Distributed Systems Programming (DSP): Notes

Antonio Battipaglia & **ChatGPT**

June 11, 2023

Contents

Acknowledgments

These notes are related to the ***DSP (Distributed Systems Programming) Course*** of ***Politecnico di Torino*** and have been written with the assistance of *ChatGPT*. They are provided as-is and have not been reviewed by any professor; therefore, there may be errors or typos. I take no responsibility for their usage.

1 Dystributed Systems Architectures

The architecture of distributed systems involves the design and organization of a system that is composed of multiple interconnected components or nodes, where these components work together to achieve a common goal. In a distributed system, the components may be geographically dispersed, operate concurrently, and communicate with each other over a network. The main goal of distributed system architecture is to enable efficient and reliable communication, coordination, and cooperation among the components, while addressing challenges such as scalability, fault tolerance, consistency, and performance. Distributed system architecture can vary widely depending on the specific requirements and constraints of the system, and it may involve various architectural styles, communication paradigms, protocols, and technologies. Overall, the design of distributed system architecture plays a critical role in ensuring the effective functioning and performance of complex distributed systems.
There might be multiple definitions of a distributed system, since it can be seen as:

- Networked Computers which communicate each other.
- Autonomous computing elements acting as a single coherent system.

A more generic term to refer to *"computing element"* is *process* or *node*.

1.1 DS: Main Properties

The keywords here are *autonomy* and *networked* since they both introduce important concepts and implications.

- **Autonomy** implies:
  - *Asynchronous and Concurrent*
  - *Heterogeneous*: there are different hardware, OS, programming languages and locations.
  - Possibility of *failure*.
- **Networked interconnection** leads to:
  - Communications take *long or variable time and can fail*.
  - High Impact and large attack surface for malicious intruders.

1.2 DS: General Requirements

In this discussion, we will explore the main requirements for a distributed system and delve into their related implications. As already explained, a distributed system is a complex network of interconnected components that work together towards a common goal. Understanding these requirements and their implications is crucial for effectively designing and implementing distributed systems.

1.2.1 Distribution Transparency

Distribution transparency is a key concept in distributed systems architecture, which refers to the ability of hiding the internal details of a distributed system from its users. The idea is to abstract the underlying complexities of a distributed system and provide a simplified and consistent view to its users, regardless of the actual distribution of data and computation across different nodes.There are several aspects of distribution transparency that need to be considered:

- *Location transparency*: this refers to hiding the details of where data or computation is located in the distributed system. Users should be able to access data or execute operations without knowing the physical location of the resources. This can be achieved through techniques such as naming and addressing, where logical names are used to identify resources instead of their physical addresses.

- *Replication transparency*: this refers to hiding the details of data replication in the distributed system. Users should be able to access replicated data without being aware of the existence of multiple copies. Replication can improve availability, fault tolerance, and performance, but managing replicas can be complex. Replication transparency ensures that users do not need to be aware of the replication mechanism.
- *Access transparency*: this refers to hiding the details of how data is accessed or operations are executed in the distributed system. Users should be able to access data or execute operations using a consistent interface, regardless of how the data is internally stored, accessed, or shared among different elements of the distributed system. Access transparency can be achieved through standard APIs or middleware that provide a uniform interface to access diverse resources.
- *Failure transparency*: this refers to hiding the details of failures in the distributed system. Users should be able to access data or execute operations even in the presence of failures, without being aware of the failures or their recovery mechanisms. Failure transparency can be achieved through fault tolerance techniques such as replication, redundancy, and error recovery mechanisms.

However, achieving full distribution transparency is not always possible or practical. For example, the physical location of resources can affect latency, and failures may disrupt transparency. In some cases, location-aware or context-aware services may be more appropriate, where the system takes into account the location or context of the user or resources to optimize performance or provide specialized services. There is often a tradeoff between transparency and performance. While distribution transparency can provide a simplified and consistent view of the distributed system to its users, it may introduce overheads in terms of communication, coordination, and management.
The design of distributed systems needs to carefully balance the tradeoff between transparency and performance based on the specific requirements and constraints of the system.

1.2.2 Dynamic Membership

Dynamic membership is a key requirement in distributed systems, referring to the ability of a system to handle changes in the membership of its components during runtime. This means that processes within the distributed system should be able to join or leave the system dynamically, without disrupting the overall functionality of the system.
The implications of dynamic membership are manifold:

1. The system must provide **seamless mechanisms for adding or removing processes**. This may involve protocols for detecting when a new process joins the system or when an existing process leaves, and updating the membership information accordingly. Maintaining consistency across the distributed system despite the dynamic changes in membership can be challenging.
2. **Processes** within the distributed system **must be able to locate other components**, including new ones that join the system after the system has started running. This requires mechanisms for discovering the current membership and location of processes, such as naming services, directory services, or dynamic lookup protocols.
3. Furthermore, **it poses challenges in terms of maintaining system stability, consistency, and fault tolerance**. For example, adding a new process may require redistributing data or workload among the existing processes to maintain load balancing or data consistency. Similarly, removing a process may require reassigning its responsibilities to other processes in a graceful manner to prevent data loss or service disruptions.

In conclusion, dynamic membership is a critical requirement in distributed systems, as it enables the system to adapt to changes in the availability, scalability, and reliability of its components, while ensuring smooth operation and continued functionality.

1.2.3 Security

Security is a critical requirement in distributed systems, encompassing the protection of data and services from unauthorized access or interference. It typically involves ensuring confidentiality, access control, integrity, and availability of the system and its components.
The main security properties in a distributed system include:

- **Confidentiality**: Ensuring that data or services are not accessed by unauthorized entities.
- **Access control**: Restricting access to data or services based on predefined permissions or privileges.
- **Integrity**: Preventing unauthorized modification or tampering of data or system behavior.
- **Availability**: Ensuring that the system remains operational and accessible to legitimate users.

These security properties are typically formulated as requirements that specify the level of protection against certain capabilities or attacks. The goal is to make it difficult for an attacker with specific capabilities to compromise the system or its components.

1.2.4 Scalability

Scalability refers to the ability of a distributed system to adapt and maintain its performance as its dimensions, such as the number of users, resources, distance among users/components, or organizational units, increase. Scalability is an important consideration in distributed systems as they need to handle growing demands and evolving requirements.
Different types of scalability in distributed systems can include:

- **Size scalability**: This refers to the ability of a system to handle an increasing number of users, resources, or data without significant performance degradation. A scalable distributed system should be able to efficiently accommodate a growing user base or an increasing volume of data without sacrificing its performance.
- **Geographical scalability**: This relates to the ability of a distributed system to span across different geographical locations or distances among users or components. As distributed systems can operate across multiple geographic locations, geographical scalability is important in ensuring efficient communication and coordination among distributed components that may be physically separated.
- **Administrative scalability**: This pertains to the ability of a distributed system to handle an increasing number of organizational units or administrative domains. As distributed systems can involve multiple organizations or administrative units, administrative scalability is crucial in managing complex distributed environments.

Since it is a complex requirement to fulfill, scalability in distributed systems presents several challenges, including:

- **Resource limitations**: Distributed systems often operate under resource constraints such as processing power, memory, bandwidth, or storage. Ensuring efficient resource utilization and management is essential to maintain scalability and avoid performance bottlenecks.
- **Latency and reliability**: Distance among users or components in a distributed system can result in increased latency and reduced reliability. Managing and mitigating the effects of latency and reliability limitations, which are inherent in distributed systems, is critical to achieving scalability.

In summary, achieving scalability in distributed systems involves addressing size scalability, geographical scalability, administrative scalability, and managing challenges related to resource limitations, latency, and reliability. Scalability is a key consideration in designing and implementing distributed systems to ensure they can effectively handle increased demands and changing requirements.

1.2.5 Openness

Openness refers to the extent to which a distributed system offers components that can be easily used by or integrated into other systems, regardless of their types or technologies. An open distributed system is designed to be interoperable and accessible to other systems, allowing for seamless integration and interaction.
Related requirements of openness might include:

- **Simplicity**: An open distributed system should be designed with simplicity in mind, making it easy for other systems to understand and use its components. Simplicity in design and implementation promotes ease of integration and reduces the complexity of interfacing with the distributed system.
- **Modifiability**: An open distributed system should be designed to be easily modifiable, allowing for updates or changes to its components without disrupting the integration with other systems. Modifiability is important in accommodating evolving requirements or technologies, and enabling smooth integration with other systems over time.
- **Documentation**: An open distributed system should be well-documented, providing clear and comprehensive documentation on its components, interfaces, protocols, and other relevant information. Documentation is essential in facilitating the understanding and usage of the distributed system by other systems or developers.

In a nutshell, openness in distributed systems involves designing systems that are interoperable, accessible, and easily integrated with other systems, while addressing requirements such as simplicity, modifiability, and documentation. An open distributed system promotes seamless integration and collaboration with other systems, enabling interoperability and extending the capabilities of the overall distributed ecosystem.

1.3 DS: General Software Architectures

The software architecture of a distributed system is typically based on the layered model, such as the OSI (Open Systems Interconnection) model, which consists of different layers responsible for different aspects of communication and data processing. In a distributed system, the software architecture is made up of a collection of processes that are physically executed on possibly different network hosts, and these processes interact with each other according to a stack of protocols.
The OSI layers that are relevant for the distributed system programmer are the application-oriented layers, which include layers 5 (session layer), 6 (presentation layer), and 7 (application layer). These layers are responsible for managing the application-level communication and processing, such as establishing sessions, handling data presentation and formatting, and implementing application-specific protocols.

1.3.1 Middleware: A Software Layer

To manage the complexity of the many requirements of a distributed system, a common approach is to build applications on top of a software layer called ***middleware***. Middleware acts as an intermediate layer between the application layer and the lower-level system layers, providing a set of common services and abstractions that simplify the development of distributed applications.

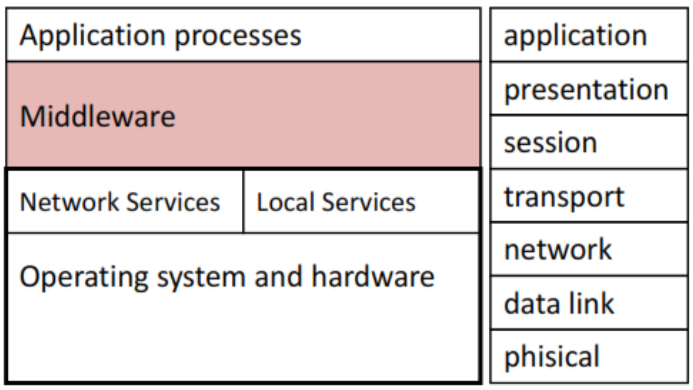

Figure 1: Middleware Position in the OSI Model

Moreover, it provides business-unaware services for coordination and communication among remote processes in a distributed system. It acts as an intermediate layer that hides the complexity of communication through the network, process and host heterogeneity, security issues, and other underlying details.
Examples of software that can be classified as middleware include web browsers and database drivers. Web browsers provide services for accessing web resources and displaying web pages, while database drivers facilitate communication between applications and databases for data retrieval and manipulation.
On the other hand, examples of software that cannot be classified as middleware are business-aware systems, such as an airline reservation system. These systems are designed for specific business purposes and are aware of the domain-specific rules and requirements, unlike middleware which is business-unaware and provides generic services for coordination and communication.

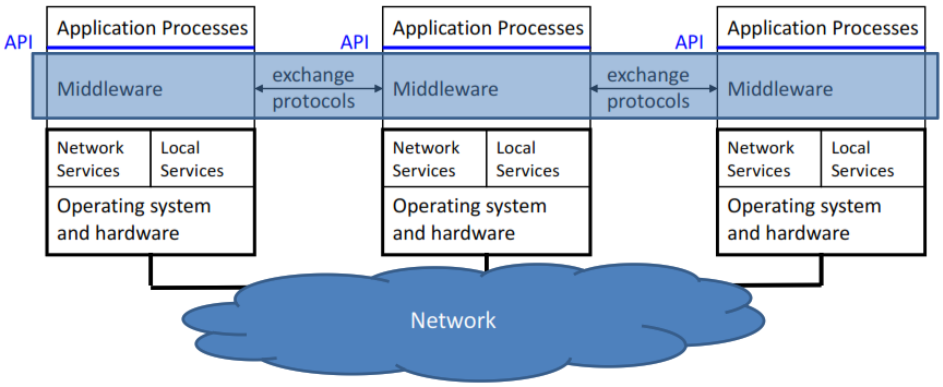

Figure 2: Acting of a Middleware

Typical services provided by middleware can be categorized into three main categories:

1. **Interaction Services**: These services include information exchange, connection management, session management, deadlock avoidance mechanisms, and other mechanisms that facilitate communication and coordination among remote processes.

2. **Services for accessing specific applications**: Middleware also provides services for accessing specific applications, such as database access, transaction processing, and distributed object management. These services enable applications to interact with underlying resources and data in a transparent manner.

3. **Management, Control, and Administration Services**: Middleware offers services for managing, controlling, and administering the distributed system. These services may include directory services for locating resources, security services for authentication and authorization, performance monitoring services for tracking system performance, and other administrative functions.

In summary, middleware provides business-unaware services for coordination and communication among remote processes in a distributed system, hiding underlying complexities. Examples of middleware include web browsers and database drivers, while business-aware systems are not classified as middleware. Middleware services can be categorized into interaction services, services for accessing specific applications, and management, control, and administration services.

1.4 DS: Main Design Strategies

In this section, we will explore the main design goals and strategies for DS, including limiting interactions, using middleware solutions, and building applications relying only on APIs. These principles aim to promote distribution transparency, scalability, code reuse, and information hiding, which are crucial for the successful implementation of distributed systems.

1.4.1 Scalability Strategies

Scalability is a critical factor in designing distributed systems to handle increasing demands. There are two common strategies for scalability:

- **Scaling up**: It involves improving the capacity of the computing, storage, or communication elements, such as replacing CPUs with more powerful ones. However, scaling up can only be applied to a limited extent due to resource limitations.
- **Scaling Out** : it involves modifying the way the system is constructed or operates, such as increasing the number of deployed machines in the system. It is typically employed in world-size scaling distributed systems, as it allows for a more distributed and parallel approach to handle larger workloads.

Focusing on scaling out techniques, since have a bigger impact on world-size distributed systems, there are several main techniques that are commonly used, including: partitioning and work distribution, replication, and communication latency hiding/limitation.

- **Partitioning and work distribution**: it involves breaking down a large task or workload into smaller, more manageable pieces and assigning them to different processes or nodes in the system. As demand increases, more processes can be added to handle the increased workload. Examples of systems that use this technique include the Web and the Domain Name System (DNS).
- **Replication**: involves creating copies of processes or data to distribute the load and increase redundancy. For example, caching is a form of replication where frequently accessed data is stored in multiple locations for faster access. Web server clustering is another example, where multiple servers are used to distribute the load and improve availability.
- **Communication latency hiding/limitation**: it is another technique used for scaling out. Asynchronous communication is used instead of synchronous communication to reduce the amount of time spent waiting for a response. Additionally, processing data where it is located rather than moving it around can help reduce communication latency.

1.4.2 Strategies for Openness

Strategies for Openness in distributed systems focus on using standard or documented/public protocols and APIs, while leaving implementations free to adhere to agreed protocols and APIs. This approach promotes neutral interfaces, interoperability of different implementations, and portability of implementations.
Using standard protocols and APIs ensures that systems can communicate and interact with each other using widely accepted standards, enabling interoperability between different implementations. This allows components or systems from different vendors or developers to work together seamlessly.
Composability and extensibility are also important strategies for openness. Systems should be designed to be composed of several simple and easily replaceable elements. This allows for flexibility in adding, replacing, or modifying components in the system without disrupting its overall functionality. This approach promotes modularity and adaptability, making it easier to extend or modify the system in the future.

1.5 DS: Architectures Classification

The classification of distributed systems architectures can be broadly categorized into three main types:

1. **Client-Server**: This architecture involves a centralized organization where clients request services from a central server. The server is responsible for processing these requests and providing the required services. Clients and servers have distinct roles, with clients typically being user-facing interfaces and servers handling the processing and storage of data. This architecture is commonly used in systems where there is a clear separation of responsibilities between clients and servers, and clients rely on servers for their functionalities.

2. **Peer-to-Peer**: In this fully decentralized architecture, all nodes in the system have the same capabilities and responsibilities. Each node can act as both a client and a server, and can request and provide services to other nodes in the network. Nodes collaborate with each other in a peer-to-peer manner, without a central server or authority. This architecture is often used in systems where there is no central point of control and nodes are expected to share resources and services directly with each other.

3. **Hybrid**: The hybrid architecture combines elements of both client-server and peer-to-peer architectures. It may involve a mix of centralized and decentralized components, or a combination of client-server and peer-to-peer interactions. This architecture is often used in systems where a combination of centralization and decentralization is desired, based on specific requirements or constraints.

These different architectures have their strengths and weaknesses, and the choice of architecture depends on the specific needs, goals, and constraints of the distributed system being designed. These models will be discussed in the next paragraphs.

1.5.1 The Client/Server Interaction Model

The Client-Server (C/S) interaction model is currently the most commonly used model in distributed systems. In this model, interactions occur between two processes, where one process plays the client role and the other process plays the server role. Each interaction is based on a message exchange, where the client sends a request and the server sends back a response.
It's important to note that the role of client or server refers to how a process plays a single interaction, and a single process can play the client role in some interactions and the server role in other interactions. In a client-server architecture, processes are divided into two classes:

- **Client Processes**: only play the client role.
- **Server Processes**: play the server role and offer some service, such as computation or storage, to many clients.

The Client-Server interaction model is widely used due to its simplicity and ease of implementation. Clients can request services from servers, and servers can respond to these requests in a controlled and organized manner. This model allows for centralized management and control, making it suitable for systems where there is a need for a clear separation of responsibilities and a central authority to coordinate and manage the interactions between clients and servers. However, it may also introduce potential scalability and single-point-of-failure issues, as the server may become a bottleneck or a point of failure if not designed and managed properly.

1.5.2 Application Tiers and Physical Tiers for C/S Architecures

In a Client-Server (C/S) system, each application can be logically divided into three main parts, also known as logical tiers: the user interface tier, the processing tier, and the data tier. These logical tiers can be mapped onto different processes running on possibly different hosts, also known as physical tiers. There are different possible mappings between the logical and physical tiers, depending on the system architecture.

C/S 2-tier Architectures
C/S 2-tier architectures consist of only two physical tiers: one client tier (front end) and one server tier (back end). This is the classical architecture, typical of the first generation of C/S systems. There are different configurations possible, depending on how the logical tiers are mapped onto the physical tiers.

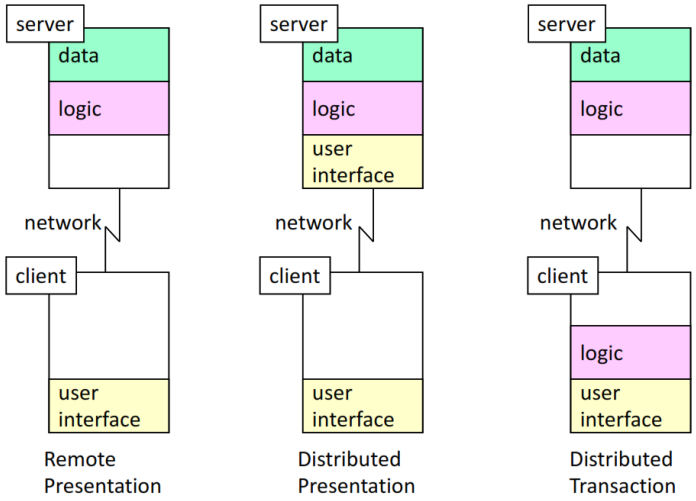

Figure 3: Client/Server 2-Tiers Architecture

C/S 3-tier Architectures

C/S 2-tier architectures may have limitations in scalability and flexibility. These limitations can be overcome by introducing an intermediate physical tier, also known as a middle machine. This middle machine can have various functions such as workload balancing on different back ends, filtering (e.g. firewall), protocol conversion, access to legacy systems (gateways), or simultaneous access to several servers offering added-value services. Another possibility is to add additional tiers to delegate and decouple specific application components, such as data.

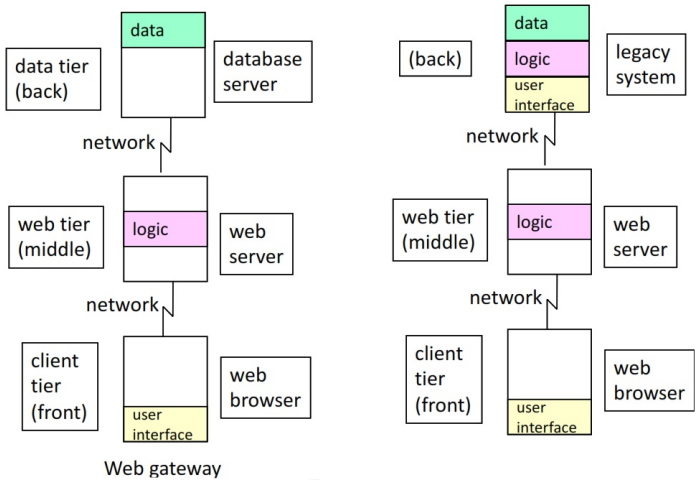

Figure 4: Client/Server 3-Tiers Architecture

C/S multi-tier Architectures

For even further flexibility and complexity, the number of physical tiers in a C/S system can be further increased. For example, a 4-tier architecture can be designed to achieve specific system requirements and performance goals. This allows for more granular control and customization of the system architecture to meet the specific needs of the application and environment.

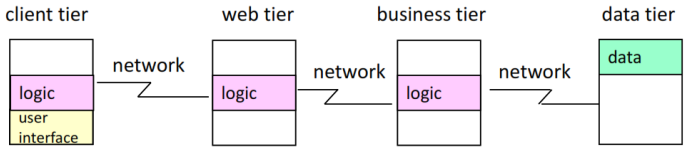

Figure 5: Client/Server Multi-Tiers Architecture

1.5.3 Peer-to-Peer (P2P): A Decentralized Architecture

In a Peer-to-Peer (P2P) architecture, all processes are considered as identical peers, capable of both client and server roles. Each peer has the ability to initiate a client-server (C/S) interaction with any other peer that it knows, at any time. This means that in a P2P architecture, there is no distinction between clients and servers, and all peers have equal capabilities and responsibilities.
Peers can initiate and respond to requests from other peers, making the architecture highly decentralized and distributed, with no central authority or control. P2P architectures are commonly used in systems where peers need to share resources, collaborate, or communicate directly with each other without relying on a central server. Examples of P2P systems include file sharing networks, distributed computing networks, and blockchain networks.

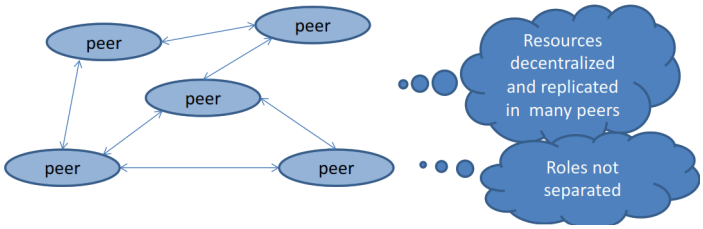

Figure 6: P2P Architecture

1.5.4 Application Tiers and Physical Tiers for P2P Architecures

In a Peer-to-Peer (P2P) system, all peers perform the same tasks, but on their own share of the complete data set. The data set can be partitioned among the peers, or data can be replicated to increase reliability and performance.

