most of the time.

Aspects of Classic Distributed Consistency

- **Distributed Objects and Persistence**: Objects that span across multiple systems and persist data in multiple locations.
- **ACID**: Atomicity, Consistency, Isolation, Durability - a set of properties that guarantee that database transactions are processed reliably.
- **Transactions**: A sequence of database operations that are executed as a single unit of work.
- **Isolation Levels**: The level of isolation between concurrent transactions, specifying how one transaction affects another.
- **Two-Phase Locking**: A protocol for enforcing serializable access to shared resources in a distributed system.
- **Distributed Transactions**: Transactions that span multiple systems and persist data in multiple locations.
- **Two-Phase Commit (2PC)**: A protocol for ensuring that a transaction is committed in a consistent state in a distributed system.
- **Failure Models for 2PC**: Models for how 2PC protocol handles system

Persistent Object Representations

- **Real storage object lives in a data store** and uses data store concepts for storage (e.g. a row in a table).
- Service works with object representations (“**Incarnations**”) and provides the illusion of a persistent object to clients.
- Java Connector Architecture provides an adapter interface for resource managers.

Mechanisms for Persistence

SQL Driver:

- Used to store object state.
- Suffers from “impedance mismatch”.
- Needs to control locking etc. in the service.

Object Relational Mapper (EJB/Hibernate):

- Used to store object state transparently for the programmer.
- Inheritance creates difficult problems for table mapping: Either performance or flexibility suffer.
  Just storing an object is simple - doing this in a way that protects from concurrent access, system failures, and across different data stores is much harder.

Persistent Object Mapping

- Enterprise integration software specializes in this kind of mapping
- Key to persistent mapping is **meta-information**:
  - **Generates object representations** for a service.
  - **Generates code** necessary for the data store to store the objects with its own mechanisms and objects.

Data Store Session Pooling

- **Number of channels** to a data store **is limited**
- If an object directly allocates a session (channel) and does not return it quickly, system throughput would become marginal
- **Session creation is expensive** (security!)
- Clients can either ask a pool for a session or the container framework can automatically allocate and return sessions
- Problems:
  - **Timeouts**
  - **Connection recycling**

Locking Against Concurrent Access

Binary locks:

- Used to synchronize an object, causing all clients except one to be blocked.
- Limitations: Binary locks are simple to use, but their performance suffers as they cannot distinguish between reads and writes.

Modal locks (read/write locks):

- Used to allow clients who only want to read to obtain read locks. Many concurrent read locks are possible.
- Advantages: Modal locks allow for a more nuanced approach to concurrent access, improving performance by allowing multiple read operations to occur simultaneously.

Lock Granularity: The granularity of locks (the scope of the resources being protected by the lock) affects the overall throughput of a system.

Optimistic Locking

Process:

1. Lock a row, read it along with its timestamp, and then release the lock.
2. Start a transaction
3. Write the data to the database.
4. Acquire locks for all data read and compare the data timestamps.
5. If one of them is newer, the transaction operation is rolled back, otherwise it is commited.

Advantages:

- Better overall throughput as locks are held for only a short period of time
- Timestamp comparison logic is implemented as a framework mechanism in the client session objects, simplifying the process

Serializability with Two-Phase Locking

Process:

1. Allocate all locks
2. Manipulate the data
3. Release all locks

Advantages: Requires that all locks be allocated before any data manipulation and released only after the manipulation is complete.

Guarantees serializability.

Deadlocks

- State where two or more processes are blocked because each **one is waiting for resources held by the other**
- Results in a situation where the processes cannot continue to run and are **stuck in a permanent waiting state**
- **Can occur in concurrent systems** where multiple processes access shared resources

Distributed Deadlock Detection

Local wait-for-graphs:

- **Correctness**: Based on the definition of a wait-for-graph, this method correctly detects deadlocks by identifying cycles in the graph.
- **Liveness**: This method can only detect deadlocks that exist within a single process or machine, so it may miss deadlocks in a distributed system.
- **Cost/complexity**: The cost of implementing this method is relatively low, as it only requires tracking locks and resource requests within a single process.
- **Failure model**: This method is susceptible to false negatives (missed deadlocks) in a distributed system.
- **Architecture type**: This method is suitable for systems with a centralized architecture, where all locks and resource requests can be monitored by a single process.

