Uni Distributed Systems Summary

Summary for the distributed systems course at HdM Stuttgart

Felicitas Pojtinger

2022-12-17

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. :)

Figure 2: AGPL-3.0 license badge

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

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

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 Distributed Systems hard for Devs?

- 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**

8

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

Characteristics of Distributed Systems

- **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,

- **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

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

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

- **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.

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

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 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

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

- **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

- 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

- **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.
- **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.
- **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 polling pattern, as it allows the system to respond to I/O operations as soon as they are

- 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

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

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

- **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

Idempotency

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

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

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

- **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 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

- **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

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

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

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**: 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 (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

73

Mechanics of Remote Calls

- **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.

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.

- **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

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

Object Models and Type Systems

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

- 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

- 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

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

- 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

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

- **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 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.

- 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

- **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).

- 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 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.

- 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.

- 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

- 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.

- 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.

- 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.

- 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 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**.

- 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.

- 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 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.

The EJB Container

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 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.

- 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.

- 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).

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

- **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**

- 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.