Overlay Network
A P2P system is built by its peers, which create an overlay network. Each peer can communicate only with some other peers through virtual channels, typically implemented by TCP connections. One key aspect of each P2P architecture is how the overlay network is created and managed. There are two main types of overlay networks used:

- **Structured Systems**: In structured P2P systems, the overlay network is built with a deterministic, regular topology, such as a ring, tree, or grid. Topology-specific routing algorithms are used to reach specific peers in the overlay network.

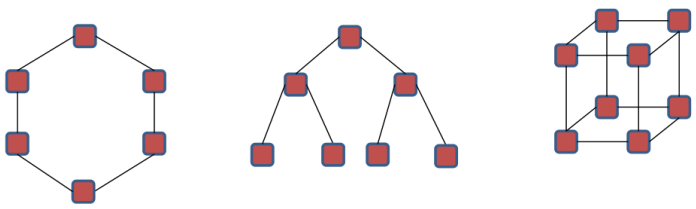

Figure 7: P2P Overlay Network: Structured System

- **Unstructured Systems**: In unstructured P2P systems, each peer connects to some other peers in a more or less random way, such as through the discovery of nearby peers. Different strategies

can be used to reach other peers in the overlay network, such as flooding, random walks, or policy-based search.

1.5.5 Client/Server vs. P2P

During design phase, in the selection of the architecture, a comparison between Client-Server (C/S) and Peer-to-Peer (P2P) architectures is needed.
There are several key differences to consider:

1. **Scalability and Reliability**: In C/S architectures, server processes may become bottlenecks as they need to run on powerful and reliable hosts. They can also be single points of failure, and can become congested with requests, affecting system performance. On the other hand, in P2P architectures, information is distributed and often replicated across peers, which can provide cheap reliability and performance. However, less control is possible in P2P architectures, making it harder to ensure consistency, reliability, and performance across all peers.

2. **Security**: Providing security in P2P architectures can be more challenging compared to C/S architectures, as information is distributed across multiple peers. Ensuring data integrity, confidentiality, and authentication can be more complex in a P2P environment where peers may have varying levels of trust and security measures.

3. **Control and Organization**: C/S architectures offer a clear separation of roles between clients and servers, making it simple to understand and manage. Clients only interact with servers, and servers provide services to clients. In P2P architectures, there is less control and organization, as all peers have equal capabilities and responsibilities. Peers can initiate and respond to requests from other peers, which can make managing and coordinating the system more challenging.

4. **Flexibility and Complexity**: P2P architectures can offer greater flexibility and decentralization compared to C/S architectures, as there is no central authority or control. Peers can collaborate, share resources, and communicate directly with each other. However, this can also increase the complexity of managing and coordinating the system. C/S architectures, on the other hand, are generally simpler and well-understood, with a clear separation of roles between clients and servers.

In summary, C/S architectures are typically used when there is a need for clear separation of roles, centralized control, and simplicity, while P2P architectures are preferred when there is a need for decentralization, flexibility, and collaboration among peers. Each architecture has its strengths and weaknesses, and the choice between C/S and P2P depends on the specific requirements and characteristics of the system being designed or implemented.

1.5.6 Hierarchical and Hybrid Models

In some P2P systems, peers are not all the same and can be categorized into different types. Two common approaches are hierarchical and hybrid P2P systems.

Hierarchical P2P Systems
In hierarchical P2P systems, peers are categorized into normal (weak) peers and super-peers. Super-peers are typically more powerful and play a special role in the overlay network. Weak-peers are directly connected to super-peers, forming an overlay network of super-peers. Super-peers can handle tasks such as indexing, searching, or managing resources, while weak-peers primarily perform regular P2P tasks.

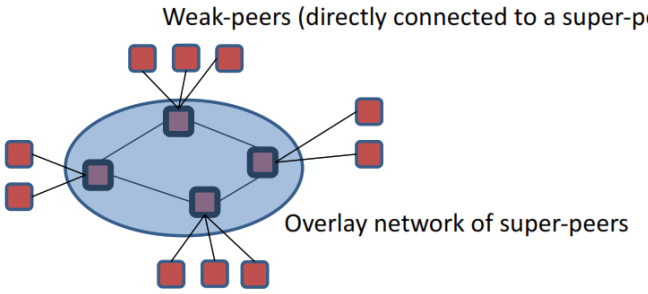

Figure 8: Hierarchical P2P system

Hybrid P2P Systems
Hybrid P2P systems combine elements of both C/S (Client/Server) and P2P architectures together. They may use a combination of client-server and peer-to-peer approaches to achieve their goals. For example, some peers may act as clients, while others act as servers, providing resources or services to other peers. Hybrid P2P systems offer flexibility and scalability by leveraging the advantages of both C/S and P2P architectures.

1.6 DS: Abstractions

In distributed systems *abstractions* provide a higher level of abstraction and encapsulation to hide the complexities of distributed systems, allowing developers to reason about the system's behavior and interactions in a more intuitive and manageable way. Abstractions in distributed systems are used to represent concepts, services, or behaviors that help in simplifying the design, implementation, and management of distributed systems.

In this context, the most commonly used abstractions are[1]:

- Remote Procedure Calls (RPC)
- Message Passing
- Shared Persistent Data

Moreover, in distributed systems, coordination among components or nodes is essential to achieve desired behaviors. Different forms of coordination can be classified based on ***temporal coupling*** and ***referential coupling***.

- **Temporally coupled** coordination involves coordination based on time or timing constraints. Components are coordinated in a way that their actions or events are synchronized in time. For example, a distributed system may require components to execute their tasks in a specific order or at specific time intervals to achieve a coordinated behavior. Temporally coupled coordination can provide precise coordination but may introduce dependencies and potential delays in the system.
- **Temporally decoupled** coordination allows components to operate independently without strict timing constraints. Components can execute tasks or events independently without coordination based on time. This can provide more flexibility and scalability in the system but may require additional mechanisms to ensure coordination and consistency among components.
- **Referentially coupled** coordination involves coordination based on references or pointers. Components share references or pointers to each other's resources or services, and coordination is achieved by referencing shared resources or services. For example, in a distributed system, components may share references to a common database or a shared service for coordination. Referentially coupled coordination can provide efficient and direct coordination but may introduce tight coupling among components.

[1] All of them will be further be analyzed and described in a second moment.

- **Referentially decoupled** coordination allows components to operate independently without sharing references or pointers. Components can operate independently without referencing each other's resources or services directly. Coordination is achieved through other means, such as message passing, events, or shared data. Referentially decoupled coordination can provide loose coupling among components, allowing for more flexibility and scalability, but may require additional mechanisms for coordination and communication.

Different forms of coordination provide trade-offs between precision, flexibility, and scalability in distributed systems. The choice of coordination mechanism depends on the requirements, characteristics, and constraints of the distributed system and its components.

1.6.1 Higher-Level Abstractions

Higher-Level Abstractions are additional layers of abstraction that are built on top of existing middleware, protocols, or APIs to provide more advanced functionality and ease of use in distributed systems. These abstractions are designed to simplify the development and management of distributed applications by encapsulating complex functionalities into higher-level constructs that are easier to understand and use.These abstractions are typically implemented using middleware or protocols such as gRPC, Java Messaging System (JMS), JavaSpaces, and others.
Higher-Level Abstractions can be classified into two main groups: ***RPC-Based*** and ***MOM-Based***. The former includes:

- Distributed Object Systems
- Distributed Services and Web Services
- Distributed Resources

While the latter:

- Publish-Subscribe Systems
- Message Queues

.

Distributed Object Systems
Distributed Object Systems provide abstractions for distributed objects or components that can be invoked remotely across a network, allowing for seamless interaction between distributed components as if they were local objects. This abstraction is often used in object-oriented programming paradigms, where objects can be distributed across multiple nodes in a distributed system. This specific abstraction can be implemented through a direct exchange between Client and Server (using a RPC) or throughout the use of a broker.

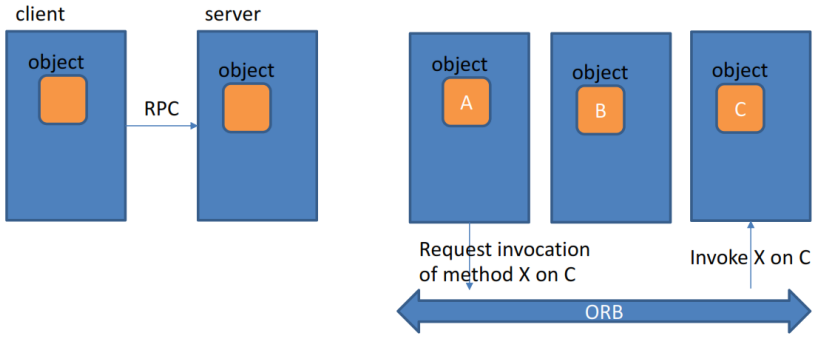

Figure 9: Direct vs broker DOS

Distributed Services
Distributed Services are an abstraction in distributed systems that is similar to distributed objects, but with some key differences. While both distributed objects and distributed services allow for remote

invocation of functionality across a network, distributed services have some unique characteristics that set them apart.
The main differences between distributed services and distributed objects are:

- **Larger Grain:** Services typically have a coarser-grained interface compared to objects. While objects provide fine-grained methods or operations that can be invoked remotely, services often expose larger units of functionality or business processes as their interface. This allows for more coarse-grained interactions between distributed components, which can be more efficient in certain scenarios.
- **Autonomy and Longevity:** Services are designed to be autonomous and long-living entities that can operate independently and persistently over time. Unlike objects, which are typically created and destroyed as needed during the execution of a program, services can be designed to be long-running processes that provide continuous functionality and are available for use by clients at any time.
- **Inter-organizational Availability:** Services may be made available for use by different clients from different organizations. Services can be designed to be exposed over the internet or other wide-area networks, allowing clients from different organizations or domains to access and utilize the functionality provided by the service. This enables inter-organizational collaborations and partnerships, where services can be shared among multiple entities.
- **Service Composition:** Services enable service composition, which refers to the ability to combine and orchestrate multiple services to create more complex and higher-level functionality. Services can be designed to be composable, allowing them to be used as building blocks to create new services or to implement complex business processes that span multiple services. This provides flexibility and extensibility in designing distributed applications.

Examples of distributed services include Service-Oriented Architecture (SOA) and web services. SOA is an architectural approach that uses services as the fundamental building blocks for creating distributed applications, where services are designed to be loosely coupled and interoperable. Web services are a popular implementation of SOA, which use web-based protocols such as SOAP, REST, and WSDL for exposing and invoking services over the internet. Web services provide a widely adopted and standardized way of implementing distributed services in modern distributed systems.

Distributed Resources
Distributed Resources provide abstractions for managing shared resources in a distributed system, such as distributed databases, distributed file systems, or distributed caches. These abstractions enable efficient and transparent access to shared resources across distributed nodes, allowing for seamless coordination and data sharing among distributed components. They are similar to distributed services, and were introduced with the advent of Representational State Transfer (REST) and the Hypertext Transfer Protocol (HTTP). In the context of RESTful architectures, a resource is a data entity with state on which fixed operations *(Create, Read, Update, Delete - CRUD)* are possible.

The main point about distributed resources is the concept of ***Resource***. A resource is a data entity with state that is accessible over a network. Resources can be anything that can be identified by a Uniform Resource Identifier (URI), such as web pages, documents, images, or any other type of data. In RESTful architectures, resources are the fundamental building blocks that are manipulated through standard HTTP methods, such as GET, POST, PUT, and DELETE, to perform CRUD operations.

Distributed resources are an important concept in modern web-based systems and are widely used in RESTful architectures, which have become a popular approach for designing scalable and loosely coupled distributed applications.

Publish-Subscribe Systems
Publish-Subscribe is a messaging paradigm used in distributed systems that achieves referential decoupling between processes or components. In this paradigm, processes do not need to know about each other, and communication is done through the concept of messages or events. Clients can perform two

elementary operations: publish, which involves generating a message or event, and subscribe, which involves expressing interest in a class of messages or events.

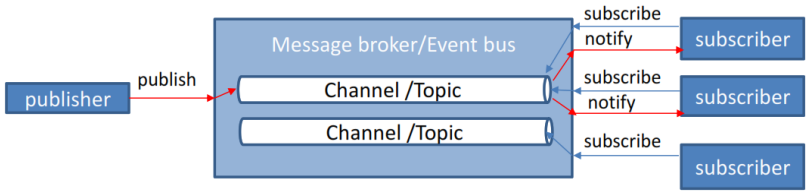

Figure 10: Publish/Subscribe System

The main features of Publish-Subscribe mechanism are:

- **Referential Decoupling**: Publish-Subscribe enables referential decoupling between processes or components in a distributed system. Processes can operate independently and do not need to have direct knowledge or dependencies on each other. This allows for loose coupling and flexibility in the system's design, as processes can evolve independently without affecting each other.
- **Publish Operation**: The publish operation involves generating a message or event by a process or component. This message or event typically contains data or information that needs to be communicated to other processes or components in the system. The published message or event is not sent directly to specific subscribers, but is instead broadcasted to all interested subscribers.
- **Subscribe Operation**: The subscribe operation involves expressing interest in a class of messages or events by a process or component. Subscribers specify their interest in certain types or categories of messages or events, and the system routes relevant messages or events to the subscribed components. This allows components to selectively receive only the messages or events that are relevant to their functionality or interest.
- **Message Broker/Event Bus**: Publish-Subscribe systems typically employ a message broker or event bus, which is responsible for handling the routing and delivery of messages or events between publishers and subscribers. The message broker or event bus takes care of notifying each published message or event to all the up and running subscribers that have expressed interest in the corresponding category of messages or events.
- **Centralized, Decentralized, or Cloud-based**: Publish-Subscribe systems can be implemented in different ways depending on the architecture and requirements of the distributed system. They can be centralized, where a single message broker or event bus handles all the messaging operations, such as in the case of MQTT (Message Queuing Telemetry Transport). They can also be decentralized, where multiple message brokers or event buses are used to distribute the messaging operations, such as in the case of Data Distribution Service (DDS). Finally, they can be cloud-based, where a cloud service provider offers a managed Publish-Subscribe service, such as Amazon SNS (Simple Notification Service) as part of Amazon Web Services (AWS).

Publish-Subscribe is a popular messaging paradigm used in various distributed systems, including Internet of Things (IoT) applications, event-driven architectures, and microservices architectures, as it provides loose coupling, scalability, and flexibility in designing distributed systems.

Message Queue

Message queues are a type of higher-level abstraction in distributed systems that provide a distributed implementation of a queue data structure. In a message queue, clients can enqueue (or queue) messages to be sent to other components or processes, and these messages remain stored in the queue until they are dequeued (or removed) by a recipient component. Message queues enable asynchronous interaction between components, where components can send messages without waiting for immediate responses, allowing for decoupling and scalability in distributed systems.

The basic operations in a message queue are enqueue and dequeue. Clients can enqueue messages by adding them to the end of the queue, and these messages are stored in the queue in the order they are

added. Clients can also dequeue messages by removing them from the front of the queue, and these messages are processed by the recipient component or process in the order they are dequeued.
Message queues provide several benefits in distributed systems:

- **Decouple Components or Processes** that are sending and receiving messages, as they do not need to know each other's details or be active at the same time. This allows for loose coupling between components, enabling them to evolve independently and be replaced or upgraded without impacting other components.
- **Enable asynchronous interaction**, where components can send messages without waiting for immediate responses. This can improve the performance and scalability of distributed systems, as components can continue processing other tasks while messages are stored in the queue, and recipients can process messages at their own pace.
- **Provide fault tolerance and reliability** in distributed systems. Messages can be persisted in the queue until they are successfully dequeued and processed, ensuring that messages are not lost due to failures or crashes in the system. Additionally, message queues can handle varying message rates and bursty traffic, allowing for smooth handling of message processing in distributed systems.

Message queues can be implemented in various ways, including using standalone message queuing systems or as part of middleware or messaging frameworks. Examples of popular message queuing systems include RabbitMQ, Apache Kafka, and ActiveMQ, while messaging frameworks such as Advanced Message Queuing Protocol (AMQP) and Java Message Service (JMS) provide standardized APIs for working with message queues in distributed systems.

1.6.2 Lower-Level Abstractions

Sockets are a type of communication abstraction that operates at a lower-level in distributed systems. They provide a generic, low-level interface for establishing communication channels between processes or systems over a network. Sockets offer elementary communication primitives, such as connect, bind, listen, send, and receive, that can be used to send and receive data across the network.

One key characteristic of sockets is that higher-level protocols must be user-defined on top of them. Sockets provide a basic interface for transmitting data, but the semantics and structure of the data, as well as the communication patterns, need to be defined by the application or system using sockets. This allows for maximum flexibility, as sockets can be used to implement custom communication abstractions that are tailored to the specific requirements of a distributed system.

There are several examples of sockets in use, including TCP/IP sockets, which are the de-facto standard for accessing layer 4-3-2 protocols in the TCP/IP stack. TCP/IP sockets provide reliable, connection-oriented communication channels that are widely used in distributed systems for applications such as file transfer, email, and web browsing.
Another example is WebSockets, which is an interface for managing TCP-based full-duplex channels over a single connection. WebSockets are based on a standard HTTP-compatible protocol and allow for bi-directional communication between a client and a server in real-time. WebSockets are commonly used in web applications for features such as real-time messaging, notifications, and collaborative editing.

In summary, sockets are lower-level communication abstractions in distributed systems that offer a generic, flexible interface for establishing communication channels. Higher-level protocols need to be defined on top of sockets, allowing for custom communication abstractions to be implemented based on specific requirements. Examples of sockets include TCP/IP sockets and WebSockets, which are widely used in various distributed systems for different communication purposes.

2 Web Services and the REST Architectural Style

In this section, we will cover the fundamentals of web services and delve into the concept of REST APIs. We will also explore various distributed architectures and models that describe the level of compliance of a web service with REST constraints.

2.1 SOA: Service Oriented Architecture

Service Oriented Architecture (SOA) is a software design approach where software systems are organized as a set of services. Services are self-contained and autonomous software components that provide specific functionality and are designed to be loosely coupled, meaning they are independent of each other and can evolve independently without affecting the overall system.

Services in SOA are typically provided through interfaces that are published and automatically discoverable. These interfaces are machine-readable, which allows for automated discovery, composition, and invocation of services. Services are described by contracts, which are made up of one or more interfaces. Contracts define the input and output messages, data formats, and communication protocols that are used to interact with services.

A service in SOA is implemented by a single instance that is always available, meaning it can be accessed and invoked by clients at any time. Services are typically coarse-grained, which means they provide relatively large and meaningful units of functionality, as opposed to fine-grained services that provide small and specific operations. Coarse-grained services allow for better encapsulation of functionality and reduce the overhead of communication between services.

One of the key principles of SOA is loose coupling. Services in SOA are designed to be loosely coupled, meaning they are independent of each other and do not have direct dependencies. This allows services to evolve independently, which provides flexibility and scalability to the system. Interactions between services in SOA are typically asynchronous, meaning they are based on message exchanges, where clients send messages to services and receive responses asynchronously.

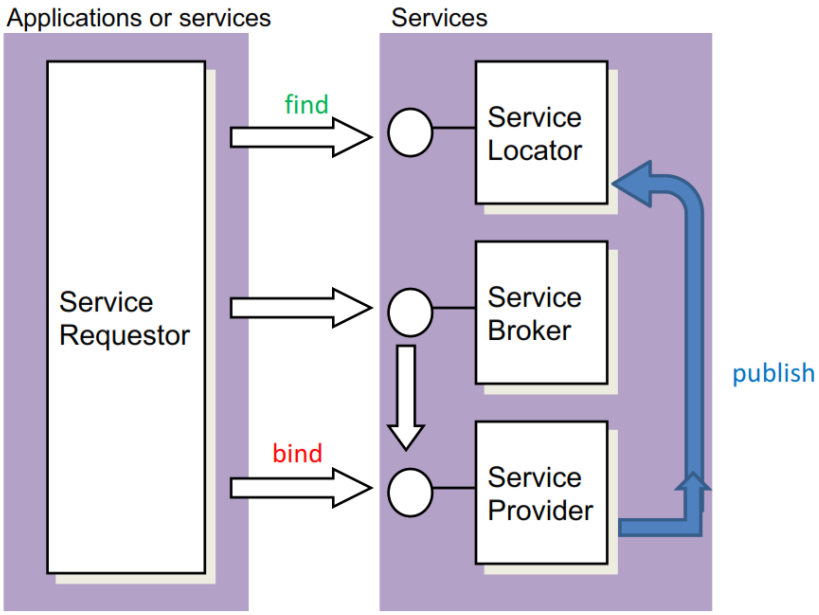

Figure 11: Service Oriented Architecture

In summary, Service Oriented Architecture (SOA) is a software design approach where software systems are organized as a set of services that are provided through interfaces, described by contracts, and implemented by single instances. Services in SOA are typically coarse-grained, loosely coupled, and

interact through asynchronous message exchanges. SOA provides flexibility, scalability, and autonomy to software systems by allowing services to evolve independently.

2.2 Web Services

Web Services are distributed services based on HTTP protocol. They provide a standardized way for software systems to communicate and share data over the internet or an intranet, regardless of the underlying hardware, operating system, or programming language.

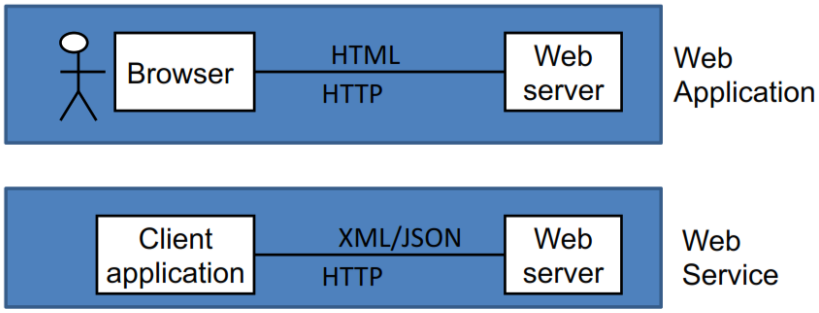

Figure 12: Web Service

Web Services follow a set of standards and protocols, collectively known as the Web Services stack, which includes XML (eXtensible Markup Language), SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language), and UDDI (Universal Description, Discovery, and Integration). These standards define how services are described, discovered, and invoked.

Moreover, they are typically designed to be platform-independent and language-agnostic, which means that can be implemented in any programming language and can be accessed by clients running on different platforms. This makes WSes highly interoperable and allows for seamless integration between different systems and technologies.
Mainly, it is possible to classify them into two main types:

- **SOAP-based Web Services**: These services use SOAP as the messaging protocol for exchanging structured XML data over HTTP or other transport protocols. SOAP provides a standardized way to format messages, define operations, and handle errors in Web Services.