Classic ACID Definitions

- **Durability**: Ensures that once a transaction is committed, its effects **persist even in the case of system failures** (e.g. a crash that causes you to lose changes made to a word file)
- **Atomicity**: Ensures that a transaction is treated as a single, indivisible unit of **work that either happens in its entirety or doesn't happen at all** (e.g. in the case of a birthday party re-schedule where not all participants were caught in time)
- **Isolation**: Ensures that the concurrent execution of transactions results in a system state that would be obtained **as if transactions were executed serially** (e.g. if two people work on a shared file, their changes should not interfere with each other)
- **Consistency**: Ensures that the **system remains in a valid state after a transaction is executed** (e.g. after you complete a friend's work for the day, the tasks remain consistent, and the system remains in a valid state)

Transaction Properties and Mechanisms

- **Atomic Changes over Distributed Resources**: This is achieved through the use of consensus or voting algorithms such as two-phase commit.
- **Consistency**: This is maintained by observing consistency constraints between objects, such that the system remains in a valid state before and after a transaction is executed.
- **Isolation from Concurrent Access**: This is accomplished through the use of locking mechanisms, such as two-phase locking or hierarchical locking.
- **Durability of Changes**: This is ensured by transferring changes made to memory objects to persistent storage, to prevent loss in case of a system failure.

Serializability and Isolation

Definition: States that the outcome of executing a set of transactions should be equivalent to some serial execution of those transactions.

Purpose: The purpose of serializability is to ensure that each transaction operates on the database as if it were running by itself, which maintains the consistency and correctness of the database.

Importance: Without serializability, ACID consistency is generally not guaranteed, making it a crucial component in maintaining the integrity of the database.

Transaction API

1. System starts in a consistent state
2. Begins transaction
3. Modifies objects

Commit transaction:

- System has a new, consistent state
- Local objects are now invalid
- Changes are visible to others

On error: Rollback:

- Either from system or from client
- Only successful commit operations become the new state durable and visible to others
- Means going back to the beginning completely
- Client does not even know that they tried an operation

Components of Distributed Transactions

Process

- Begin()
- Commit()
- Rollback()

RPCs:

- Register (transactional servers)
- Vote (objects)
- Commit/rollback (objects, resource managers)
- Read/write/prepare (resource managers)

Components:

- Transaction
- Transactional client
- Transactional servers (objects)

Service Context

- Some **services require context** information to flow **with a call**
- **Security**: Needs to flow user information, access rights, etc.
- **Transactions**: Needs to flow information about ongoing transactions to participants
- The additional information **needs to be standardized** to allow different vendor implementations of services to interoperate.

Vote:

- To achieve atomic operations in a distributed setting, the **TA-Coordinator asks all participants for their vote** on committing or rolling back.
- Upon receiving a commit() call from a client, objects part of the TA **vote by asking resource managers (e.g. databases) to prepare for the commit**.
- A successful return of “prepare” from resource managers means that **both the object and the resource manager have promised to commit** the changes if the coordinator sends a commit.

Completion:

- The **coordinator is the only entity that can commit or abort** a TA after the prepare phase.
- If the vote phase was successful and all participants have prepared

Failure Models of Distributed Transactions

Work Phase:

- **If a participant crashes** or becomes unavailable, the **coordinator calls for a rollback**.
- **If the client crashes** before calling commit, the **coordinator will timeout** the TA and call for a rollback.

Voting Phase:

- **If a resource becomes unavailable** or has other issues, the **coordinator calls for a rollback**.

Commit Phase (Server Uncertainty):

- **In case of a crashed server**, it will consult the coordinator after restart and **ask for the decision** (commit or rollback).