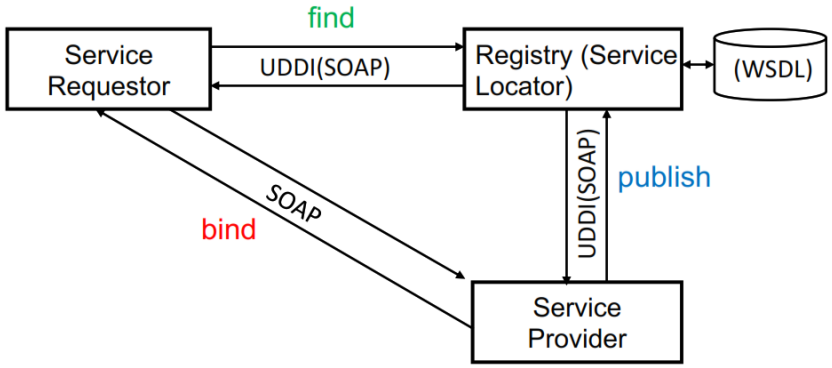

Figure 13: SOAP-Based Web Service

- **RESTful Web Services**: These services use the principles of Representational State Transfer (REST) architecture for designing and implementing distributed services. RESTful Web Services use standard HTTP methods such as GET, POST, PUT, and DELETE to perform operations on resources, which are identified by URLs.

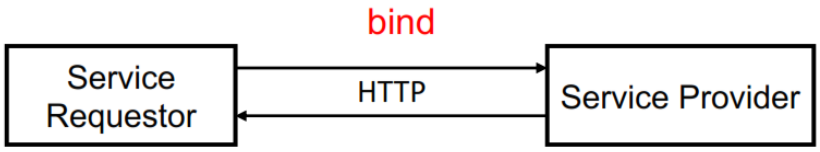

Figure 14: RESTful Web Service

Widely used in various industries, including e-commerce, finance, healthcare, logistics, and more, web services enable integration between different systems, allow for service composition and orchestration, and provide a scalable and flexible approach to building distributed applications.

2.3 REST Architectural Style

***Re**presentational **S**tate **T**ransfer* (REST) is an architectural style for designing distributed systems, introduced by Roy Fielding, who was one of the main designers of the HTTP 1.1 protocol. REST is a set of constraints that define how systems should be structured and how they should communicate over the web or any other network.
The REST architecture is built on a set of constraints that define the behavior of the system. These constraints provide guidelines for designing distributed systems that are scalable, maintainable, and interoperable. The main constraints of REST are:

- **Client/Server**: The system is divided into two separate components - the client and the server. The client is responsible for the user interface and user experience, while the server is responsible for processing requests and managing resources. This **separation of role** concerns allows for independent evolution of the client and server components, keeping them simple, and enabling scalability and flexibility in the system architecture.

- **Stateless**: Each request from a client to a server must contain all the information needed to understand and process the request. The server should not store any information about the client's state between requests. This allows for easy load balancing and scalability, as servers can handle requests from any client without relying on stored state. Moreover, it improves visibility and monitoring (since the server must not monitor all the interactions), but also reliability: indeed since the state is not kept on the server, it simplifies recovery from partial failure.

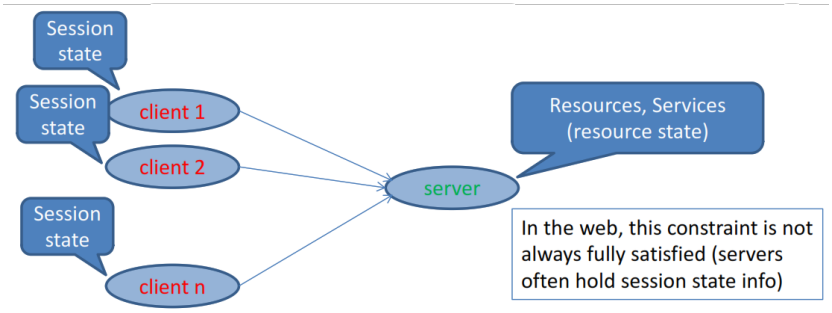

Figure 15: REST Stateless Interaction Protocol

- **Cacheable**: Responses from the server can be cached by the client, by labeling them as cacheable or non-cacheable, improving performance and reducing the load on the server. Caching allows for efficient resource utilization and reduces the need for repeated requests to the server.

- **Layered**: The architecture can be composed of multiple layers, where each layer provides a specific set of functionality. Layers can be added or removed without affecting the overall behavior of the system in a "Vertical way". This allows for flexibility in the system architecture and promotes separation of concerns, de facto it produces a multi-tier architecture.

- **Uniform Interface**: The interface between the client and server is uniform, with a fixed set of well-defined methods that are consistent across all resources. This simplifies the system architecture, makes it easy to understand and use, and allows for evolution and extension of the system without affecting clients.

Overall, REST provides a simple, scalable, and loosely coupled approach for architecting distributed systems, making it a popular choice for building modern web applications, microservices, and other distributed applications.

2.4 Resources

In the REST architecture, the uniform interface constraint requires that server interfaces are resource-oriented. The key concept in this constraint is the resource, which is an information item with time-varying state. A resource can be anything that can be identified by a URI (Uniform Resource Identifier), such as a document, an image, a user profile, or any other entity with a unique identity.
Resources in RESTful systems can have zero or more representations, which are different representations of the same resource in different media types such as XML, JSON, HTML, etc. Representations are used to transfer the state of a resource between the client and server. For example, a client may request a resource in JSON format and the server may respond with the state of that resource serialized as JSON.

In REST, when a client requests an operation on a resource, a representation of the resource, which can be its current, past, or future state, may be transferred between the client and server. However, the actual resources themselves always stay on the server side, and their URIs serve as pointers to access them.

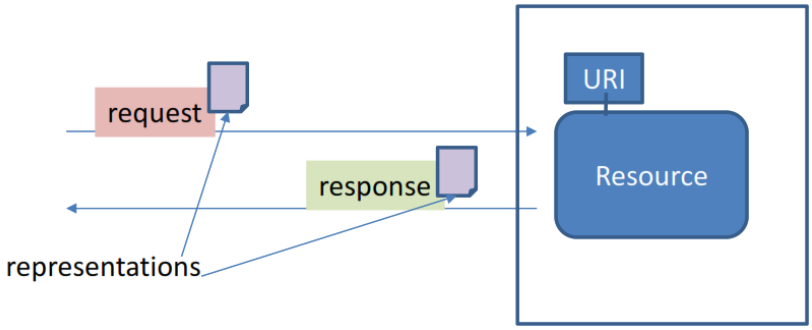

Figure 16: A Resource request and response

HTTP Protocol offers a set of fixed and predefined interfaces for acting on resources, the requested operation has the following semantic for HTTP:

- **Requested Method**
- **Request Body**
- **Request Headers** (Control Metadata)
- **Target URL** (Including also query strings)

The result of the operation is communicated in the response via: a *status code*, a *response body* and other *response headers*. Basically, this kind of uniform interface offers the CRUD operations defined by REST, following is provided a schema describing the correspondence:

CRUD Operation	HTTP request	HTTP response (if no error)
Create a new resource under *res* (or send data to *res* for resource-specific processing)	POST *res* (resource or data representation in the body)	URL of new resource created and/or result of processing (in body)
Read resource *res*	GET *res* (no body)	representation of current state of *res* (in the body)
Update resource *res* by replacing its state with a new one (*res* can be created)	PUT *res* (new resource representation in the body)	description of executed operation (200 or 201 or 204 status code)
Delete resource *res*	DELETE *res*	

Figure 17: CRUD Operations in REST Domain

Some other methods which provide more specific operations are defined:

Operation	HTTP request	HTTP response (if no error)
Read information about resource *res* without transferring its representation	HEAD *res* (no body)	same as for GET but without body (only headers sent)
Read supported methods for a resource	OPTIONS *res* (no body)	list of supported methods (in the Allow header and possibly in the body)
Test the connection to resource *res* (loopback testing similar to echo)	TRACE *res*	the received request (in the body)
Partially Update the resource *res* by applying a patch	PATCH *res* (patch to be applied to *res* in the body)	description of executed operation (200 or 201 or 204 status code)

Figure 18: Other Operations in REST Domain

For all the provided methods it is possible to evaluate of which properties they possess, specifically in terms of: *Idempotence, Safeness, Cacheable Response, Request Body Presence.* Here is a detailed schema:

HTTP method	CRUD operations	idempotent	Safe	Request body	Cacheable response
POST	C	no	no	yes	(no)
GET	R	yes	yes	no	yes
PUT	U, C	yes	no	yes	no
DELETE	D	yes	no	no	no
HEAD	R	yes	yes	no	yes
OPTIONS	R	yes	yes	no	no
TRACE	-	yes	yes	yes	no
PATCH	U, C	no	no	yes	no

Figure 19: HTTP Methods and properties

For a more detailed comprehension of REST operations over resources, let's provide an example: consider `courses` a resource that represents a collection of courses, while `courses/id` a resource representing a single course:

Method URI pair	Meaning
POST courses	Create a new course with given representation, and add it to the collection as a new resource under courses. Return its URI
GET courses	Read the whole collection of courses
GET courses/id	Read only course id in the collection
GET courses?year=1	Read only the courses of the first year in the collection
PUT courses	Update the whole collection with the given representation
PUT courses/id	Update only course id in the collection with the given representation
DELETE courses	Delete the whole collection of courses
DELETE courses/id	Delete only course id in the collection

Figure 20: Resources Operations: Example

2.5 HATEOAS (Hypermedia as the Engine of Application State)

HATEOAS stands for *Hypermedia as the Engine of Application State*. It is a constraint of the REST *(Representational State Transfer)* architectural style that governs the design and interaction of distributed systems, particularly web services.

In a distributed system that follows the HATEOAS constraint, hypermedia links are used to dynamically represent the available actions or state transitions that a client can perform on a resource. In other words, the server provides links in the response that indicate what actions are allowed on the resource, and the client can follow these links to navigate and interact with the system.
HATEOAS promotes a self-descriptive and discoverable approach to APIs, where the client does not need to have prior knowledge of the endpoints or actions, but can dynamically discover and navigate the API based on the hypermedia links provided by the server. This allows for flexible and evolvable systems, where changes to the server-side resources or actions can be made without requiring changes to the clients.

A common example of HATEOAS in practice is in the context of web APIs, where the server may provide links in the response as hyperlinks or URLs, and the client can follow these links to perform actions like retrieving related resources, creating new resources, updating resources, or deleting resources.

HATEOAS can provide benefits such as decoupling of client and server, flexibility in API evolution, and improved discoverability and navigation of APIs. However, it can also add complexity to API design and client implementation, as clients need to understand and interpret the hypermedia links provided by the server. Proper documentation and adherence to HATEOAS design principles are important for successful implementation of this constraint in distributed systems.

2.6 Richardson Maturity Model

In this section, we will discuss the Richardson Maturity Model, which provides a framework for assessing the level of RESTfulness in web services or APIs. The model, proposed by Leonard Richardson, consists of four levels that describe the evolution of RESTful architectures:

- **Level 0: The Swamp of POX (Plain Old XML)**
  At this level, the API primarily uses HTTP as a transport mechanism for XML payloads. There is

no clear separation of concerns between resources and their representations, and HTTP methods are not used for operations on resources. This level is not considered fully RESTful as it lacks key principles such as resource identification through URIs and the use of HTTP methods for operations.

- **Level 1: Resources**
  At this level, the API starts to use HTTP methods to perform CRUD (Create, Read, Update, Delete) operations on resources. Each resource is identified by a URI, but the API still primarily uses payloads, typically in XML format, for data exchange. While this level introduces resource identification and basic use of HTTP methods, it still lacks some key RESTful principles such as the use of hypermedia and self-descriptive messages.

- **Level 2: HTTP Verbs**
  At this level, the API fully embraces the use of HTTP methods for operations on resources, according to their semantics. HTTP methods such as GET, POST, PUT, and DELETE are used to perform operations on resources, and HTTP status codes are used to indicate the outcome of requests. The API responses may be in various formats such as XML, JSON, or others. This level is more RESTful as it adheres to the principles of resource identification, stateless client-server communication, and the use of standard HTTP methods.

- **Level 3: Hypermedia Controls**
  At this highest level, the API uses hypermedia controls, typically in the form of hyperlinks, to drive the state transitions of a client application. Clients can dynamically discover and interact with resources through hypermedia controls, making the API more self-descriptive and decoupled. This level fully embraces the HATEOAS (Hypermedia As The Engine Of Application State) principle, making the API highly RESTful.

The Richardson Maturity Model is often used as a guideline for designing RESTful web services or APIs that adhere to the principles of Representational State Transfer (REST), which is an architectural style for designing networked applications.

3 Abstract Syntaxes and Schemas

In this section, we will discuss the practical approach to designing REST APIs through the use of abstract syntaxes and schemas. One of the main challenges in designing RESTful systems is ensuring data transparency, particularly when dealing with heterogeneous systems that may have different protocols and data formats.
Marshaling, also known as serialization, refers to the process of converting data from an internal representation to a format suitable for transmission over a network. Unmarshaling, or deserialization, is the reverse process of converting received data back into an internal representation. These processes are critical in RESTful APIs as they enable the communication and exchange of data between different systems with potentially varying data representations.
The use of abstract syntaxes and schemas provides a practical way to define the structure and format of data in REST APIs, ensuring consistency and interoperability between systems. Abstract syntaxes define the structure and semantics of data in a language-independent manner, while schemas provide a way to specify the concrete syntax and validation rules for data in a specific format or encoding, such as JSON or XML. By defining abstract syntaxes and schemas, REST APIs can achieve data transparency and interoperability, allowing systems with different protocols and data formats to effectively communicate and exchange data.

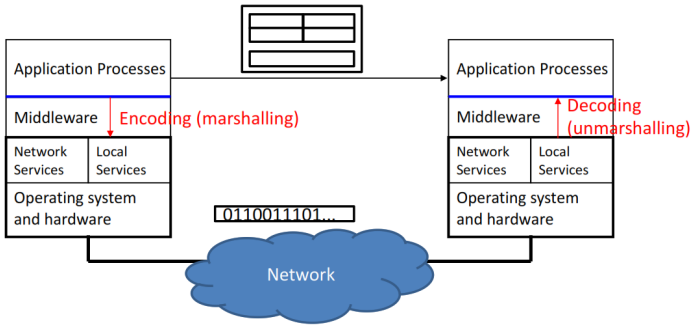

Figure 21: Marshaling and Unmarshaling of Data

Applications typically rely on standard ways for system-independent data representation, which may include binary solutions *(such as ASN.1, XDR, Protocol Buffers, etc.)* or character-oriented solutions *(such as XML, JSON, YAML, etc.)*.
These standards generally consist of:

1. A ***language*** that defines abstract data types, also known as abstract syntax.
2. "*Neutral*" **data representations** that are not tied to any specific system, and can be used to represent any abstract data type defined by the language.

A data representation mechanism is required for data receivers to properly separate, validate, and decode the data. Depending on the scenario, the receiver may need to have prior knowledge of the data type, or it may infer the data type from the data itself.

3.1 The JSON Schema Language

In this section we will give a look at how it is possible to specify JSON schemas and use them to validate JSON messages. As we use a JSON document to describe a JSON data type, this description is machine readable and serializable; in this way it is easy to send this data over the network. Data types that can be described with this language correspond to types comonly available in other programming languages *(e.g. Javascript)*.
Following is provided an example:

```json
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Example Schema",
  "description": "This is an example JSON schema.",
  "type": "object",
  "properties": {
    "property1": {
      "type": "string",
      "description": "This is the description for property1."
    },
    "property2": {
      "type": "integer",
      "description": "This is the description for property2."
    }
  }
}
```

Listing 1: Example of a JSON Schema

The JSON schema has the following structure:

- **Metadata Section:**
  - `$schema`: Specifies the version of the JSON schema being used.
  - `title`: Provides a title for the schema.
  - `description`: Provides a description for the schema.
- **Type Description:**
  - `type`: Specifies the type of the JSON object. In this example, it's set to "object" to define an object type.
  - `properties`: Defines the properties of the JSON object, each with its own type and description.

3.1.1 JSON Schema Data Types

The JSON schema supports the following data types:

- **string:**
  - `type`: Specifies the data type as "string".
  - `format`: Specifies a format for the string (e.g., "date", "email", "uri").
  - `pattern`: Specifies a regular expression pattern that the string must match.
  - `minLength`: Specifies the minimum length of the string.
  - `maxLength`: Specifies the maximum length of the string.
  - `enum`: Specifies a list of allowed string values.
- **number:**
  - `type`: Specifies the data type as "number".
  - `format`: Specifies a format for the number (e.g., "float", "double").
  - `minimum`: Specifies the minimum allowed value of the number.
  - `maximum`: Specifies the maximum allowed value of the number.
  - `exclusiveMinimum`: Specifies if the minimum value is exclusive (i.e., not allowed).

  - `exclusiveMaximum`: Specifies if the maximum value is exclusive (i.e., not allowed).
  - `multipleOf`: Specifies a multiple that the number must be divisible by.

- **integer:**
  - `type`: Specifies the data type as "integer".
  - `format`: Specifies a format for the integer (e.g., "int32", "int64").
  - `minimum`: Specifies the minimum allowed value of the integer.
  - `maximum`: Specifies the maximum allowed value of the integer.
  - `exclusiveMinimum`: Specifies if the minimum value is exclusive (i.e., not allowed).
  - `exclusiveMaximum`: Specifies if the maximum value is exclusive (i.e., not allowed).
  - `multipleOf`: Specifies a multiple that the integer must be divisible by.

- **boolean:**
  - `type`: Specifies the data type as "boolean".

- **object:**
  - `type`: Specifies the data type as "object".
  - `properties`: Specifies the properties of the object, each with its own schema.
  - `additionalProperties`: Specifies if additional properties are allowed and their schema.
  - `required`: Specifies a list of required property names.
  - `minProperties`: Specifies the minimum number of properties allowed.
  - `maxProperties`: Specifies the maximum number of properties allowed.
  - `dependencies`: Specifies the dependencies between properties[2].
  - `enum`: Specifies the allowed values by enumeration (using JSON syntax).

- **array:**
  - `type`: Specifies the data type as "array".
  - `items`: Specifies the schema for the items in the array.
  - `additionalItems`: Specifies if additional items are allowed and their schema.
  - `minItems`: Specifies the minimum number of items allowed in the array.
  - `maxItems`: Specifies the maximum number of items allowed in the array.
  - `uniqueItems`: Specifies if all items in the array must be unique.

- **null:**
  - `type`: Specifies the data type as "null".

Note: The "type" property is common to all data types and specifies the data type of the value. Additional properties and constraints may vary depending on the data type.

[2]Dependencies in JSON schema can be defined into two ways:

* **Property Dependencies**: This type of dependency specifies that a property depends on the presence or absence of another property in the same object. It can be defined using the "dependencies" keyword with the property name as the key and an array of dependent property names as the value.
* **Schema Dependencies**: This type of dependency specifies that the schema of a property depends on the presence or absence of another property in the same object. It can be defined using the "dependencies" keyword with the property name as the key and a JSON schema as the value.

3.1.2 Schema Combination

A JSON Schema can be the result of a combination of schemas; it is possible to combinate them using one among these attributes:

- `allOf`: it requires validity of all the sub-schemas.
- `anyOf`: require validity against at least one sub-schema.
- `oneOf`: require validity against exactly one sub-schema.
- `not`: require not validity.

Alternatively, there can be also a schema combination based on the `if, then, else` or by referring to another schema using `$ref`, this reference can be of three types:

- Internal Reference: referencing a portion of the schema. In this case, it is possible to do it by *"defining"* a reusable piece of schema with the "`definitions`" property. Here is provided an example:

```
{
    "definitions": {
        "address": {
            "type": "object",
            "properties": {
                "street": { "type": "string" },
                "city": { "type": "string" },
                "country": { "type": "string" }
            }
        }
    },
    "type": "object",
    "properties": {
        "orderDate": { "type": "string", "format": "date"},
        "quantity": { "type": "integer" },
        "shippingAddress": { "$ref": "#/definitions/address" },
        "billingAddress": { "$ref": "#/definitions/address" },
        ...
    }
}
```

Listing 2: Example of a `$ref` using an internal schema

- External Reference: referencing an external schema in the local file system.
- External Reference: referencing an external schema available on the network.

4 Service Interface Design

In the software life-cycle, Interface Design is a crucial part of the software design phase. It involves defining the architecture, including modules and their interfaces. During the design phase, various problems need to be addressed, such as determining how to partition the system into modules, deciding on the granularity of modules, designing interfaces for communication between modules, and planning the deployment of modules in distributed software systems.
Software design is a creative activity that requires careful consideration of various factors. However, there are fundamental principles that have been identified to guide the design process. One such principle is **Information Hiding**, which emphasizes a clear separation between interfaces and implementations. This helps in creating modular and maintainable software systems, where the interfaces provide a well-defined abstraction for communication between modules, while the implementation details remain hidden.

4.1 Partitioning Principles

In software design, partitioning principles are used to divide a system or software application into smaller, manageable components or modules. These principles aim to improve the overall design, maintainability, and performance of the system. Here's a brief overview of each of the partitioning principles:

- **Maximize internal cohesion**: Modules within a system should be designed to have high internal cohesion, meaning that the elements within each module are closely related and work together towards a common purpose. This creates modular and self-contained components that are easier to understand, maintain, and update.
- **Minimize coupling**: Modules should be designed to have low coupling, meaning that they should have minimal dependencies on other modules. This creates loosely-coupled components that are more independent and can be modified or replaced without affecting the entire system.
- **Foresee and favor future changes and reuse**: Modules should be designed in a way that anticipates and accommodates future changes and promotes reusability. This involves considering potential changes or modifications that may be required in the future, and designing modules in a flexible and modular way that allows for easy modification, extension, or reuse of components. This helps in creating adaptable and scalable systems that can evolve over time with changing requirements.
- **Balance the load of distributed subsystems**: In distributed systems or environments where components are distributed across multiple hosts or nodes, the workload or tasks should be evenly distributed among the various distributed subsystems. This helps in optimizing the performance and scalability of the distributed system by ensuring that no single subsystem is overwhelmed with an excessive workload, while other subsystems remain underutilized.

By following these partitioning principles, software designers can create well-organized, modular, and maintainable systems that are adaptable to future changes and promote reusability. These principles contribute to the overall quality and efficiency of software design, leading to robust and scalable software systems.

4.2 Choosing the Granularity Level

In software design, determining the appropriate granularity level for partitioning a system into modules is crucial for achieving a well-designed and efficient system. Here are some key considerations to keep in mind:

- **Finding the right trade-off**: Partitioning a system into modules requires striking the right balance between granularity and cohesion. If the granularity level is too fine, meaning that the system is divided into very small modules, it can negatively impact the system's performance and coupling. Fine-grained partitioning can lead to overhead in terms of communication and coordination between modules, resulting in decreased performance. Additionally, fine-grained

modules may be tightly coupled, meaning that changes in one module may have a ripple effect on other modules, increasing the coupling between modules and making the system harder to maintain and update.

- **Avoiding too coarse grain partitioning**: On the other hand, if the granularity level is too coarse, meaning that the system is divided into large, monolithic modules, it can negatively impact the system's internal cohesion, flexibility, and reuse. Coarse-grained partitioning can result in modules that are too large and complex, making it harder to understand and maintain the internal logic and functionality of each module. It can also limit the flexibility and adaptability of the system to changes, as making modifications or updates may require modifying large portions of the system. Furthermore, coarse-grained modules may not be easily reusable in other contexts or projects, reducing the potential for code reuse and efficiency.

In summary, choosing the right granularity level for partitioning a system into modules is a critical decision in software design. It requires finding the right trade-off between fine-grained and coarse-grained partitioning, considering factors such as performance, coupling, internal cohesion, flexibility, and reuse. By carefully considering these factors, software designers can create a well-organized, modular, and maintainable system that strikes the right balance between granularity and cohesion, leading to an efficient and scalable software system.

4.3 Interface Design: Best Practice

As already presented, interface design is a critical aspect of software design that involves creating a formal description of interfaces for modules or components in a system. The following best practices should be considered:

- **Coherence with partitioning principles**: Interfaces should be designed in a way that aligns with the partitioning principles of the system. This includes exposing only the necessary information needed by the service client, following the principle of information hiding. Interfaces should also have high internal cohesion, meaning that the functionality offered by the interface should be closely related and limit the number of interactions required to access its services, thus minimizing coupling between components. If the interface is object-oriented, inheritance should be adopted to promote reusability.

- **Foreseeing particular cases**: Interfaces should account for all possible particular cases, including exceptions. This practice helps in debugging and contributes to constructing a robust and secure system. By anticipating potential exceptions and handling them gracefully, the system can be better prepared to handle unexpected situations.

- **Prefer idempotent methods**: Idempotent methods are those that produce the same result regardless of how many times they are executed. Designing interfaces with idempotent methods can help in limiting uncertainty on the results of distributed calls, improving system reliability and consistency.

- **Consideration of performance**: Interface design should take into account performance considerations. Limiting the number of interactions between components can improve performance, but the size of messages transmitted over the interface should also be considered. Fragmentation, buffering, and processing time can adversely affect performance, so interfaces should be designed with appropriate consideration of these factors.

- **Selective data transmission**: Interfaces should allow clients to select what data to send/receive and how much, in order to avoid transmitting unnecessary data. This can help in reducing network overhead, improving efficiency, and minimizing the impact of unnecessary data transmission on system performance.

In a nutshell, following best practices of interface design presented above, can result in well-designed interfaces that promote reusability, robustness, and efficiency in the software system.

4.4 Interface Design: Approaches

Here will be presented the best approaches to perform interface design. In order to gain a deeper understanding of each approach, we will examine the same example using the approach being considered. Let's imagine to have a service that facilitates the execution of operations on a **bank account**. This bank account is identified by an **account ID** and can be subjected to three operations: *reading* information about the account, *depositing* a specified amount into the account, and *withdrawing* a specified amount from the account. Additionally, the service allows for specification of whether the requested withdrawal amount can be reduced if the full amount is not available in the account. Both the deposit and withdrawal operations require a description, which is a string that can provide additional information or context.

4.4.1 Method centric

In the method-centric approach, the design of interfaces is focused on operations and their input/output arguments. It follows a classical programming language-like approach, where a fixed endpoint is used. This approach is commonly used in traditional procedural or object-oriented programming paradigms. The example described above, can be easily formalized in the following way, according the method-centric approach:

```
public interface AccountReader {
    public Account getAccountInfo(String id)
        throws UnknownAccountIdException;
}

public interface ImprovedAccountUpdater {
    public void addDeposit(
        String id,
        float amount,
        String description,
        String transId)
        throws ReplicatedTransIdException,
        UnknownAccountIdException;

    public float addWithdrawal(
        String id,
        float requestedAmount,
        boolean reducible,
        String description,
        String transId)
        throws ReplicatedTransIdException,
        UnknownAccountIdException,
        NoAvailabilityException;
}
```

4.4.2 Message centric

In the message-centric approach, the design of interfaces is centered around messages and endpoints. A fixed, single-operation interface is used, such as "`send(Message)`", where the design is focused on the exchange of messages between components or services. This approach is commonly used in messaging or event-driven architectures, where components communicate asynchronously through messages.

4.4.3 Constrained

The constrained approach is a middle ground between the method-centric and message-centric approaches. It involves a fixed, multiple-operation interface, and is commonly used in architectural styles like *Representational State Transfer (REST)*. In this approach, the design is focused on endpoints, allocation of operations, and their input/output arguments, while still maintaining some level

of constraint on the operations that can be performed. This approach provides a balance between flexibility and structure, allowing for constrained interactions between components or services.

4.5 REST-specific Issues/Best Practices

Designing a good RESTful API goes beyond just considering the general principles and adhering to the REST constraints. There are other important issues to be considered to ensure an effective RESTful API design.

- **Designing/Organizing Resources:** Careful consideration should be given to how resources are designed and organized in the RESTful API. Resources should be identified and represented in a logical and intuitive manner, following a consistent naming convention and URL structure. Resource endpoints should be designed to be simple and easily understandable, and should represent the state of the resource.
- **Mapping Conceptual Operations to Methods:** The mapping of conceptual operations to HTTP methods (e.g., GET, POST, PUT, DELETE) should be done thoughtfully. The operations that can be performed on a resource should be aligned with the standard HTTP methods, and the use of these methods should adhere to their intended semantics. This ensures that the API is intuitive, easy to use, and follows RESTful principles.
- **Documentation and API Self-description:** Proper documentation and self-description capabilities are crucial for a RESTful API. Documentation should be comprehensive, clear, and up-to-date, providing information on how to use the API, its available resources, endpoints, and operations. API self-description capabilities, such as Hypermedia controls, can be used to provide information about the API's capabilities, available actions, and links to related resources.
- **Follow Design Patterns:** There are established design patterns and best practices for RESTful API design, such as the HATEOAS (Hypermedia as the Engine of Application State) pattern, which allows clients to navigate the API and discover available resources and actions dynamically. Following these design patterns can improve the maintainability, scalability, and evolvability of the API.
- **Verify Compliance with REST:** Not every API labeled as "REST API" may fully adhere to RESTful principles. It's important to verify and ensure that the API design strictly follows the principles and constraints of REST architecture to achieve the desired benefits of a RESTful API.

4.6 Mapping Resources to URIs

When mapping resources to URIs in API design, conventions can be used to make the API easy to understand and self-describing. For example, using **plural names for collections** and **singular names for non-collections** can help maintain consistency. Additionally, using **URLs that reflect the hierarchical relationships** between resources can enhance the intuitiveness of the API.

However, according to the principle of HATEOAS (Hypermedia as the Engine of Application State), URLs should be "**opaque**", meaning that clients should not rely on the structure of URLs for navigation, but instead should follow links provided in responses to discover and interact with resources. This approach allows for more flexibility in evolving the API without breaking client implementations that rely on specific URL patterns.

There is ongoing debate and differing opinions in the API design community about whether URLs should be opaque or follow conventions. Some argue that using conventions in URLs can aid in API discoverability and ease of use, while others advocate for the strict adherence to the HATEOAS principle and using opaque URLs to promote flexibility and decoupling between clients and servers. Ultimately, the choice between using conventions or opaque URLs in API design depends on the specific requirements and goals of the API and the preferences of the design team.

Once resources have been mapped, it is possible to map also available operations over them. When defining operations in a service, the following steps are typically followed:

- Identify the resource(s) and HTTP method(s) to be used for each operation.
- Check for matches between resources and methods. If no match is found, it may indicate that some resources are missing or need to be added.

Moreover, for each allowed resource/method pair, the following details are defined:

- How the method should be invoked, including accepted query parameters, request headers, and body contents.
- The possible results, including status codes that may be returned, and the corresponding response headers and body contents for each status code.

This mapping of operations to resources, methods, and their associated details helps in designing a well-structured and self-describing API, making it easier for clients to understand how to interact with the service. Following is provided an example considering a bibliography schema and this kind of request:

"Search items by keyword, type (article/book) and publication year"

Resource	Verb	Query params	Status		Resp. Body
biblio/items	GET	keyword: string type: "article"|"book" beforeInclusive:date afterInclusive:date	200	OK	filtered items (items)
biblio/items/{id}	GET		200	OK	item
			404	Not found	

Figure 22: Example of Mapping Operations

It is also possible to have and map other kind of operations, not only *READ* ones:

Resource	Verb	Req. body	Status		Resp. body
biblio/items	POST	item	201	Created	item
			400	Bad Request	string
biblio/items/{id}	PUT	item	204	No Content	
			400	Bad Request	string
			404	Not Found	
biblio/items/{id}	DELETE		204	No Content	
			404	Not Found	

Figure 23: Example of further Mapping Operations

4.7 Efficiency Considerations

Other than simply mapping requests, it is also important to make considerations about what can be done and which are the best practices to increase performances when requesting and responding. More precisely, we should consider the size of responses that may be returned by a service. Here are some common best practices for avoiding large messages:

- **Query parameters**: Instead of returning all data in a single response, allow users to specify query parameters to retrieve only the data they need. This can include filtering, sorting, or limiting the number of results.
- **Paging**: When dealing with large collections of data, consider using pagination to retrieve data in smaller chunks rather than returning all data at once. This can help reduce the size of the response and improve performance.
- **Using references**: Instead of including full representations of related resources in the response, use references or links to indicate the location of the related resources. Clients can then retrieve the full representation of the related resources only when needed, reducing the size of the initial response.

Moreover, on requests side, to help optimizing network bandwidth and reduce unnecessary processing on the server side, it is possible to exploit the HTTP conditional requests. They allow clients to specify that an operation should be executed only under certain conditions. Here are some common headers used for conditional requests:

- **If-Modified-Since**: This header specifies the date and time of the last modification of the resource. The server will only execute the requested operation if the resource has been modified since the value specified in this header.
- **If-Unmodified-Since**: This header specifies the date and time of the last modification of the resource. The server will only execute the requested operation if the resource has not been modified since the value specified in this header.
- **If-Match**: This header specifies the entity tag (ETag) of the resource. The server will only execute the requested operation if the ETag of the resource matches one of the values listed in this header.
- **If-None-Match**: This header specifies the entity tag (ETag) of the resource. The server will only execute the requested operation if the ETag of the resource does not match any of the values listed in this header.

Regarding conditional-requests, following are presented three important use-cases:

- **Management of Client Cache**: it is possible to re-send the resource (a representation of it) only if modified since last time it was cached.

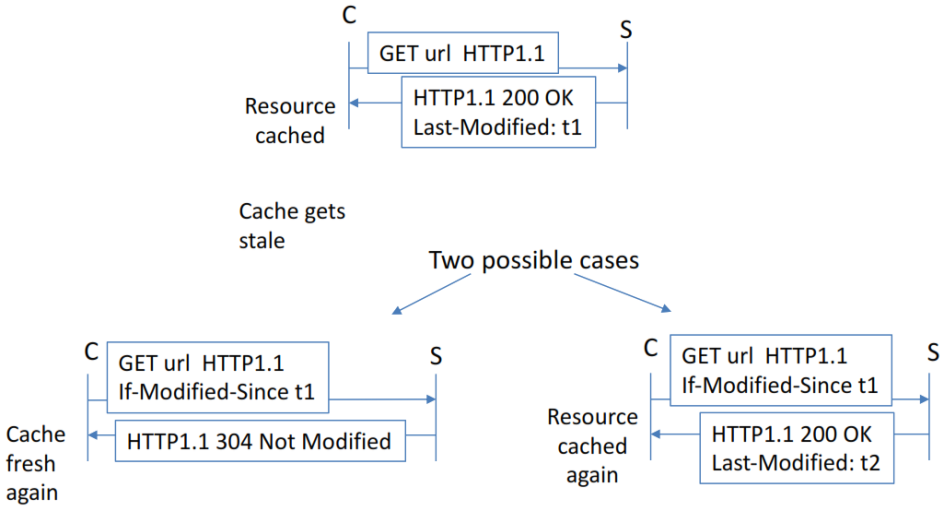

Figure 24: Conditional Requests: Cache Use-Case

- **Lost Update Race Condition**: when multiple clients are working on the same resource (updating it), it might happen that one of the two updates get lost since the other client has not re-read the resource.

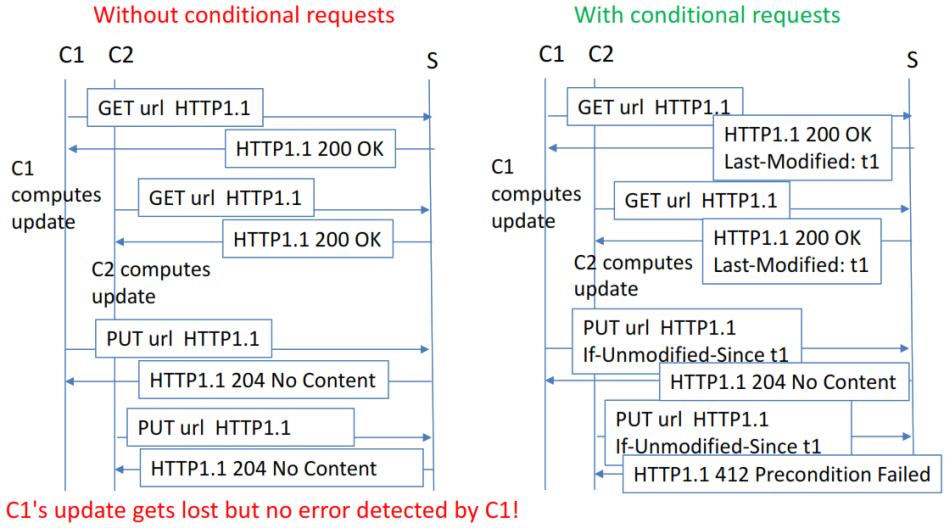

Figure 25: Conditional Requests: Read-update Race Condition

- **Put (Creation) Race Condition**: when the put method is used by multiple clients to create or update a resource simultaneously, it might happen that a first client would create the resource and right after the second one would overwrite it, without any pre-condition.

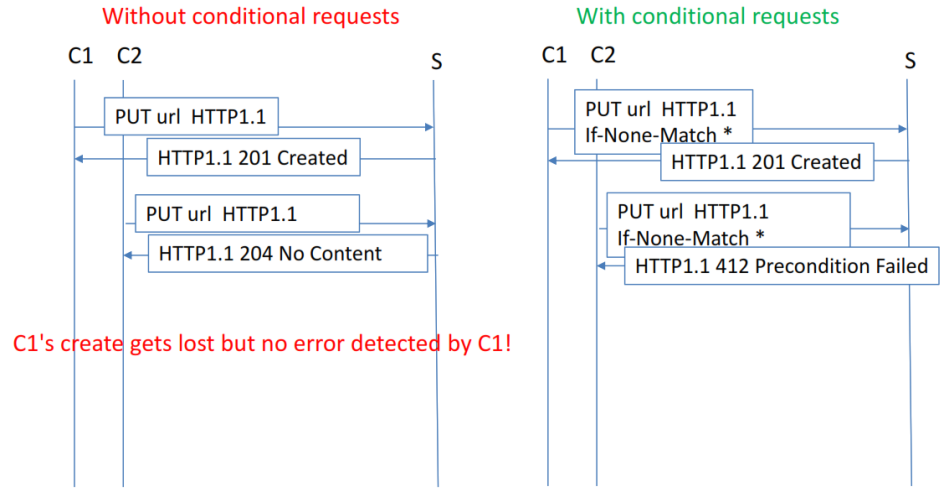

Figure 26: Conditional Requests: Put(Create) Race Condition

5 Remote Procedure Call (RPC)

Remote Procedure Call (RPC) is a communication protocol that allows a computer program to execute a procedure or function in a different address space, typically on a remote computer or server, as if it were a local procedure call. It enables a distributed system to invoke functions or methods across different processes or machines, providing a way for programs to interact and communicate with each other over a network.
The concept of RPC is based on the idea of abstracting the remote procedure invocation process to make it transparent to the calling program. From the perspective of the client program, invoking a remote procedure using RPC appears similar to calling a local procedure or function. The client program does not need to be aware of the underlying network communication details or the location of the remote procedure. Basically, it acts as a Client-Server architecture, in which the client asks for an operation while the server is implementing it.

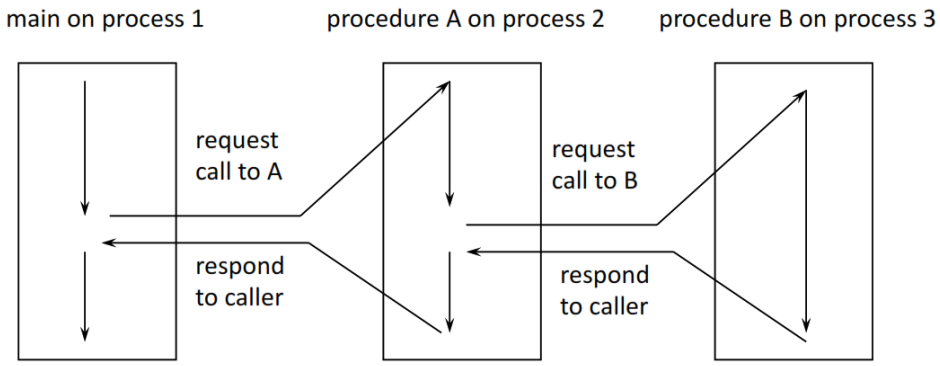

Figure 27: RPC: Remote Procedure Call

Here's a high-level overview of how RPC works:

1. **Client Invocation**: The client program makes a procedure call as if it were calling a local procedure. The parameters of the procedure call are marshaled (serialized) into a suitable format for transmission over the network.

2. **Client Stub**: The client-side RPC runtime system intercepts the procedure call and marshals the parameters into a message. It then sends the message over the network to the remote server.

3. **Network Communication**: The client-side RPC runtime system uses a network protocol, such as TCP/IP, to transmit the message containing the procedure call to the remote server.

4. **Server Stub***(also called skeleton)*: The server-side RPC runtime system receives the message from the network, extracts the procedure call information, and invokes the corresponding server procedure or function.

5. **Server Execution**: The server procedure or function executes the requested operation using the provided parameters.

6. **Server Response**: After executing the procedure, the server returns the results (if any) back to the client.

7. **Network Communication**: The server-side RPC runtime system marshals the results into a message and sends it back to the client over the network.

8. **Client Stub**: The client-side RPC runtime system receives the message from the server, extracts the results, and returns them to the original caller.

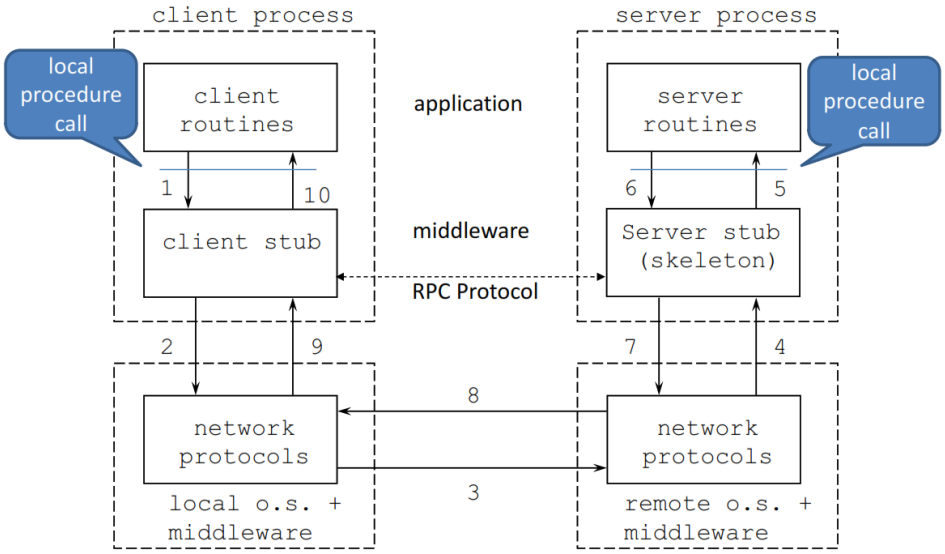

Figure 28: RPC: Remote Procedure Call Model

The key aspect of RPC is the transparency it provides to the calling program. The client and server programs can be written in different programming languages and can run on different operating systems or hardware architectures. RPC abstracts away the complexities of network communication, serialization/deserialization, and protocol handling, allowing developers to focus on the functional aspects of the distributed application. Of course, we must take into account that **full distribution transparency cannot be achieved** *(see subsection 1.2.1)*, this is easy to understand considering that execution time is not the same as in a system with a local procedure and that moreover, it might happen that the client must take an action without having received a response (failure), in this sense, a local procedure cannot fail.
RPC is widely used in distributed systems, client-server architectures, and web services to enable seamless communication and interaction between different components or services. It simplifies the development of distributed applications by providing a familiar programming model and hiding the low-level networking details. Popular RPC frameworks and technologies include gRPC, Apache Thrift, CORBA, and Java RMI (Remote Method Invocation).

5.1 The RPC Middleware

There are two ways of implementing RPC:

1. **Language-based support**: In this approach, RPC is offered as a feature of a programming language, such as Java RMI (Remote Method Invocation). It takes advantage of language-specific serialization techniques to facilitate the communication between client and server. This approach provides high transparency, meaning that the remote procedure invocation appears similar to a local procedure call. However, it is often limited to applications written in a single language.

2. **Stub Generation**: In this approach, the RPC middleware generates stubs based on an interface definition language (IDL). The interface definition describes the methods and their signatures that can be remotely invoked. From the IDL, the RPC middleware generates client stubs and server stubs. These stubs act as proxies for the client and server, respectively, and handle the marshaling and unmarshaling of procedure arguments and return values. This approach allows RPC to be used across multiple programming languages.

When implementing an RPC system, the middleware must address several challenges:

- **Heterogeneity**: The caller and callee may be developed using different programming languages, run on different hardware or software platforms, and have different data representations. The RPC middleware must handle the conversion (marshaling/unmarshaling) of procedure arguments and return values to ensure compatibility.

- **Client-Server Communication**: The client and server typically run as separate processes on different hosts. The server's location must be determined before issuing the remote procedure call, often through dynamic linking. Additionally, the RPC middleware must handle the synchronization of multiple incoming calls on the server side.
- **Passing Arguments by Reference**: The RPC middleware needs to define how arguments can be passed by reference, considering that the address spaces of the client and server are usually disjoint. This includes managing the memory allocation and deallocation for referenced data.
- **Partial Failure**: In a distributed environment, it is possible for failures to occur during the RPC process. The RPC middleware must provide mechanisms to detect and handle partial failures, ensuring the system remains reliable and resilient.

5.2 Managing Partial Failure (Remote Call Semantics)

When making remote procedure calls, the semantics of the call differ from local calls due to the distributed nature of the system. Remote calls can encounter various issues, including server crashes, network outages, and message duplications. To handle these problems, RPC middleware protocols provide mechanisms to address partial failure. However, it's important for programmers to be aware of these issues and handle them accordingly.
In the context of remote calls, two kinds of return are possible:

1. **Successful Call**: The remote call completes successfully, and the expected result or response is returned to the client. This indicates that the remote procedure executed without any errors or exceptions.
2. **Unsuccessful Call**: The remote call encounters an error or failure during execution. This could be due to various reasons such as server crashes, network issues, or exceptions thrown during the procedure execution. In such cases, the client may receive an error response or an exception indicating the failure of the remote procedure call.