Special Problems of Distributed Transactions

Resources:

- Participants in distributed TA's **consume many system resources** due to logging all actions to temporary persistent storage.
- **Large parts of the system may become locked** during a TA.

Coordinator as a Single Point of Failure:

- The coordinator must also prepare for a crash and log all actions to temporary persistent storage.

Heuristic Outcomes for Transactions:

- In certain circumstances, the outcome of a transaction may only be determined heuristically if the real outcome cannot be determined.

Transaction Types

Flat Transactions:

- Characterized by all-or-nothing behavior.
- Any failure causes complete rollback to original state.
- Can result in loss of significant amount of work if many objects have been handled.

Nested Transactions:

- Allow partial rollbacks with a parent transaction.
- Child TA rollback doesn't affect parent TA, but parent TA rollback returns all participants to initial state.
- Example: Allocation of a travel plan (hotel, flight, rental-car, trips, etc.).

Long-running Transactions:

- Challenge is resource allocation and increasing amount of work lost

ANSI Transaction Levels

Problems:

- **Dirty reads**: Occurs when a transaction reads data written by another concurrent transaction that has not yet been committed.
- **Non-repeatable reads**: Occurs when a transaction re-reads data it has previously read and finds that the data has been modified by another transaction that has since committed.
- **Phantom reads**: Occurs when a transaction re-executes a query returning a set of rows that satisfies a search condition and finds that the set of rows satisfying the condition has changed due to another recently-committed transaction.

Transaction Levels:

- **Read Uncommitted**:
  - Prevents: Nothing
- **Read Committed**:

Filesystem Block Order Guarantees

- BOB (Block Order Breaker) is a **tool used to evaluate the behavior of modern file systems** in regards to building crash consistent applications.
- It tests the **order guarantees of blocks** in file systems.

Eventually Consistent Storage Systems

- **Consistency without Coordination**: The design of modern data management systems has been impacted by the rise of Internet-scale geo-replicated services.
- **Weak Alternatives**: To reduce the cost of expensive coordination, many systems have sought weaker alternatives that still ensure application integrity.
- **Cost of Coordination**: Classic mechanisms like serializable transactions come with associated availability, latency, and throughput penalties.
- **When to Forego Coordination**: The question of when it's safe to forego the cost of expensive coordination versus when it's necessary to pay the price is an important one.

Grid Storage vs. NAS/SAN

Grid Storage:

- **Requires** POSIX-Grid **gateway**
- **Special caching may be needed** for video (readahead)
- Offers **huge bandwidth and scalability**
- May require **special maintenance**
- May be **proprietary**
- Supports **parallel processing**
- **Requires special applications**
- **Compatibility** with existing apps is **questionable**
- **Disaster recovery** across sites is **unclear**
- Requires **more electric power and space**

NAS/SAN:

- **POSIX**-compatible
- Special **caching is difficult** to implement

Forces behind NoSQL

- Need for low-latency and high-throughput access to data
- Difficulty in managing and maintaining consistency in a distributed system
- Increased focus on scalability and flexibility
- Changing data requirements and needs for real-time processing
- Cost and complexity of traditional RDBMs in large-scale systems
- Inability of RDBMs to handle large amounts of unstructured data
- The need for horizontal scaling in storage
- Lack of support for real-time, complex data processing using RDBMs
- The need for automatic scaling in storage to keep up with rapidly growing data
- Relaxed data consistency requirements in some applications.

Scaling Problems of RDBMs

- **Poor time complexity of SQL joins**: $O(m + n)$ or worse
- **Difficulty in horizontally scaling**, resulting in loss of joins or jumping between nodes
- **Unbounded nature of queries**, which can lead to a single query overloading a database
- **Optimized for storage efficiency** (no duplicates), integrity, and flexibility of access through arbitrary joins.

NoSQL Design Patterns

- Use **partition keys** with many distinct values for better scalability and data distribution.
- Opt for a **single table design with hierarchical modeling** and de-normalization to simplify the data structure.
- Ensure that values are evenly requested to avoid hot spots.
- Utilize **composite secondary keys** for 1:n and n:n queries.
- Limit query responses with paging token for better performance.
- Consider the use case and access patterns before finalizing the data layout.
- **Avoid relational modeling** and instead focus on simplifying the data structure.
- **Data integrity** is an application concern and **should be handled by the application logic**.
- Data storage efficiency is not a primary concern.