It is important for programmers to handle both successful and unsuccessful remote calls appropriately in their application code. They should handle exceptions, validate returned results, and implement appropriate error handling strategies to ensure the correctness and robustness of the distributed system.
When making remote procedure calls, the semantics of the call differ from local calls due to the distributed nature of the system. Remote calls can encounter various issues, including server crashes, network outages, and message duplications. RPC middleware protocols can handle these problems only partially, and programmers must be aware of them.
Depending on the RPC protocol used, different call semantics can be obtained:

- **At Least Once**: With at least once semantics, the RPC protocol ensures that the remote procedure call is executed at least once. In the event of a failure, the client will retry the call until a successful response is received. This semantics guarantees that the request will eventually be processed, but it can lead to duplicate execution of the procedure if the response is lost or delayed. In this case , procedures must be idempotent (i.e. not sensitive to number of calls), so that if executed multiple times, they can provide an idempotent result. In this case the transport layer protocol is UDP and the RPC protocol incorporates mechanisms for associating responses with requests and managing retransmissions.
- **At Most Once**: At most once semantics guarantee that the remote procedure call is executed at most once. The RPC protocol takes measures to ensure that duplicate requests are eliminated. This is typically achieved by assigning a unique identifier to each request and having the server maintain a record of processed requests to detect duplicates. If a request is lost or fails, the client can reissue the request without concern for duplicate execution. Here, the TCP transport protocol can be used, and the RPC protocol associates the response to the request via the connection. Otherwise, in UDP transport, the RPC protocol includes response-request association, retransmission management, and duplicate checking at the server.

- **Exactly Once**: Exactly once semantics aim to guarantee that the remote procedure call is executed exactly once. This is the strongest guarantee among the three semantics. Achieving exactly once semantics can be challenging, as it requires careful coordination between the client and server. Various mechanisms, such as message sequencing, duplicate detection, and idempotent operations, can be used to achieve exactly once semantics.

The choice of call semantics depends on the requirements of the distributed system and the trade-offs between reliability, performance, and simplicity.

5.3 Other RPC Issues

Mainly there are two other fields of issues to deal with when discussing RPC:

- **Security**: Main security requirements in RPC include access control, authentication, confidentiality, and integrity.
- **Performance**: A significant performance issue in RPC is that a remote call is much slower compared to a local call, often by orders of magnitude.

5.4 Different Types of RPC Request/Response Mechanisms

There are different types of RPC mechanisms in terms of Request/Response interaction.

5.4.1 Classical (Synchronous) RPC

Basic Request/Response interaction in which a client sends a request to a server and starts waiting for a response. The request is made up by:

- **Procedure Identification**
- **Input Parameters**

While the response sent back by the server:

- **Success/Failure Indication**
- **Return Value** and Output parameters

5.4.2 One-Way (Asynchronous) RPC

This is the case in which the client starts the interaction sending a request but is not waiting for a response back. In this case we are sure that there will not be exceptions thrown and that the operation performed by the server is successful (or at least we will rely on that).

5.4.3 Two-Way (Asynchronous) RPC

This is almost the same as the first one (the synchronous procedure) but in this case the client itself will not wait for the response. It might be that a new thread would wait for the answer, so that the main client is still able to perform other actions. Callback manages the response.

5.5 On top of Socket APIs or RPC?

We need to make a consideration about what happens if we rely only on TCP as transport protocol instead of relying on top of RPC Middleware for a distributed application. Basically, sockets only provide communication services, so in that case we need to solve many other problems in our application like: data encoding for exchanged data, interaction procedures, management of partial failures, etc., as a result the complexity increases. Generally the idea is to develop middlewares on top of Socket APIs or custom application-level protocols. On the other hand, developing the application on top of RPC allows to develop it as normal centralized code but having in mind that interactions will be distributed. Application modules are distributed on multiple hosts, and the application is tested in the distributed environment, in this case the programmer can focus more on application logic than on communication.

Another important key point to highlight is that applications can be seen as a combination of services. Services are designed and developed as reusable components that interact via RPC, this brings many advantages: services are simple reusable components, scaling out can be achieved by creating multiple service instances that are based on request volume.

6 HTTP/2

Before diving into gRPC, it's important to understand the fundamentals of the HTTP/2 protocol. Initially introduced by Google and later standardized by IETF, HTTP/2 retains the same semantics as HTTP/1.1 (RESTful), but enhances efficiency and performance through different data encoding and the utilization of TCP. One notable feature is the unsolicited push of resources, allowing the server to push resources to the client without explicit requests. This mechanism greatly improves the overall efficiency and responsiveness of web communication.

6.1 HTTP/2 vs. HTTP/1.1

There are many differences between HTTP/2 and HTTP/1.1. Particularly, the main differences are in: data encoding, multiplexing, compression and push possibility.

	HTTP/2	HTTP/1.1
Data Encoding	Binary	Character-oriented
Multiplexing	On single TCP connection	On multiple TCP connections
Compression	Header and body	Only body
Push Possibility	Yes	No

When discussing the exchange of binary messages in HTTP/2, two important concepts come into play: frames and streams. A frame represents a binary message exchanged between a client and a server. On the other hand, a stream refers to an independent, bidirectional sequence of frames. Each stream corresponds to an ordered sequence of logical operations, such as Request/Response or Push, which are essentially HTTP messages.
In HTTP/2, streams are multiplexed on a single TCP connection, meaning that multiple streams can be transmitted simultaneously. The ordering of frames within each stream is preserved, ensuring the integrity of the message flow. Additionally, window-based flow control mechanisms are introduced to regulate the rate of data transmission for each stream.

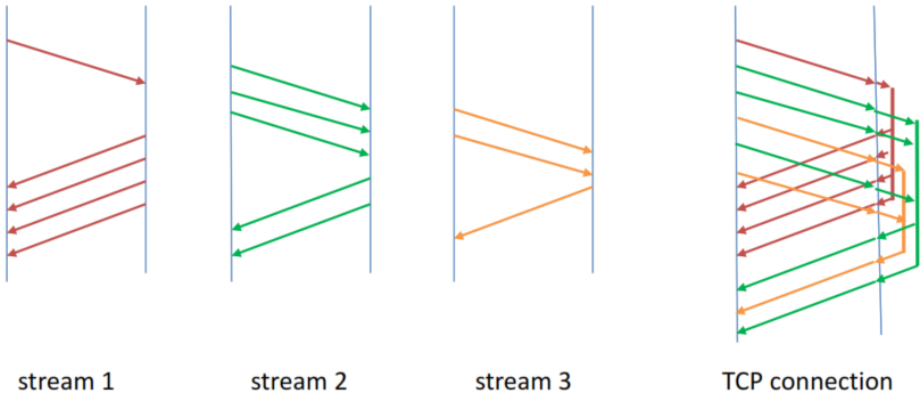

Figure 29: HTTP/2: Stream Multiplexing

One notable feature of HTTP/2 is server push. This allows the server to proactively push additional representations to the client, anticipating its future requests. This approach helps to reduce latency by trading bandwidth usage. For example, when a web page contains links referencing other resources, instead of waiting for the client to request each resource individually, the server can push these resources to the client preemptively.

6.2 HTTP/2 Connection Initiation, Negotiation and Upgrade

When the client knows the server supports HTTP/2, the communication process begins by establishing a TCP connection. The client initiates the connection by sending the 24-byte preface string ("`PRI *`

`HTTP/2.0\r\n\r\nSM\r\n\r\n`") followed by a SETTINGS frame. The server responds with its preface string plus a SETTINGS frame. Streams can then be opened for further communication.
In cases where the client has no prior knowledge, HTTP/2 negotiation is required. For TLS-based negotiation, commonly used in HTTPS connections, the client can employ the TLS Application Layer Protocol Negotiation (ALPN). The client requests one of several protocols, including HTTP/2, from the server. The server selects a protocol and informs the client in the response. If HTTP/2 is selected, the same handshake process as mentioned earlier takes place, involving the exchange of preface strings and SETTINGS frames.
Alternatively, HTTP-based negotiation can be used. In this approach, the client can negotiate HTTP/2 via the HTTP/1.1 Upgrade mechanism. The client sends an HTTP/1.1 request that includes an **Upgrade header** indicating the desire to upgrade to HTTP/2. If the server supports HTTP/2, it responds with a 101 Switching Protocols status code and includes an Upgrade header confirming the upgrade to HTTP/2. Once again, the subsequent handshake process occurs as previously described.
In both cases, the negotiation process allows the client and server to establish communication using the HTTP/2 protocol, ensuring efficient and optimized data exchange.

6.3 Error Management

In addition to the error reporting mechanisms provided by HTTP/1 *(Response-level Error Reporting)*, HTTP/2 offers additional error reporting mechanisms at both the stream-level and connection-level.

- **Stream-level Errors**: These errors occur in the management of individual streams. HTTP/2 allows for more precise reporting of errors related to specific streams, providing detailed information about the issues encountered during the processing of a particular stream.
- **Connection-level Errors**: These errors occur in the management of the TCP connection. HTTP/2 includes mechanisms to report errors that affect the entire connection, such as issues related to the establishment or maintenance of the TCP connection itself.

By offering stream-level and connection-level error reporting, HTTP/2 enhances the precision and granularity of error reporting compared to HTTP/1. This enables more detailed and accurate information about non-completed requests and errors encountered during the communication between the client and server. In addition to the error management mechanisms mentioned earlier, there are other ways to learn about the status of non-completed requests in HTTP/2.

- GOAWAY frame: The `GOAWAY` frame is used to indicate the highest stream number that the server has processed. This information is valuable for clients to determine the safety of retrying requests. Any requests sent on streams with higher numbers than indicated in the `GOAWAY` frame can be safely retried, as they are guaranteed to have not been processed by the server.
- REFUSED_STREAM error code: The `REFUSED_STREAM` error code can be included in a `RST_STREAM` frame. When a server sends a `RST_STREAM` frame with the `REFUSED_STREAM` error code, it indicates that the stream is being closed prior to any processing having occurred. In this case, any request that was sent on the reset stream can be safely retried.

By using the GOAWAY frame and the `REFUSED_STREAM` error code, HTTP/2 provides mechanisms for clients to gather information about the status of non-completed requests. This allows clients to make informed decisions on whether to retry requests or take appropriate actions based on the server's response.

7 gRPC

gRPC is an RPC (Remote Procedure Call) mechanism that operates on top of HTTP/2. It leverages the performance-oriented features of HTTP/2, such as binary messages and header compression. Similar to SOAP, gRPC utilizes HTTP/2 as its transport layer. The typical data representation used in gRPC is protocol buffers, which provides a language-agnostic way to define the structure of messages. Some key features of gRPC include:

- **Multi-language support**: gRPC supports multiple programming languages, allowing clients and servers to be implemented in different languages while still communicating seamlessly.
- **At most once semantics**: gRPC ensures that a remote procedure is executed at most once, providing reliability and consistency in the communication between clients and servers.
- **Security management**: gRPC supports security features such as Transport Layer Security (TLS), ensuring secure communication over the network.
- **Multiple ways of operation**: gRPC supports various modes of operation, including synchronous, asynchronous, and stream-oriented communication. This flexibility allows developers to choose the most suitable approach based on their application requirements.

By utilizing the advantages of HTTP/2 and protocol buffers, gRPC offers efficient and reliable communication between distributed systems, making it a popular choice for building modern microservices architectures.

7.1 Protocol Buffer: a serialization mechanism for gRPC

Protocol Buffers is a system-independent data representation and Interface Definition Language (IDL) that provides an abstract language for defining the structure of messages. It offers a binary representation format that is more compact and efficient compared to other data interchange formats like JSON or XML, this especially because HTTP/2 is already using binary message representation.
Some key features of Protocol Buffers include:

- **System-independent representation**: Protocol Buffers provide a language-agnostic way to define the structure of messages. This means that you can define your message formats once and generate code bindings for all major programming languages, enabling seamless communication between different systems.
- **Binary representation**: Protocol Buffers use a binary format for data serialization, which allows for more efficient encoding and decoding of messages. The compact binary format helps in reducing the payload size and improving the performance of data transmission.
- **Bindings for major programming languages**: Protocol Buffers provide language-specific bindings for all major programming languages, including Java, C++, Python, Go, and more. These bindings allow developers to easily work with Protocol Buffers in their preferred programming language and leverage the generated code for efficient message serialization and deserialization.

By offering a system-independent data representation and providing language-specific bindings, Protocol Buffers simplify the development of cross-platform and interoperable applications. Its compact binary format and efficient code generation make it a popular choice for structuring and serializing data in various domains, ranging from microservices architectures to distributed systems. At this point let's give a look to the message types: in gRPC, message types are user-defined named data structures that can be used for RPC. These message types consist of a collection of typed, named, and numbered fields. Each field can be optional and singular (allowing 0 or 1 repetitions) or optional and repeated (allowing 0 or more repetitions).
For example, consider the message type "**Book**" which has fields for "**isbn**", "**title**", and "**authors**". The "**isbn**" field has the number 1, the "**title**" field has the number 2, and the "**authors**" field has the number 3. If a field is absent when the message is sent, its default value will be returned when

the message is read on the receiving end. In these notes we will not discuss about data types, enums, map types and other details of gRPC.
This structured approach to defining message types allows for efficient and standardized data exchange in gRPC across different programming languages.

7.2 gRPC: RPC Supported Types

gRPC supports various types of Remote Procedure Calls (RPC) that cater to different communication patterns and requirements:

- **Classical Synchronous RPC**: This is the traditional RPC approach where the client makes a request to the server and waits for a response. The communication is synchronous, meaning the client is blocked until it receives a response from the server.
- **Two-Way Asynchronous RPC**: In this type of RPC, the client sends a request to the server, but instead of waiting for a response, it continues its execution. The server processes the request and sends the response back to the client separately. This enables concurrent execution on both the client and server sides.
- **Streaming Mode**: gRPC supports streaming in both the request and response directions. This allows for a continuous flow of data between the client and server, enabling scenarios such as real-time updates, data streaming, and bidirectional communication.

8 Synchronization

In a distributed system, processes operate asynchronously, which means they may progress at different speeds and experience varying delays. However, there are scenarios where synchronization becomes necessary to enforce ordering of events or meet specific timing requirements. Two common approaches for achieving synchronization in decentralized systems are:

- **Synchronize Physical Clocks**: This approach involves synchronizing the physical clocks of different processes in the system. By ensuring that all processes have a common notion of time, it becomes easier to coordinate actions and enforce temporal ordering. Techniques like the Network Time Protocol (NTP) can be used to synchronize clocks across distributed nodes.
- **Logical Clocks**: In cases where precise time synchronization is not feasible or not required, logical clocks provide a more relaxed form of synchronization. Logical clocks assign logical timestamps to events based on causal relationships. They enable ordering of events without relying on physical time and are particularly useful for maintaining consistency in distributed systems.

These synchronization mechanisms play a crucial role in distributed systems, ensuring coordination, consistency, and reliable communication among processes.

8.1 Physical Clock Synchronization

Physical clock synchronization in distributed systems can be achieved through various methods. Some possibilities include:

- **UTC Receivers:** Synchronization using UTC (Coordinated Universal Time) receivers provides highly accurate results with precision up to 50ns. However, this approach can be expensive to implement.
- **NTP:** Network Time Protocol (NTP) is a more cost-effective solution for clock synchronization. It relies on time servers and can achieve reasonable accuracy, although it may not be as precise as UTC receivers.

The accuracy of NTP synchronization can vary depending on the network conditions:

Table 1: NTP Synchronization Accuracy

Network Type	**Accuracy**
LAN	$<$ 1ms
Public Internet	10-50ms
Public Internet under Congestion	$>$ 100ms

Physical clock synchronization is an important aspect of distributed systems, ensuring consistent and coordinated behavior across different nodes.

8.2 Logical Clocks

In distributed systems, achieving agreement on the ordering of events is often sufficient for synchronization purposes. Logical clocks are distributed algorithms that can be used to achieve this sort of agreement.

Logical clocks provide a way to establish a partial ordering of events in a distributed system without relying on physical clock synchronization. Instead, each process in the system maintains its own logical clock, which is incremented as events occur. The ordering of events is determined based on the values of these logical clocks.

There are different algorithms for implementing logical clocks, such as Lamport clocks and vector clocks. These algorithms allow processes to assign timestamps or vector values to events, enabling the determination of causality relationships among events.

By utilizing logical clocks, distributed systems can achieve synchronization in terms of event ordering, even in the absence of physical clock synchronization. This enables processes to reason about the causal relationships and dependencies between events in a distributed environment.

8.2.1 Lamport Clocks

Lamport clocks are a type of logical clock algorithm developed by Leslie Lamport. They provide a simple way to establish a partial ordering of events in a distributed system based on the notion of the "happens-before" relation.
In Lamport clocks, each process in a distributed system maintains a local clock value that is incremented as events occur. The clock value of a process represents the logical time at which an event happens in that process. The ordering of events is determined based on the comparison of these clock values.
The happens-before relation in Lamport clocks can be observed in the following cases:

- When event a happens before event b within the same process.
- When event a is the sending of a message and event b is the receiving of the same message by different processes.

By using Lamport clocks, processes in a distributed system can establish a partial ordering of events that captures causal relationships between events. However, it is important to note that Lamport clocks do not provide a global notion of physical time and do not guarantee accuracy in terms of real-time ordering.
Lamport clocks serve as a fundamental building block in distributed systems for reasoning about event ordering and ensuring consistency in message passing protocols.

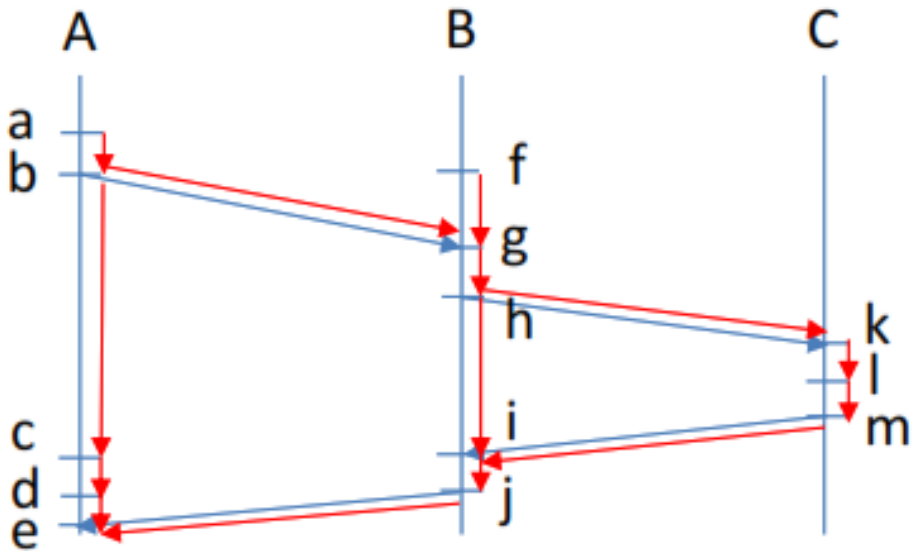

Figure 30: Lamport Clock

The Lamport clock algorithm operates as follows:

- Each process P_i maintains a local time counter C_i.
- Process P_i timestamps each local event a with the current value of C_i, denoted as $C_i(a)$, after incrementing C_i.
- Process P_i timestamps each message it sends with the timestamp $C_i(s)$ assigned to the send event s.
- When process P_i receives a message with timestamp C_r:
  - If $C_r > C_i$, process P_i adjusts its local clock by setting $C_i = C_r$.
  - After adjusting the clock, process P_i timestamps the receive event as usual.

The clock adjustment step ensures that the local clock C_i of process P_i is always equal to or greater than any timestamp received from other processes.
The clock adjustment step can be equivalently expressed as $C_i = \max(C_i, C_r)$, ensuring that the local clock is adjusted to the maximum value between its current value and the received timestamp.However, it's important to note that Lamport clocks do not provide accurate real-time ordering since they rely solely on the ordering of events and message exchanges.

Lamport Clocks: Discussion
When using Lamport clocks, it is important to note that some events in different processes may be assigned the same timestamp. If we want to avoid this situation, we can include the process ID into the timestamp. In this case, the timestamp of event a occurring in process P_i would be represented as $< C_i(a), i >$. This assumption relies on the uniqueness of process IDs within the distributed system.
It's important to highlight that Lamport clocks provide a partial ordering of events based on their timestamps. If event a happens before event b, then $C(a) < C(b)$ holds true. However, the reverse is not always true. It is possible for two events to have different timestamps ($C(a) < C(b)$) even if they are not causally related ($a \not\to b$).
The Lamport clock algorithm can be implemented by utilizing a timestamping function that increments and returns the counter value. Additionally, hooks can be incorporated into the send/receive functions to timestamp the send/receive events and messages, as well as adjust the counter value based on received timestamps.
By implementing the necessary functions and hooks, the Lamport clock algorithm enables processes in a distributed system to assign timestamps to events in a way that captures the causal ordering between events. This allows for the establishment of a partial ordering that helps synchronize and coordinate the execution of processes within the system.

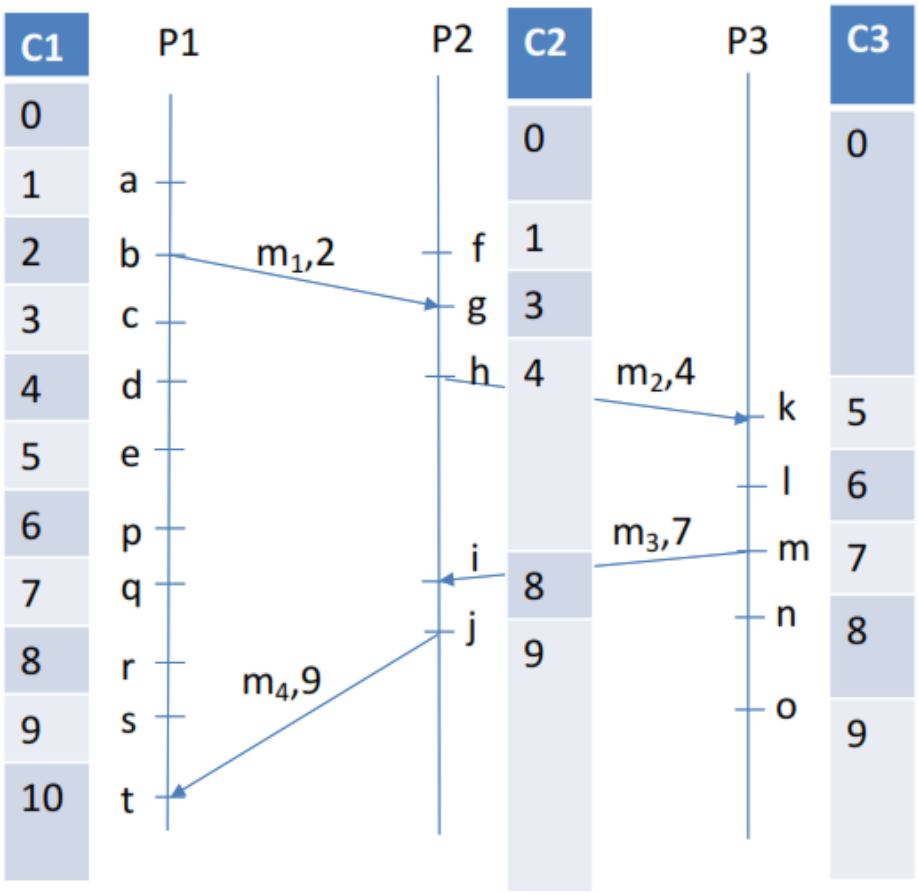

Figure 31: Lamport Clock Algorithm Example

Total-Ordered Multicast

In certain scenarios, such as replicated servers, there is a need for total-ordered multicast where all multicast messages must be delivered in the same order to each receiver. This application is particularly relevant when there are multiple identical replicated servers, aiming for fault-tolerance and latency reduction.