DynamoDB Design Principles

- Decentralized design with no single point of failure (no master node)
- Supports heterogeneous hardware
- Symmetric peers for better scalability
- Incrementally scalable to handle increasing load
- Eventually consistent data replication
- Requires a trusted environment for data security
- Replication support for higher data availability. Always-write enabled with conflict resolution during read
- Multi-version store with conflict resolution policies for better data management

Overview

- **Order-insensitive processing** using CALM (Consistency as Logical Monotonicity) principles in EC (Eventual Consistency) programs
- Converging replicated data types (**CRDTs**) divided into two types:
  - State-based CRDTs
  - Operation-based CRDTs

CALM Principle

- “Consistency as Logical Monotonicity”
- **Links consistency with logical monotonicity**, where monotonic programs ensure eventual consistency regardless of the order of delivery and computation.
- **Monotonic programs do not require coordination**, unlike non-monotonic programs where adding an element to the input set can revoke a previously valid output.
- **Non-monotonic programs require coordination** schemes that wait until inputs are complete before proceeding.

CALM Operations

Logically Monotonic:

- Initializing variables
- Accumulating set members
- Testing a threshold condition

Non-monotonic:

- Overwriting variables
- Set deletion
- Resetting counter
- Negation

CRDTs

State-based CRDTs:

- Calculate the new result at one node and then propagate it to replicas.
- The data structure must be commutative, associative, and idempotent, e.g., sets.

Operation-based CRDTs:

- Send the requested operation to each replica and calculate the results locally.
- The operations must be commutative with “exactly once” semantics (idempotent) and in FIFO order.
- These delivery guarantees are difficult to achieve, making state-based CRDTs more popular currently.

Bending the Problem

- **Separates data store and application-level consistency** concerns.
- CALM, ACID 2.0, and CRDT appeal to higher-level consistency criteria in the form of application-level invariants.
- Instead of requiring strong consistency for every read and write, the **application only needs to ensure semantic guarantees** (e.g., “the counter is strictly increasing”).
- This grants more flexibility in how reads and writes are processed.

Examples of CRDTs

Counters:

- Grow-only counter: Merge operation is max(values), payload is a single integer
- Positive-negative counter: Consists of two grow counters, one for increments and another for decrements

Registers:

- Last Write Wins register: Uses timestamps or version numbers, merge operation is max(ts), payload is a blob
- Multi-valued register: Uses vector clocks, merge operation takes both values

Sets:

- Grow-only set: Merge operation is union(items), payload is a set, no removal is allowed

Features:

- Configuration changes and notifications
- Updates for failed machines
- Dynamic integration and deconfiguration of new machines
- Elastic configuration with partial failures
- API for watches, callbacks, automatic file removal, and triggers
- Simple data model (directory tree model)
- High performance and highly available in-memory cluster solution
- No locks for updates, but total ordering of requests for all cluster replicas
- All replicas answer reads
- Wait-free implementation of coordination service with client API performing locks, leader selection, etc

Liveness and Correctness:

Zookeeper API

- **create**: Creates a node at a specified location in the tree
- **delete**: Deletes a node from the tree
- **exists**: Tests if a node exists at a specified location in the tree
- **get data**: Retrieves the data stored at a node
- **set data**: Writes data to a node
- **get children**: Retrieves a list of children of a node
- **sync**: Waits for data changes to be propagated to all nodes in the cluster.

Primary-Order Atomic Broadcast with Zab

- Primary sends **non-commutative, incremental state changes** to backup units
- **Order** of incremental changes **maintained even in case of primary crash**
- Multiple outstanding requests possible
- Identification scheme to **prevent re-ordering of updates**
- Synchronization phase to **ensure old updates delivered before new ones stored**.