Consider a system with two or more replicated servers. The replication goals of such a system include ensuring fault-tolerance and reducing latency. In this setup, service requests can be sent to any server within the group. Upon receiving a request, the server then multicasts the request to all other servers in the group.

The key requirement in this scenario is that the operations must be executed in each replica in the same order. Total-ordered multicast ensures that all servers process the multicast messages in a consistent and synchronized manner. By guaranteeing the same order of execution, the system achieves the desired consistency across replicas.

Total-ordered multicast is crucial for maintaining the integrity and correctness of the replicated servers. It ensures that all replicas process requests in a consistent order, preventing any potential inconsistencies or conflicts that could arise from executing operations in different orders.

By employing total-ordered multicast, systems can achieve reliable and consistent behavior across replicated servers, enabling fault-tolerance and ensuring that the latency is reduced by parallel processing while maintaining the required order of execution.

The Total-ordered Multicast Algorithm utilizes Lamport clocks, which are maintained in each process within the system. The algorithm ensures that each multicast message is timestamped by the sender with the timestamp of the send event. Received messages are then queued and ordered based on their timestamps.

Here are the steps involved in the Total-ordered Multicast Algorithm:

1. Lamport clocks are maintained in each process of the system.
2. When a process sends a multicast message, it timestamps the message with the timestamp of the send event.
3. Upon receiving a multicast message, each receiver adds the message to its message queue, ordering the messages based on their timestamps.
4. Each receiver acknowledges the reception of the message to the other receivers.
5. Once acknowledgments for the message at the head of the queue have been received from all other receivers, that message is dequeued and handed to the application for further processing.

By utilizing Lamport clocks and the queue-based approach, the Total-ordered Multicast Algorithm ensures that messages are delivered and processed in the same order by all receivers. The acknowledgment mechanism provides a way to synchronize the receivers and guarantee that a message is only dequeued when all receivers have acknowledged its reception.

This algorithm is essential in scenarios where maintaining a consistent order of message delivery is crucial, such as in replicated server environments. By enforcing total-ordered multicast, the algorithm facilitates reliable and synchronized message processing across all receivers, ensuring the desired consistency and correctness in the system.

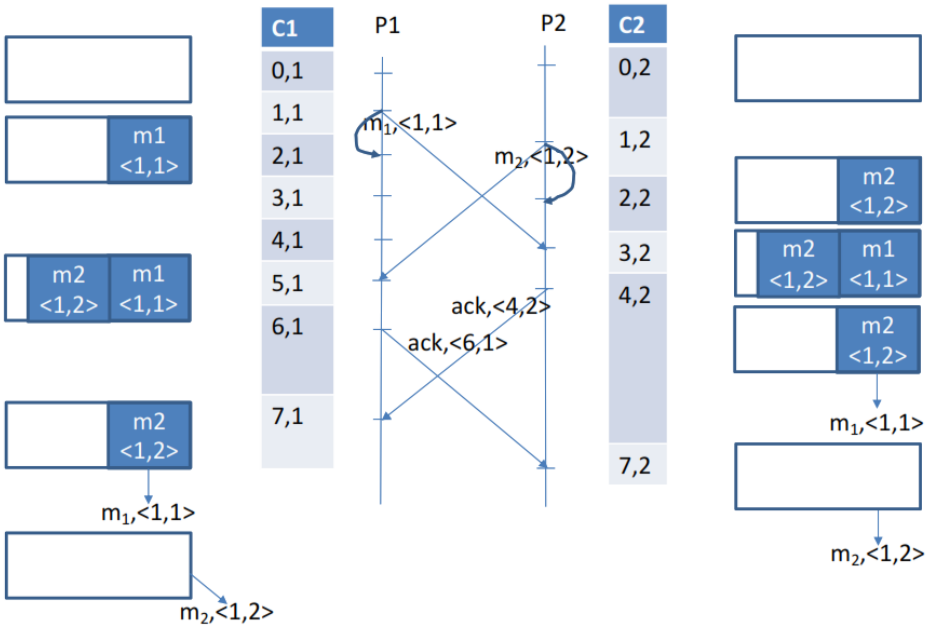

Figure 32: Total-Ordered Multicast Algorithm Example

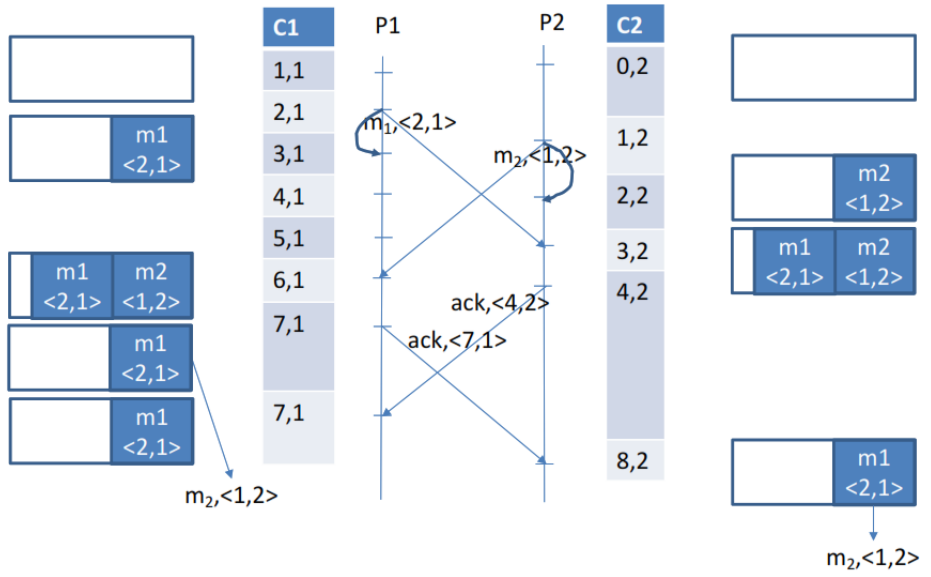

Figure 33: Total-Ordered Multicast Algorithm: Other Scenario

Total-Ordered Multicast: Discussion
The Total-ordered Multicast Algorithm has been proven to deliver messages to all destination processes in the same order, given the following assumptions:

1. No messages are lost during transmission.
2. Messages from the same sender are received by the receivers in the same order in which they were sent.

When these assumptions hold true, the algorithm guarantees that the order of message delivery is consistent across all receivers. This is particularly important when the messages being multicast represent operations to be executed on replicated data in the same order.

By implementing the Total-ordered Multicast Algorithm, the system achieves state machine replication. State machine replication ensures that each replica processes the operations in the same order, guaranteeing consistency in the replicated data. The algorithm enables reliable and synchronized execution of operations on replicated data, regardless of the order in which the operations are received by the replicas.
It is worth noting that the algorithm relies on the assumptions mentioned above. If messages are lost or received out of order, the algorithm's guarantee of total-ordered message delivery may not hold, and additional mechanisms, such as retransmission or error handling, may need to be implemented to handle such scenarios.
Overall, the Total-ordered Multicast Algorithm provides a robust solution for achieving consistent message delivery and synchronized execution of operations in replicated environments. By ensuring the same order of message processing across all receivers, it enables reliable and fault-tolerant state machine replication, contributing to the overall correctness and integrity of the system.

8.2.2 Vector Clocks

While Lamport clocks provide a total ordering of events, it is important to note that this ordering does not necessarily imply causality. In other words, if $C(a) < C(b)$, it does not guarantee that event a causally precedes event b.
To address this limitation, vector clocks were introduced as a mechanism to provide a partial ordering of events that captures causality. In vector clocks, each process maintains a vector of timestamps, where each element represents the logical clock value of a particular process.
The vector clock property $C(a) < C(b)$ is defined as follows:

- If event a causally precedes event b (a $\rightarrow$ b), then $C(a) < C(b)$.
- If $C(a) < C(b)$, it does not imply a causal relationship between a and b. They could be concurrent or unrelated.

By using vector clocks, we can determine the causal relationship between events, even in a distributed system. Each process increments its own logical clock value whenever it performs a local event and includes its vector clock in messages it sends to other processes. Upon receiving a message, a process updates its vector clock by taking the maximum value for each element between its own vector clock and the received vector clock.
The comparison of vector clocks allows for the identification of causally related events. If $C(a) < C(b)$, it implies that event a causally precedes event b. This property enables the detection of causality within a distributed system, aiding in various applications such as event ordering, distributed snapshots, and distributed consistency protocols.
It's important to note that vector clocks provide a partial ordering and cannot capture the exact causal dependencies between all events. However, they offer a practical and efficient approach for capturing causality in distributed systems, allowing for reasoning about causality and preserving causally related relationships between events.

The Vector Clocks algorithm allows for capturing causality in a distributed system. The algorithm involves the following steps:

1. Each process `Pi` maintains a local vector of time counters `VCi[]`, where `VCi[i]` represents the local event counter for process `Pi`, similar to Lamport clocks. Additionally, `VCi[j]` represents the knowledge of the local time at process `Pj` as known by process `Pi`.
2. When a local event occurs at process `Pi`, it timestamps the event with the current value of `VCi`, denoted as `ts(a)`. Before timestamping, `VCi[i]` is incremented to reflect the occurrence of the event.
3. When a message is sent by process `Pi`, it is also timestamped with the value `ts(s)`, assigned to the send event `s`.
4. Upon receiving a message with timestamp `tsr` by process `Pi`:

- `Pi` updates its local vector clock `VCi` by taking the maximum value between each element of `VCi` and `tsr`. In other words, `VCi[k]` is set to `max(VCi[k], tsr[k])` for each index k, ensuring that the vector clock reflects the most up-to-date knowledge of time among the processes.
- After adjusting the vector clock, `Pi` timestamps the receive event as usual with the updated `VCi` value.

By maintaining and updating the vector clocks at each process, the algorithm ensures that the causal dependencies among events can be captured. The vector clock adjustment during message reception allows processes to synchronize their knowledge of time and incorporate the timestamp information from other processes.
The comparison of vector clocks across different processes enables the determination of causal relationships between events, aiding in various applications such as event ordering, distributed snapshots, and distributed consistency protocols.
It's important to note that the Vector Clocks algorithm requires processes to exchange and update vector clock information during message passing, allowing them to maintain a consistent view of time across the distributed system and accurately capture causality between events.
The scenario of Lamport Clock presented previously (*figure: 31*) is now presented in Vector Clock version:

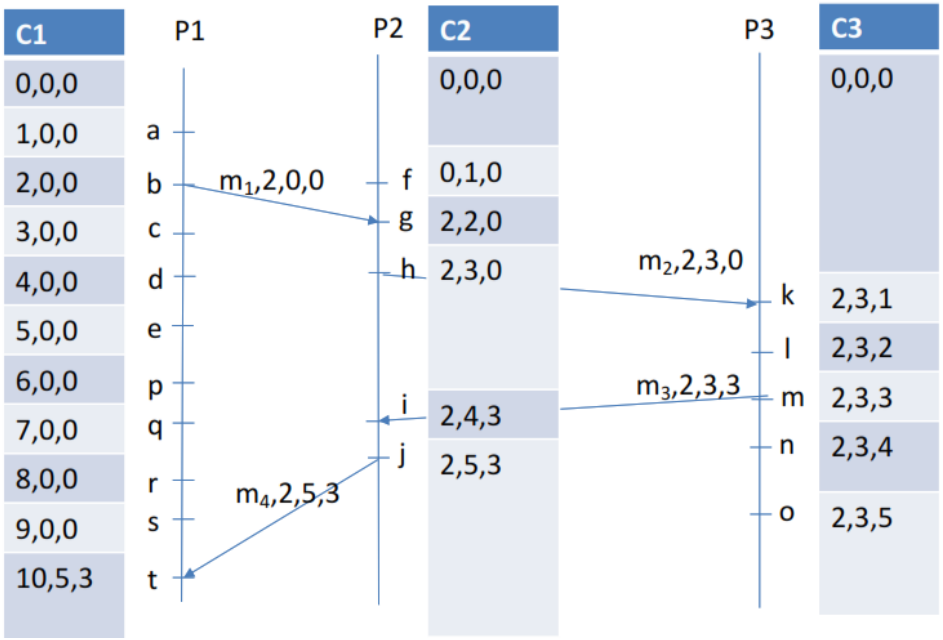

Figure 34: Vector Clock Algorithm

Vector Clock Application Example: Causal Ordered Multicast
In the context of multicast communication in a distributed system, ensuring causal ordering is important. Causal ordering means that each multicast message should be delivered only after all causally related messages sent earlier have been delivered. However, non-causally related messages sent earlier may be delivered in any order.
To achieve causal ordered multicast, vector clocks can be used. Each process maintains a local vector of time counters, denoted as `VC[i]`, where i represents the process identifier. When sending a multicast message, the sender timestamps it with its current vector clock value. Upon receiving a multicast message, a process updates its vector clock by taking the element-wise maximum of its local vector clock and the received timestamp.
By using vector clocks, a distributed system can ensure that multicast messages are delivered in a causally ordered manner, respecting the dependencies among messages while allowing non-causally related messages to be delivered in any order.

9 Coordination Algorithms

In distributed systems, coordination among multiple processes is crucial to ensure proper resource utilization and orderly execution. Coordination algorithms play a vital role in achieving synchronization, mutual exclusion, leader election, and consensus among processes. These algorithms enable processes to collaborate efficiently, maintain consistency, and make collective decisions in a distributed environment.

This section explores various coordination algorithms that address common challenges in distributed systems. We begin by examining the concept of mutual exclusion, which guarantees exclusive access to shared resources. Different types of mutual exclusion algorithms, such as token-based and permission-based approaches, are discussed.

Another critical coordination problem is the election of a leader or coordinator among a group of processes. We examine the Election problem and its requirements, assumptions, and solutions. The Bully Algorithm, which elects a coordinator through message exchanges, and the Ring Algorithm, which uses a logical ring structure for the election process, are explained in detail.

Lastly, we touch upon the concept of consensus, which involves achieving agreement among processes on a common output value. Consensus is a more general coordination problem and can be approached using leader election or mutual exclusion techniques.

Through this section, we gain insights into the fundamental coordination algorithms that facilitate efficient and reliable collaboration among distributed processes. Understanding these algorithms empowers us to design robust distributed systems capable of handling concurrent access to shared resources, electing leaders, and reaching collective decisions.

9.1 Mutual Exclusion

To guarantee mutually exclusive access to shared resources by multiple processes in a distributed system, there are different types of algorithms:

- Token-based
- Permission-based
  - Centralized
  - Distributed

9.1.1 Token Ring Mutual Exclusion

In this algorithm, processes are organized in a ring overlay. There is one token in the system that continually circulates on the ring. When a process P wants to access a resource, it waits for the token. Upon receiving the token, process P starts accessing the resource and keeps the token until its access is finished. Then, it passes the token to the next process in the ring.

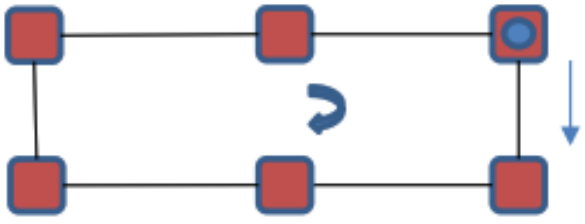

Figure 35: Token Ring Mutual Exclusion Algorithm

9.1.2 Centralized Mutual Exclusion

This algorithm involves a central manager (C) responsible for managing shared resources. When a process P wants to access a resource, it sends a request to the central manager and waits for a

permission response. The central manager delays granting permissions while the resource is already engaged.

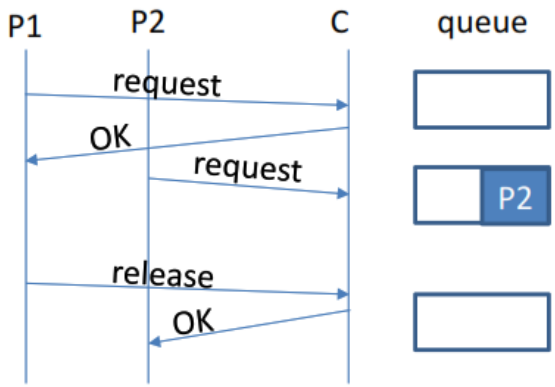

Figure 36: Centralized Mutual Exclusion Algorithm

9.1.3 Distributed Mutual Exclusion

The distributed mutual exclusion algorithm is based on Lamport clocks for totally ordered multicast. When a requester process wants to access a resource, it sends a request to all processes (including itself) and waits for permission from every process. In case of conflict, where another process also wants to access the resource, the request with the lower timestamp takes precedence.

9.1.4 Performance Comparison

After describing the previous exclusion algorithm, it is strongly meaningful to compare their performances:

Algorithm		#Messages/access	delay before entry (# messages)
Permission based	Centralized	3	2
	Distributed	2 (N-1)	2 (N-1)
Token based	Token ring	1...	0 ... N-1

Figure 37: Performances Comparisons

9.2 Election

The election problem involves electing a process from a group of processes to fulfill a certain work or take a specific role among the others (e.g., a coordinator). In this context some assumptions are needed:

- Each process has a unique identifier id(P).
- Each process knows all the other processes in the group.
- Processes in the group can be up and running or down, but channels are reliable.

The algorithm must elect the up process with the highest id, and at the end of the algorithm, all processes should agree on who the elected process is.

9.2.1 Election: The Bully Algorithm

The Bully Algorithm is used to elect a coordinator from a group of processes. Each process has a unique identifier. When a process detects that the coordinator is missing, it starts an election by sending an ELECTION message to higher numbered processes. If no response is received, the process wins the election; otherwise, it gives up. The winner informs the other processes about its election.

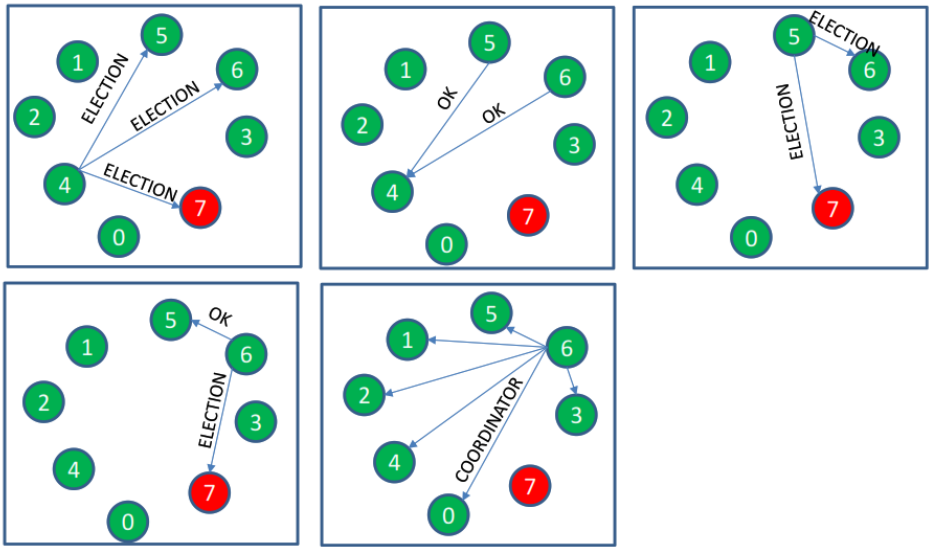

Figure 38: Election Bully Algorithm

9.2.2 Election: Ring Algorithm

The Ring Algorithm elects a coordinator using a logical ring structure. When a process detects that the coordinator is missing, it starts an election by sending an ELECTION message to its successor in the ring. If no response is received, it forwards the message to the next successor until a response is received. The ELECTION message carries the list of senders. When the message returns to the starting process, it stops the circulation and computes the winner. The winner then circulates a COORDINATOR message on the ring to inform about the winner.

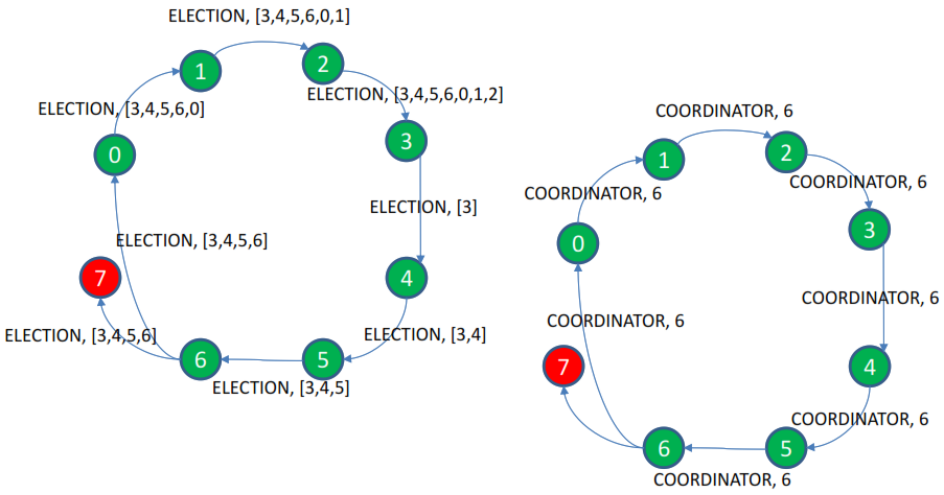

Figure 39: Election Ring Algorithm

9.3 Consensus

Consensus is a more general coordination problem where n processes, proposing an input value, need to agree on the same output value. Mutual exclusion and leader election are special cases of consensus. Consensus can be achieved by using leader election or mutual exclusion.

10 Replication Consistency

Data replication is a technique used to improve the reliability and performance of a system. In this section, we will focus on performance-related data replication, which involves scaling out techniques for size and geographical scalability. Ideally, data replication should be transparent to the user, but it introduces a challenge related to consistency. When replicated data changes, inconsistent states can arise, temporarily making the replicas not identical and breaking transparency. However, we can try to hide these inconsistencies at the cost of additional latency and performance overhead.

10.1 The Cost of Consistency

In order to resolve inconsistencies, changes in data are copied to all replicas. This introduces bandwidth and computation overhead for making copies and adds latency to hide the inconsistent states. While replication can improve performance and scalability, keeping replicas consistent comes with a performance cost. The balance between consistency and performance depends on the specific consistency requirements.