Consistency Requirements for ABCast (Reliable Ordered Atomic Broadcast)

- **Validity**: If a correct process broadcasts a message, all correct processes will eventually deliver it.
- **Uniform Agreement**: If a process delivers a message, all correct processes will eventually deliver it.
- **Uniform Integrity**: Every process delivers a message at most once, only if it was previously broadcast by sender.
- **Uniform Total Order**: If processes p and q both deliver messages m and m0, their order must be the same.

Primary Order

- **Local primary order**: If primary broadcasts (v, z) before (v', z'), process that delivers (v, z) must have delivered (v', z') before (v, z).
- **Global primary order**: If Pi broadcasts (v, z) and Pj > Pi broadcasts (v', z'), process delivering both (v, z) and (v', z') must deliver (v, z) first.
- **Primary integrity**: If Pe broadcasts (v, z) and some process delivers (v', z') broadcast by Pe' < Pe, Pe must have delivered (v', z') before broadcasting (v, z).

HA Transactions

- **Provide transactional guarantees without unavailability** during system partitions or high network latency (Non-failing replica must respond)
- **Not CAP**: Can't provide linearizability as reading the most recent write from a replica
- **Not HAT-compliant**: Serializability, Snapshot Isolation, Repeatable Read Isolation
- **Possible with algorithms relying on multi-versioning and client-side caching**: Read Committed Isolation, transactional atomicity, etc.
- **Causal consistency with phantom prevention** and ANSI Repeatable Read need affinity with at least one server (sticky sessions)
- **Unable to prevent concurrent updates to shared data items**, cannot provide recency guarantees for reads.

Design of Distributed Systems

Overview

- Key principles for system design
- Utilizing caching and replication for efficient operations
- Importance of architecture in optimizing performance
- Validation of architectural design
- Techniques for improving performance in large fan-out architecture
- Strategies for achieving fault-tolerance in high-scale systems.

Overview

- **Consideration of Latency**: Examination of buffering and round-trip times
- **Importance of Locality**: Proper placement of heavily interacting components
- **Avoiding Duplication of Work**: Utilizing resources effectively
- **Resource Pooling**: Reusing resources in communication such as connections or thread pools
- **Parallelization**: Design for concurrent operations and minimize serialization
- **Evaluating Consistency**: Determining the appropriate level of consistency with caching and replication
- **Caching and Replication Strategies**: Utilizing prediction and bandwidth to reduce latency
- **End-to-end Argument**: Minimizing heavy guarantees at lower levels of the system.

Sharing Ressources and Data

- **Pooling resources** can improve performance even in local systems
- High-frequency requests can lead to memory allocation issues and poor performance
- **Caching is crucial** for the effectiveness of distributed applications
- **Minimizing backend requests** while maintaining sane application logic
- **Breaking down information** into smaller fragments can reveal reusable parts

Connection Pooling

- **Matching** server and database CPU **capabilities**
- **Avoiding blocking** and app threads holding onto connections
- **Careful monitoring of wait time** in the pool
- **Checking I/O** rates with new hardware
- **Understanding what constitutes a “connection”** to storage
- **Monitoring core/thread ratio**, etc.

Horizonal Scaling/Parallelization

- Horizontal scaling through **parallel processing**
- Every **request can be handled by any thread on any host**
- **Avoid synchronization points** in servlet engines or database connections.

Caching and Replication

- Caching components are responsible for maintaining data validity
- Data source is responsible for keeping replicas consistent and up-to-date
- Focus on reducing back-end requests for improved efficiency.

End-to-End Argument

- **User/Developer**: Compensation for behavior through application
- **Application Layer**: Use of special commands, such as “Select for Update” or “Begin Transaction”
- **Intermediate Layer**: Compiler/Languages utilizing technologies such as Software Transactional Memory and memory models
- **Base Layer**: Considerations for CPU cache coherence, database isolation levels, and real-time streaming, etc.