10.2 Consistency Models

A consistency model serves as a contract between processes and the data store. It defines the consistency properties guaranteed by the data store in read/write operations. Strict consistency, for example, ensures that each data change is propagated to all replicas before executing the next operation on the data store. This model is easy to understand and use but incurs a high performance cost. In practice, to improve performance, consistency requirements are relaxed, making the model less straightforward for programmers. Generally, this is the reference system model:

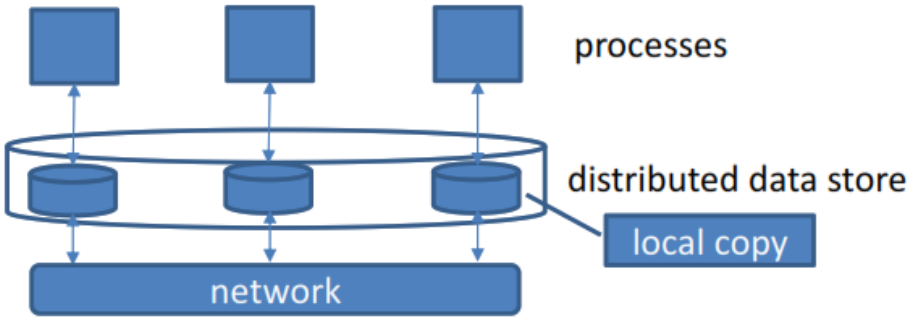

Figure 40: Consistency Model

10.2.1 Data-Centric Consistency Models

Consistency properties can be expressed in terms of how read/write operations on the global data store behave. For example, strict consistency ensures that each read returns the last written data. By relaxing strict consistency, we relax the constraints on read and write operations.

10.2.2 Continuous Consistency

Consistency can be expressed and measured as the amount of deviation tolerated from strict consistency. This can include numerical deviation, staleness deviation, and deviation in the ordering of write operations. The continuum of consistency measures provides flexibility in balancing performance and consistency.

10.2.3 Consistency Units (Conits)

When measuring consistency, it is common to refer to data units called conits rather than the entire data store. The granularity of conits introduces tradeoffs. Large conits may lead to significant inconsistencies, while small conits require managing a larger number of units.

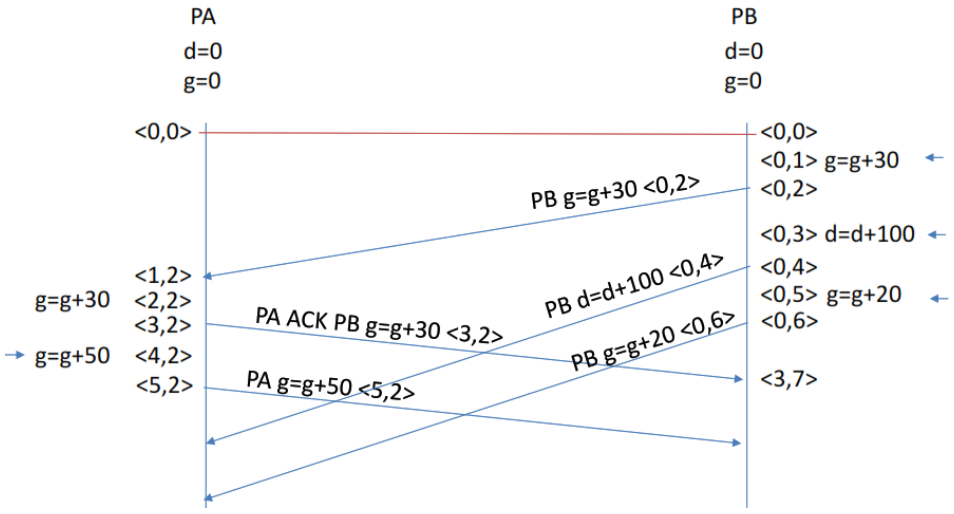

Figure 41: Example: (see Van Steen, Tanenbaum)

10.2.4 Sequential Consistency (Lamport)

According to Lamport's sequential consistency model, the result of any execution is the same as if the read and write operations by all processes on the data store were executed in some sequential order specified by their programs. While any interleaving of read/write operations satisfying the order specified by the programs is possible (nondeterminism), all processes see the same interleaving.

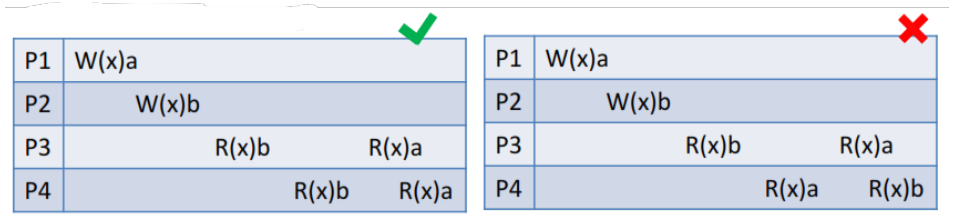

Figure 42: Example: Sequential Consistency Lamport Clock

10.2.5 Causal Consistency

Causal consistency ensures that writes that are potentially causally related must be seen by all processes in the same order. However, concurrent writes may be seen in a different order on different machines. Causal consistency is weaker than sequential consistency.

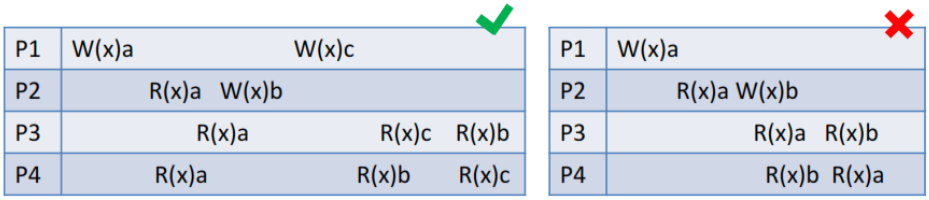

Figure 43: Example: Causal Consistency

10.2.6 Entry Consistency

Entry consistency is a consistency model that refers to groups of atomically executed operations, often referred to as critical sections. Consistency properties in entry consistency are expressed in terms of read, write, lock, and unlock operations. Acquiring a lock can only succeed when all writes to the

associated shared data have been completed, exclusive access to a lock can only succeed if no other process has exclusive or nonexclusive access to that lock, and nonexclusive access to a lock is allowed only if any previous exclusive access has been completed.

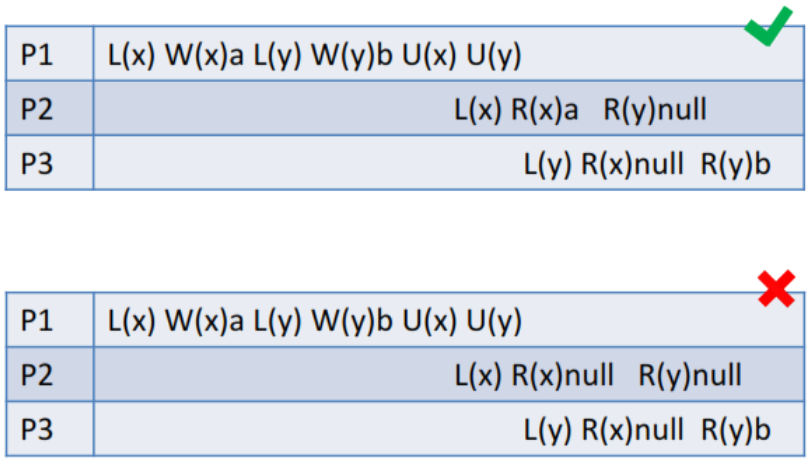

Figure 44: Example: Entry Consistency

10.2.7 Eventual Consistency

In some applications, weaker consistency guarantees are acceptable. Eventual consistency is a model where data stores, seldom written by few processes but frequently read, have slow propagation of updates. Eventually, all replicas become identical if no updates take place for a long time, although the duration is not strictly specified. Eventual consistency works well when there are few updates and when processes always access the same replica, but process mobility may cause problems.

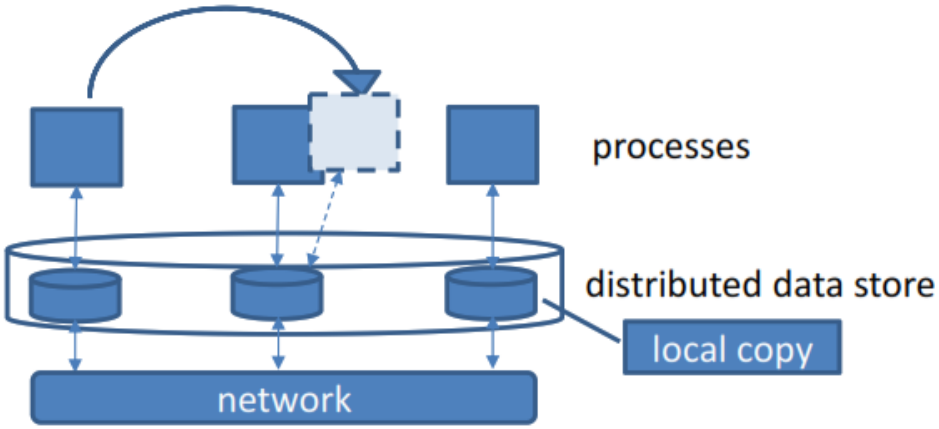

Figure 45: Example: Eventual Consistency

10.2.8 Client-Centric Consistency Models

While data-centric consistency models provide system-wide data consistency properties, client-centric consistency models provide consistency properties for each individual process or client. These models can be added to limit problems related to mobility when eventual consistency is used.

xj	**version** j of variable x
Wk(xj)	Pk writes xj
Wk(x1;x2)	Pk writes x2 which follows from x1
Wk(x1|x2)	Pk writes x2 concurrently with x1 (potential write-write conflict)

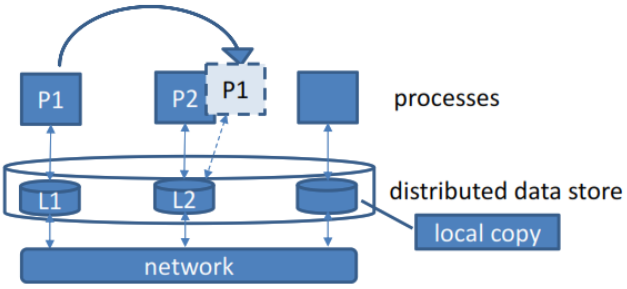

Figure 46: Example: Client-Centric Consistency Models with Eventual Consistency

Monotonic Reads According to the monotonic reads consistency model, if a process reads the value of a data item, any successive read operation on the same data item by that process will always return the same value or a more recent value.

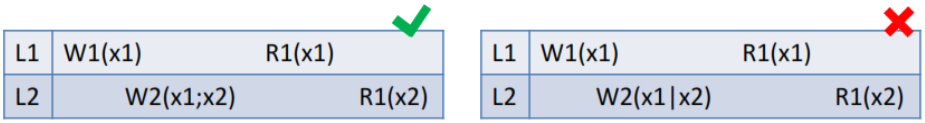

Figure 47: Example: Monotonic Reads

Monotonic Writes In the monotonic writes consistency model, a write operation by a process on a data item is completed before any successive write operation on the same data item by the same process.

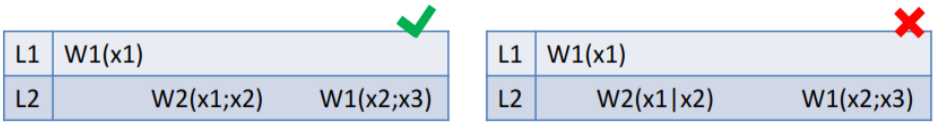

Figure 48: Example: Monotonic Writes

Read Your Writes The read your writes consistency model states that the effect of a write operation by a process on a data item will always be seen by a successive read operation on the same data item by the same process.

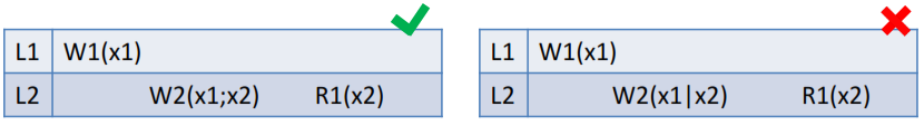

Figure 49: Example: Read your Writes

Writes Follow Reads According to the writes follow reads consistency model, a write operation by a process on a data item following a previous read operation on the same data item by the same process is guaranteed to take place on the same or a more recent value that was read.

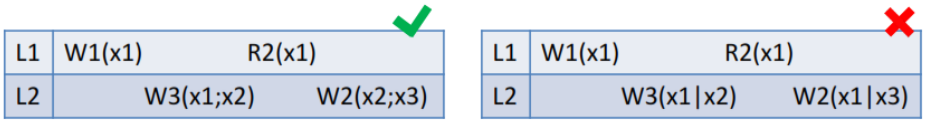

Figure 50: Example: Writes Follows Reads

10.3 Replica Management

Replica management involves deciding which servers or contents to replicate and where to place the replicas. It is an important aspect of data replication to ensure performance and reliability. Mainly, there are 2 different ways of performing it:

- **Server Replication**: it focuses on determining which servers to replicate and the placement problem, which involves deciding where to put the replicated servers. This is particularly important for achieving load balancing and fault tolerance.
- **Content Replication**: this, on the other hand, deals with deciding what contents should be replicated and the placement problem of where to put the replicas. The goal is to optimize content availability and reduce latency for accessing data.

10.3.1 Update Propagation Strategies

Update propagation strategies define how updates are propagated in a replicated system. Different strategies can be used depending on the read-to-update ratio, consistency requirements, and other factors. Basically, update propagation can take different forms:

- **Update notification propagation**: also known as invalidation protocols, involves notifying replicas about updates. This strategy works best in scenarios with a low read-to-update ratio.
- **Data propagation**: involves propagating the actual data to replicas. This strategy is more suitable for scenarios with a high read-to-update ratio.
- **Update operation propagation**: also known as active replication, involves propagating the update operations themselves. However, this approach is not always feasible or convenient in all scenarios.

10.3.2 Push (Server-based) vs. Pull (Client-based) Protocols

Pull and push protocols are different approaches to replica synchronization in a replicated system. Push protocols are generally used for permanent and server-initiated replicas. They are suitable when strong consistency is required and work well in scenarios with a high read-to-update ratio. Pull protocols, on the other hand, are generally used for client-initiated replicas. They are more suitable when weak consistency is acceptable and work well in scenarios with a low read-to-update ratio. However, response time may increase with cache misses.

	Push	Pull
State at server	list of client replicas+caches	none
Messages sent	update (+ fetch if invalidation)	poll (+ fetch if changed)
Response time at client	0 (or fetch time if invalidation)	poll (+fetch) time

Figure 51: Push vs Pull Protocols

10.4 Protocols for Continuous Consistency

Protocols for continuous consistency involve processes performing writes on local copies tentatively. Each process propagates its local writes to other processes, which then detect conflicts and apply updates to their local copies. Consistency metrics are monitored, and corrective actions are taken when deviations occur, such as stopping the execution of more writes on the local copy or forcing/requesting update propagation.

10.5 Protocols for Sequential Consistency

Protocols for sequential consistency focus on ensuring that the execution of read and write operations on a replicated system behaves as if they were executed in some sequential order. Two commonly used protocols for achieving sequential consistency are primary-based protocols and replicated write protocols.

10.5.1 Primary-Based Protocols

Primary-based protocols assign a primary process responsible for coordinating write operations on a particular data item. There are two main variations of primary-based protocols:

- **Remote Write (Primary Backup) Protocols**: In this variant, the primary for a data item is fixed. The primary process receives write requests and ensures the coordination of write operations on the data item.
- **Local Write Protocols**: In local write protocols, the primary for a data item migrates to the process that needs to write to it. This migration involves transferring the primary copy of the data item to the writer process. By migrating the primary, write operations can be performed locally, avoiding the need for communication with a fixed primary process.

10.5.2 Replicated Write Protocols

Replicated write protocols allow write operations to be initiated at different replicas. They ensure that the global ordering of operations remains consistent across the replicas. There are two commonly used approaches for achieving replicated write protocols:

- **Active Replication Protocols**: Active replication protocols maintain a global ordering of operations by performing ordered multicast or using centralized sequencers. These protocols ensure that write operations are applied in the same order across all replicas, guaranteeing sequential consistency.
- **Quorum-Based Protocols**: Quorum-based protocols record versions of data and require agreement from a majority of processes for write and read operations. Each process willing to write or read initiates a voting procedure, and only operations that receive agreement from the majority of processes are allowed to proceed. This approach ensures that the replicas reach a consensus on the ordering of operations, preserving sequential consistency.

10.6 Cache Coherence Protocols

Cache coherence protocols are specific to systems that involve caching, typically seen in client-server systems like web applications. These protocols aim to maintain coherence and consistency across caches.

10.6.1 Coherence Strategy

Coherence strategies determine how inconsistencies are detected and handled within the system. There are three common coherence strategies:

- **Client Validates Consistency Before Proceeding**: In this strategy, the client validates the consistency of the data before proceeding with a transaction. If inconsistencies are detected, the transaction is not carried out.

- **Client Validates Consistency While Proceeding (Optimistic Approach)**: This strategy involves the client proceeding with the transaction and validating consistency during the process. If validation fails, indicating inconsistencies, the transaction is aborted.
- **Client Validates Consistency at the End of the Transaction**: In this strategy, the client completes the transaction and validates consistency at the end. If inconsistencies are found, the transaction is aborted.

10.6.2 Enforcement Strategy

The enforcement strategy determines how consistency is enforced in the system. Two common approaches are used for enforcing consistency in read-write caches:

- **Read-Only Caches with Push or Pull Mechanisms**: In this approach, caches are designated as read-only, and mechanisms like push or pull are used to update the cache with the latest data. When inconsistencies are detected, the cache is updated through these mechanisms.
- **Read-Write Caches with Primary-Based Local-Write Protocols**: This approach involves using protocols similar to primary-based local-write protocols. The client requests a lock on the data, and its cache becomes a temporary primary, allowing concurrent writes with a conflict resolution strategy to be admitted.

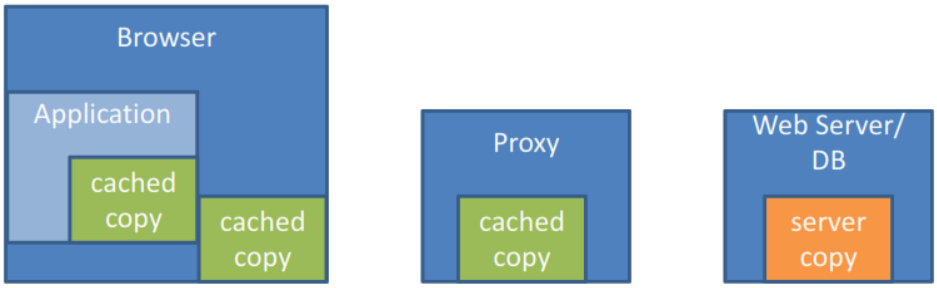

Figure 52: Cache Management in Web Applications

11 WebSockets

WebSockets have become a popular choice for implementing interactive and real-time applications. They provide reliable, low-latency, and bidirectional communication channels, making them suitable for various use cases such as cooperative work, highly interactive user interfaces, gaming, video-conferencing, and asynchronous notifications from servers to clients.

While request-response interactions have been widely used for many applications, they are not always suitable for scenarios requiring real-time push notifications. Traditional solutions like polling and long polling have proven to be inefficient, especially for frequent notifications. COMET, which includes techniques like HTTP streaming and subscription, is another approach but requires the client to play a server role and is not fully satisfactory for bidirectional low-latency communication.

WebSockets are an application-level protocol designed to provide reliable, low-latency, and bidirectional communication channels. They offer a **TCP-like** communication service with some differences and leverage the existing web infrastructure. This includes reusing the same HTTP ports (80, 443), compatibility with proxies and other web intermediaries, and the ability to work side-by-side with regular HTTP-based communications. WebSockets also follow the same-origin policy (SOP)-based security model. Communication in WebSockets is based on message-based communication over a binary framing structure layered upon TCP. This allows for efficient and flexible data transmission.

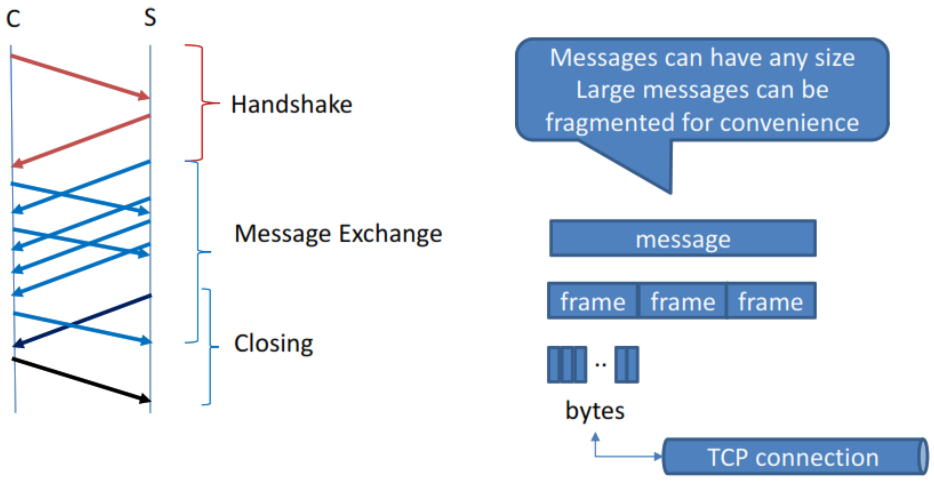

Figure 53: Web Socket Protocol (RFC 6455)

11.1 WebSockets vs TCP

WebSockets provide TCP connections with some additional features, like:

- Web origin-based security model for browsers.
- Addressing and protocol naming mechanism that supports:
  - Multiple endpoints on one port
  - Multiple host names on one IP address.
- Framing and messaging mechanism with no message length limit.
- In-band additional closing mechanism that works with proxies and intermediaries.

11.2 WebSocket URIs

Websocket endpoints are uniquely identified by URIs, similar to HTTP URIs but with the "`ws`" or "`wss`" scheme. The "`ws`" scheme is used for WebSocket connections over TCP, while the "`wss`" scheme is used for WebSocket connections over TLS on TCP.