Design Methodology

- **Back-of-the-envelope** calculations
- Decide on **geographical distribution and replication strategy**
- Determine **data segregation**, including single or multi-tenancy models and partitioning
- Divide **business requirements into REST**-like services
- **Define SLAs for services**, including availability, latency, throughput, consistency, and durability
- **Define security context** with IAAA (Identity, Authentication, Authorization, Audit) and perform risk analysis
- Complete **monitoring and logging setup**
- **Plan for deployment, release changes, testing, and maintenance** using fault-tolerant features.

Uncomfortable Real-World Questions

- How many application servers are needed to support the customer base?
- What is the optimal ratio of users to web servers?
- What is the maximum number of users per server?
- What is the maximum number of transactions per server?
- Which specific hardware configurations provide the best performance?
- What is the current production server capability?
- What do the users do? (These are business process definitions.)
- How fast do the users do it? What are the transaction rates of each business process?
- When do they do it? What time of day are most users using it?
- What major geographic locations are they doing it from?
- How many connections can the server handle?
- How many open file descriptors or handles is the server configured

Overview

- **Information** Architecture
- **Distribution** Architecture
- **System** Architecture
- **Physical** Architecture
- **Architectural** Validation

Information Architecture (to analyze Caching)

Defines pieces of information to aggregate or integrate

Data/changed by	Time	Personalization
Country Codes	No (not often, reference data)	No
News	Yes (aging only)	No, but personal selections
Greeting	No	Yes
Message	Yes (slowly aging)	Yes

tells portal how to map/locate fragments defined in the information

Data Type	Source	Protocol	Port	Avg. Resp.	Worst Resp.	Downtimes	Max Conn.	Loadbal.	Security	Contact/SLA
News	hostX	http/xml	3000	100ms	6 sec.	17.00-17.20	100	client	plain	Mrs.X/News-SLA
Research	hostY	RMI	80	50ms	500ms	0.00-1.00	50	server	SSL	Mr.Y/res-SLA

Additional factors to consider:

- Available bandwidth
- Number of planned requests
- Distance to the device
- Availability numbers

Service Access Layer

Is determined by the distribution architecture.

- Handle changes in the interface
- Monitors backend system connections
- Disable connections that are not functioning properly (“fail fast”)
- Add new sources to the system
- Poll and re-enable sources that have been temporarily disabled
- Keep track of statistics on all sources.

Simple Alternative: Sidecar, contains circuit breaker & service discovery

Advanced Alternative: Service mesh with separate data and control plane

Physical and Process Architecture

Physical Architecture:

- Deals with reliability issues (replication, high-availability, etc.) and scalability (horizontal and/or vertical)
- Need to define scalability methods from the beginning due to their impact on overall system architecture

Horizontally scalable application:

- Replicated on multiple hosts
- Avoids single point of failure

Vertically scalable application:

- Can only install more CPUs or RAM on single instance of host
- Limited scalability and availability (HA application)

Architecture Validation

In the architecture validation phase these questions are answered: How does the architecture ...

- Handle security and privacy?
- Handle data consistency and durability?
- Handle disaster recovery and business continuity?
- Handle performance, scalability and capacity?
- Handle integration with other systems and data sources?
- Handle upgrades, maintenance and support?
- Align with the organization's goals, strategies and plans?

Problems with Naive Application Designs

The portal had no caching etc.

- **GUI architecture**: Long time with empty page
- **System architecture**: No System Architecture Diagram!
- **Performance**: Very slow construction of home-page
- **Reliability**: Frequent stalls and crashes of the application
- **Throughput**: 10 users max. with top-notch hardware!
- **Team**: Little understanding of performance or architecture

Improving the GUI Architecture

- Minimize HTTP requests by combining files and using sprites
- Use asynchronous loading for non-critical resources
- Avoid heavy use of animations and dynamic effects
- Preload essential resources
- Use a content delivery network (CDN) for static resources
- Lazy load images and videos
- Use browser caching effectively
- Minimize the use of plugins and third-party scripts.
- Minimize the number of DOM manipulations
- Use lazy loading for images and other resources
- Avoid using large, blocking scripts
- Use a content delivery network (CDN) to distribute resources
- Consider using server-side rendering or progressive web apps (PWA)
- Implement caching strategies at the server and client side
- Monitor and optimize the page load time regularly

Improving the System Architecture

Use a system architecture diagram:

- Show the main components and their interactions
- Indicate the flow of data and processing
- Highlight the main functionalities and relationships between components
- Reflect the overall structure and design of the system
- Provide a clear and concise representation of the system architecture.

Lessons Learned from Naive Portal Designs

- Request time is affected by the **sum of individual calls**
- **Each delay** in a call **contributes** to the runtime
- Back-end server problems lead to poor request time
- **Long timeout settings** negatively impact response times
- Small improvements in sub-requests matter with many concurrent requests
- **Lack of central architecture diagram** limits understanding of performance impact and throughput
- **JEE restrictions** prevent creating custom threads.

Overview

Calls are parallel instead of serial.

- The overall **request time is determined by the slowest sub-request**
- Each **delay in an individual call adds to the runtime**
- Long timeout settings negatively impact response times
- Using **short timeouts** for back-end server calls is **recommended**
- Running short requests in separate threads may not be productive, **consider request bundling**
- **Error from one sub-call should not block the whole request**, have a fallback
- **Avoid all threads getting stuck** on a dysfunctional sub-call (bulkhead)
- Temporarily **close dead connections (circuit-breaker)**.

Reliability Issues in Dependencies

- System load becomes worse due to **hanging requests** occupying resources and leading to heavy garbage collection
- Dead servers can cause a **buildup of threads** due to even short timeouts
- The portal was **frequently impacted by failing back-end servers**
- **Avoid lengthy waiting time** for sub-requests in the homepage action handler: Adopt the “Fail-fast” pattern today.

Fragments

Pages: Unique to customers, cannot be re-used

Page fragments:

- Can be shared and heavily re-used
- Allows huge reduction in back-end requests
- Downside: If fragments change, mechanism needed to invalidate dependent pages.

Latency Reduction & Tolerance

- **Keep response times tight** but aware of stragglers
- **Fight stragglers** with backup requests and cross-server cancellation
- **Watch for overload** at sender when responses come back
- **Do NOT distribute load evenly**, synchronize background load across machines instead
- Reduce **head-of-line blocking** (partition large requests)
- **Partition** data across machines
- Cheat by **coming back with partial data**
- Cross request adaptation
- Increase **replication** count
- Beware of the **incast** problem

Avoiding Getting Stuck

- **Fail Fast**: Don't wait for problematic resources
- **Timeouts**: Use timeouts when accessing a service
- **Exponentially Decreasing Retries**: Use if needed
- **Fallback**: Use alternatives when service doesn't work, such as serving stale data
- **Caching**: Retrieve data from cache if real-time dependency is unavailable, even if data is stale
- **Eventual Consistency**: Queue writes to be persisted once dependency is available
- **Stubbed Data**: Revert to default values if personalized options can't be retrieved
- **Empty Response (Fail Silent)**: Return null or empty list that UIs can ignore.

Circuit Breakers

- Purpose: **Handle faults that might take a long time to recover from**
- Provide control mechanism to **prevent application from continually trying** to perform a failing operation
- Allows application to **fail fast and respond to failures quickly**
- Acts as a switch that “trips” when system detects a failure
- Stops application from making further attempts to perform operation until reset
- Helps **prevent application from becoming unresponsive**
- **Protects other parts of the system** from being affected by the failure.

Bulkheads

- **Bulkhead pattern** is a design for fault-tolerant applications
- Elements of an application are **isolated into pools**
- If one pool fails, **others will continue to function**
- Named after the sectioned partitions (bulkheads) of a ship's hull
- Example: **semaphores and thread pools**

Blast Reduction

- Partition app into geographical regions (e.g. US, DACH etc.)
- Splitting regions further into specific availability zones and further cells
- Shuffle sharding: Provide a single-tenant-like isolation for shared workloads
- Splitting app itself into separate control and data planes