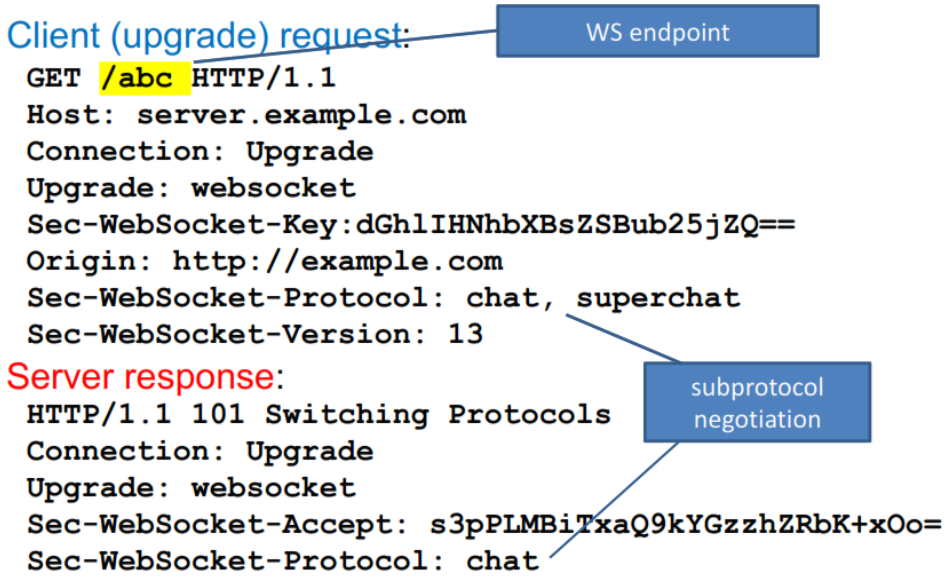

Figure 54: Handshake

11.3 Frame Types

WebSockets define different frame types for control and data:

- **Control Frames**: they are used for specific purposes like connection close and ping/pong operations.
- **Data Frames**: these carry the actual payload of the messages exchanged between the client and the server. They can be further categorized as binary frames, text frames (UTF-8 encoded), or continuation frames.

11.4 Messages and Closing

WebSockets handle messages as sequences of data frames, all of the same type. There are two types of messages:

- **Unfragmented messages**: they are transmitted as a single data frame, so there is the `FIN` bit set to `1` in the single frame.
- **Fragmented messages**: they are divided into multiple frames for transmission, so only the `FIN` bit of the last frame is set to `1`.

Either endpoint can initiate a closing handshake in WebSockets. Once a close frame is sent, no more data frames are exchanged. The other endpoint responds with a close frame if it has not already sent one. After the closing handshake, the underlying TCP connection is closed.

11.5 Using WebSockets

WebSockets provide the basic infrastructure for data transmission, but applications using WebSockets often implement higher-level protocols on top of them. These higher-level protocols define message types, metadata, and procedures for more specific application requirements. Subprotocols, which are named protocols built upon WebSockets, can also be used to provide additional functionality and structure to the WebSocket communication. Following is described the web socket programming in Javascript[3]:

[3]In these notes will not be discussed any Web Sockets Programming, for that look at slides.

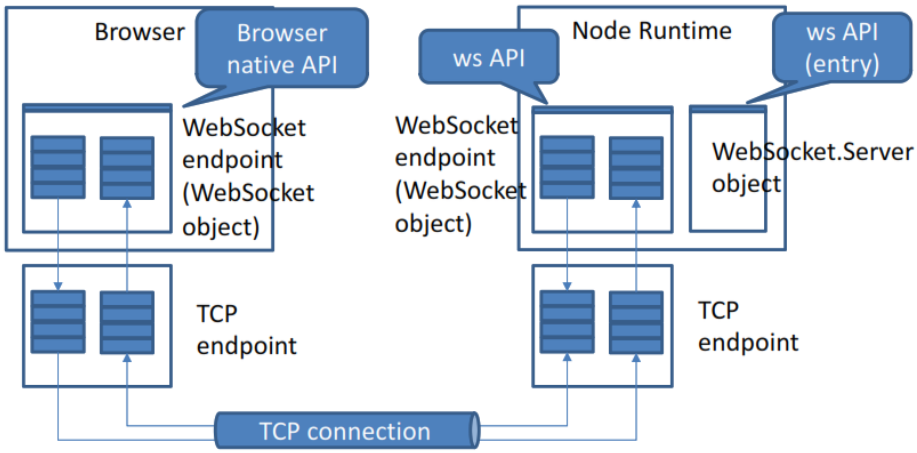

Figure 55: Web Socket Programming with Javascript

12 TCP/IP Sockets

TCP/IP sockets are a method of communication between two devices or programs over a network. A socket is essentially a combination of an IP address and a port number, and it represents one endpoint in a network communication.
TCP/IP sockets are used extensively in client-server applications, where a client program communicates with a server program over a network. The client and server each create a socket and use it to establish a connection between them. Once the connection is established, the two programs can exchange data. There are, mainly, two types of sockets in TCP/IP:

- **Stream sockets**: they provide a reliable, ordered, and error-checked flow of data between the two endpoints.
- **Datagram Sockets**: provide an unreliable, unordered, and unchecked delivery of data.

Additionally, there is one more type called **Raw Sockets**, which provides direct access to the services provided by layer 2-3 protocols.
In order to establish a TCP/IP socket connection, the client and server must follow a series of steps, known as the "*handshake*". This involves the exchange of packets between the two endpoints to establish and confirm the connection. Once the connection is established, data can be sent and received using the socket.
TCP/IP sockets are used in a wide variety of applications, from web browsing to email to file transfer. They are an essential component of network communication and allow programs to communicate and exchange data over the internet.

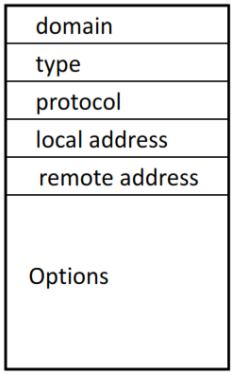

Figure 56: Data Structure Associated with a Generic Socket

When using TCP sockets, the endpoints involved in the communication can take different states or forms depending on their role in the communication process. TCP sockets are widely used for reliable, bidirectional, and stream-oriented communication.

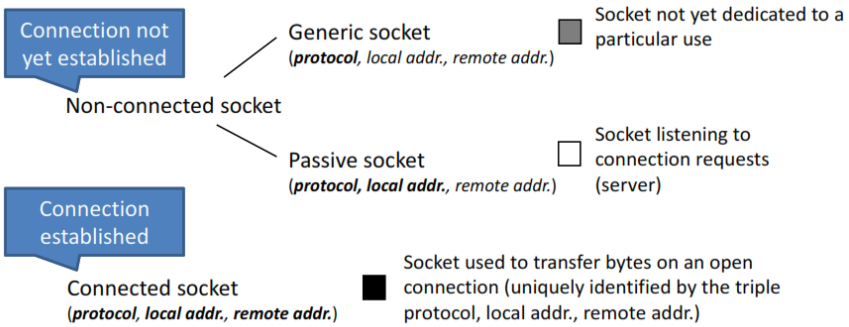

Figure 57: Sockets different roles

When we talk about roles, and sockets interaction, we should also deal with the different phases of the connection. It is possible to split the interaction in three main phases:

- **Handshake**: in this phase client and server perform different actions, such as:
  - Allocate local resources for communication.
  - Specify endpoints.
  - Open a connection (client-side).
  - Wait for the establishment of a connection (server-side).
- **Message Exchange**: it is the next and longer phase in which:
  - Data are sent and received on a connection (including urgent data).
  - Client and Server are notified when data arrive.
- **Closing**: last phase used for:
  - Gracefully terminate a connection or abort a connection.
  - Respond to graceful termination requests and abort conditions.
  - Release resources when a connection terminates

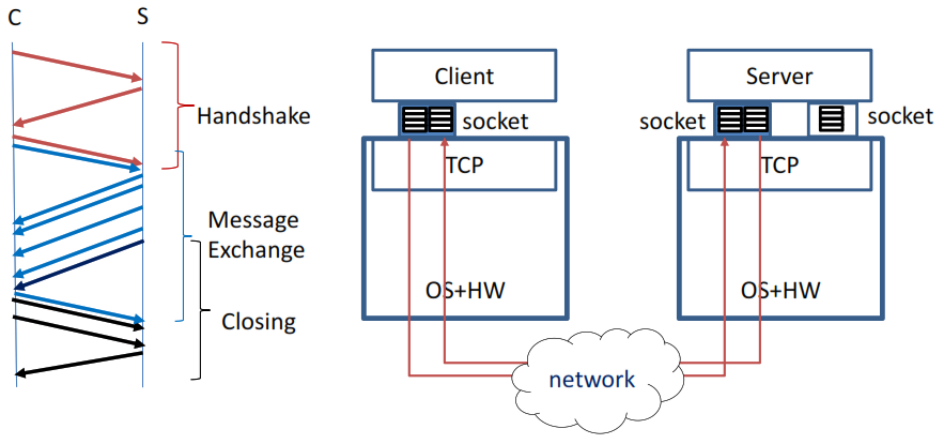

Figure 58: Connected and Passive Sockets

13 Fault Tolerance

Fault tolerance in distributed systems refers to the ability of a system to continue delivering services even in the presence of partial failures. The goal is to ensure that the system can automatically recover from failures, detect failures, and continue operation with minimal impact on overall performance.

13.1 Faults, Errors, and Failures

In fault tolerance, the terms *fault*, *error*, and *failure* are commonly used. A fault refers to a defect or deviation from the expected behavior of a system component and can be transient, intermittent or permanent. When a fault manifests itself and affects the system's behavior, it is called an error. If an error leads to the system's inability to perform its required function, it is considered a failure. Failures can be categorized into various types, including crashes, omissions, timing errors, incorrect responses, and arbitrary (byzantine) behavior.

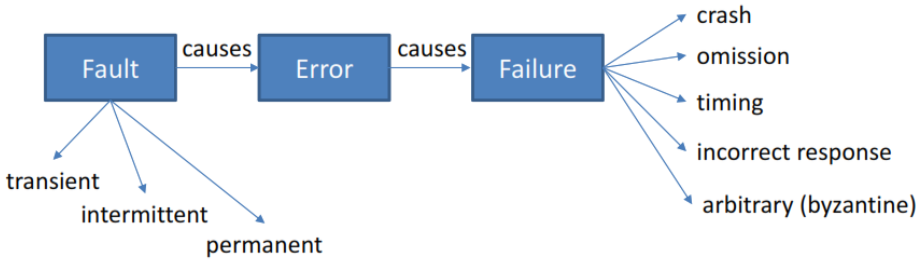

Figure 59: Main Fault Tolerance Concepts

Dependability is a key concept in fault tolerance and encompasses several attributes, including availability, reliability, safety, security, and maintainability. These attributes are crucial for building systems that can withstand faults and ensure continuous service delivery. Dependability can be achieved through a combination of fault prevention, fault tolerance, fault removal, and fault forecasting techniques.

More specifically, fault tolerance in distributed systems refers to the system's ability to continue delivering services even in the presence of partial failures, but it also aims to ensure that the overall performance is not seriously affected and that the system can automatically recover from these failures.
To achieve fault tolerance, the following measures need to be taken:

- **Failure Detection:** Faults and failures need to be detected in the system. Various techniques, such as monitoring, heartbeat mechanisms, or timeout mechanisms, can be employed to identify failures.
- **Recovery Procedures:** Once a failure is detected, appropriate recovery procedures must be implemented. These procedures can include restarting failed components, reconfiguring the system, or activating backup resources.
- **Continued Operation:** The system should continue its operation even in the presence of partial failures while waiting for repairs or recovery. This ensures that the system can still deliver services, although possibly with some degradation in performance.

By ensuring fault tolerance, distributed systems can maintain their availability and reliability, allowing them to withstand failures and continue delivering services to users.

Failure Type	Detection Techniques	Recovery/Repair Techniques
crash	timeout	replacement, restart, redundancy
omission	timeout	redundancy (including retransmission)
timing	timeout	redundancy
incorrect response	input validation /comparison	redundancy
arbitrary	input validation /comparison	redundancy

Figure 60: Detection, Recovery and Repair Techniques

13.2 Redundancy Techniques

Redundancy is a fundamental approach in fault tolerance. It involves the use of redundant components or processes to ensure system resilience. Time redundancy is achieved by retransmitting messages or redoing failed operations. Physical redundancy, on the other hand, involves adding more equipment, processes, or data copies. One common solution for physical redundancy is process replication, where multiple copies of a process are maintained.

13.3 Detecting Process Crash

Detecting process crashes in distributed systems can be challenging due to the asynchronous nature of the system and the absence of upper bounds on message delays. A process can detect the crash failure of another process by timing out when no data arrives from that process. However, crash detections may be erroneous or delayed. Different types of crash failures, such as fail-stop failures, fail-silent failures, fail-safe failures, and fail-arbitrary failures, have varying levels of observability and severity.

13.3.1 Example: Detecting Crash of Process Connected via TCP

In a system where processes are connected via TCP (Transmission Control Protocol), the TCP layer itself can detect when a connection has been closed or reset by the peer. A process can be informed about these conditions through the Socket API when reading or writing. However, in scenarios such as host crashes or permanent network disconnections, the detection of process crashes can be more complex and may require additional timeout mechanisms.

	Read	Write
connection closed	returns -1	successful
connection reset	throws SocketException	throws SocketException

Figure 61: Example: Detecting Crash of Process Connected via TCP

13.4 Dependability through Redundancy: Process Groups

To achieve fault tolerance, process groups can be used to create a dependable system. A process group refers to a collection of processes perceived as a single entity. Implementing a process group requires distributed algorithms and protocols for managing the group's membership, leadership, and data replication. These protocols aim to make the process group fault-tolerant and resilient to failures. Process groups can be organized in hierarchical or flat structures. In hierarchical organizations, primary-based protocols are commonly used, where one process acts as the primary coordinator. In flat organizations, replicated-write protocols, active replication, or quorum-based protocols are employed to ensure fault tolerance and consistency within the group.

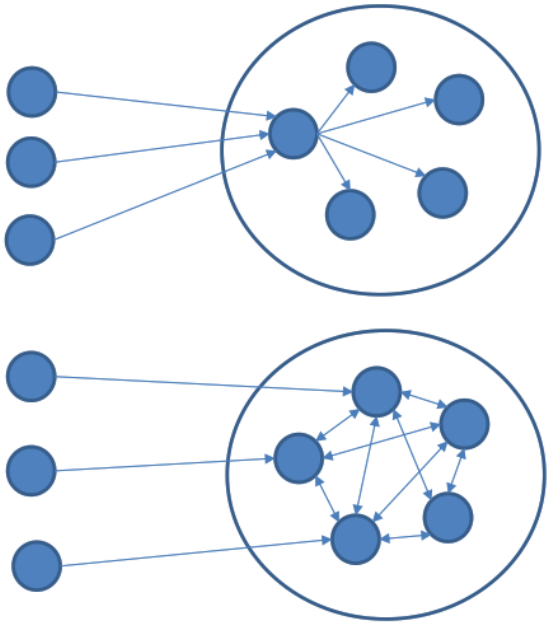

Figure 62: Process Group Organizations

13.5 Obtaining the Desired Fault-Tolerance

The desired level of fault tolerance in a process group depends on the number of faults the group can withstand. A group is considered k-fault tolerant if it can tolerate faults in up to k members. The choice of consensus protocols and the total number of processes required to achieve k-fault tolerance depend on the types of failures that are possible in the system.

Failure type	#Processes needed for k-fault tolerance	Consensus protocol Examples
crash, omission, timing, network	k+1	flooding
+incorrect response	2k+1	voting, Paxos
+arbitrary (byzantine)	3k+1	PBFT, Blockchain

Figure 63: Obtaining The Desired Fault-Tolerance

13.6 The CAP Theorem

The **CAP** (***Consistency***, ***Availability***, ***Partition Tolerance***) theorem states that in a distributed system, it is impossible to simultaneously achieve all three properties: *consistency*, *availability*, and *partition tolerance*. However, it's important to note that the CAP theorem is a theoretical result and not an absolute rule that applies to all distributed systems in every scenario.
The reason why the CAP theorem has been a topic of debate and criticism is that it presents a simplified model of distributed systems, making certain assumptions and trade-offs. Here are a few reasons why some argue that the CAP theorem is not entirely accurate:

- **Oversimplification**: The CAP theorem oversimplifies the complexities of distributed systems by considering only three properties. Real-world distributed systems often have additional dimensions, such as latency, scalability, durability, and performance, which are not explicitly addressed by the CAP theorem.

- **Real-world Variations**: While the CAP theorem assumes that network partitions are possible, it does not consider the variations and severity of partitions that can occur in real-world scenarios. In practice, different systems may handle partitions differently based on their specific requirements and trade-offs.
- **CAP as a Design Philosophy**: Critics argue that the CAP theorem is more of a design philosophy than a strict theorem. It encourages system designers to prioritize certain properties based on their application requirements, rather than considering them as mutually exclusive. In reality, designers often make trade-offs and employ techniques to achieve a balance between the CAP properties.
- **CAP in Distributed Databases**: The CAP theorem is commonly associated with distributed databases, and its application in this context has received significant scrutiny. Various distributed database systems, such as NewSQL and NoSQL databases, have challenged and extended the CAP theorem's assumptions, providing different consistency models and approaches to handle partitions.

It's important to consider that the CAP theorem provides a theoretical foundation for understanding the trade-offs in distributed systems, but it should not be viewed as an absolute rule governing all distributed systems. System designers and architects need to carefully evaluate their requirements, constraints, and desired properties when designing and implementing distributed systems.

14 MQTT

MQTT (MQ Telemetry Transport) is a client-server publish-subscribe messaging transport protocol, typically based on TCP. It is designed to be open, lightweight, and simple, making it suitable for constrained environments such as IoT (Internet of Things) and M2M (Machine-to-Machine) communication. MQTT prioritizes small code footprint, ensuring simplicity, and offers efficient communication over restricted network bandwidth and low-power devices. It also provides reliability and fault-tolerance features to handle unreliable communication channels characterized by wireless connections, noise, and frequent disconnections.

14.1 MQTT Architecture

MQTT follows a publish-subscribe architecture, which distinguishes it from traditional message queue systems. In this architecture, the broker delivers notifications synchronously to connected subscribers. However, MQTT also allows the option to use retention/persistency mechanisms for messages.

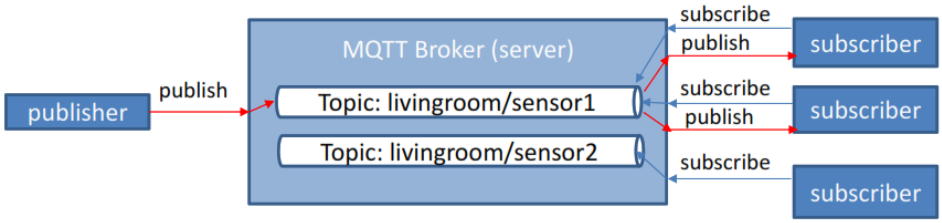

Figure 64: MQTT Architecture

14.2 MQTT Topics

In MQTT's publish-subscribe system, clients do not need to know each other directly. The addressable units in MQTT are the channels called topics. Clients can communicate with each other as long as they know the shared topics. MQTT topics are uniquely identified within a broker using hierarchical path-like names. For example, a topic could be named "`apartment1/livingroom/temperature`". Topics are not created or destroyed explicitly; they exist inherently, and clients simply need to reference them.

14.2.1 Special Topics

Topics starting with the symbol `$` are reserved and cannot be used by applications for their own purposes. Topics starting with `$SYS` provide system-related information. For instance:

- `$SYS/broker/version`
- `$SYS/broker/uptime`
- `$SYS/broker/clients/connected`
- `$SYS/broker/clients/disconnected`
- `$SYS/broker/clients/total`

14.2.2 Topic Filters

Topic filters are used to refer to collections of topics by employing wildcards. MQTT supports two types of wildcards:

- **Single-level wildcard**: represented by the "`+`" symbol. For example, "`apartment1/+/temperature`" matches topics like "`apartment1/bedroom/temperature`" and "`apartment1/livingroom/temperature`".

- **Multilevel wildcard**: represented by the "`#`" symbol, which must be at the end of the topic. For example, "`apartment1/#`" matches topics like "`apartment1/bedroom/temperature`", "`apartment1/livingroom/temperature`", and "`apartment1/kitchen/humidity`", capturing all subtopics under "`apartment1`".

14.3 The MQTT Protocol

Clients, both publishers, and subscribers, establish a connection with the MQTT broker using a layer-4 connection, typically TCP or TCP+TLS, which provides a bidirectional, ordered, and lossless stream of bytes. MQTT can be implemented over TCP or secure TCP (TLS) for enhanced security. As TCP connections may encounter failures, a client may experience temporary disconnection from the server.

As soon as the connection is established, an MQTT request/response handshake takes place, which includes the following steps:

- **Client identification and authentication**: The client provides identification information and authenticates itself to the broker, ensuring authorized access to the MQTT network.
- **Association with client session state, if any**: If the client has previously established a session with the broker, it can associate itself with the corresponding session state. This allows clients to maintain continuity across disconnections and resume communication seamlessly.
- **Possibility to set/negotiate client preferences**: During the handshake, clients have the opportunity to set and negotiate preferences with the broker. This includes aspects such as QoS levels, message persistence, and other protocol-specific parameters.

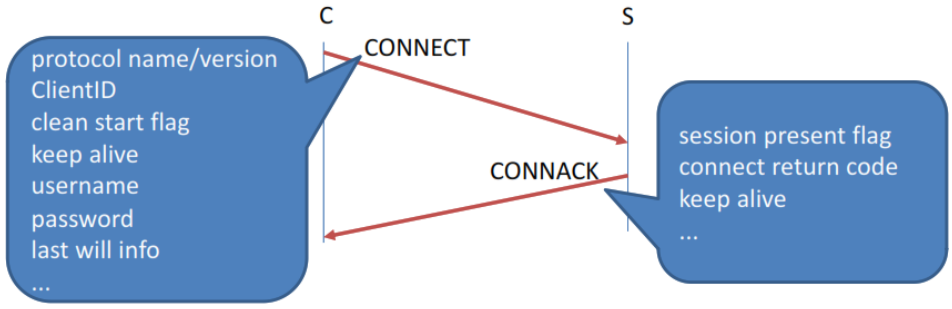

Figure 65: MQTT Handshake

14.4 Operations

After the initial handshake, a client can perform a sequence of operations by exchanging control packets with the broker. The operations supported by MQTT include:

- **Publish**: Clients can publish messages to a specific topic, which are then distributed to interested subscribers.
- **Subscribe**: Clients can subscribe to specific topics to receive messages published on those topics from other clients.
- **Unsubscribe**: Clients can unsubscribe from previously subscribed topics, no longer receiving messages published on those topics.
- **Ping**: Clients can send a Ping control packet to the broker to maintain the connection's vitality and detect any possible disconnections.

A client can terminate a session gracefully by sending a Disconnect control packet to the broker.

14.4.1 Publish Quality of Service (QoS)

The Publish operation in MQTT supports three Quality of Service (QoS) levels:

1. **QoS 0**: At most once delivery (best effort). Messages are sent with a fire-and-forget approach, providing no additional guarantees beyond what is ensured by the underlying TCP protocol.

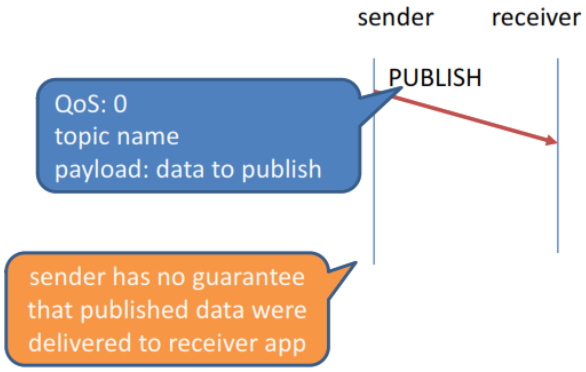

Figure 66: QoS 0

2. **QoS 1**: At least once delivery (may deliver multiple copies). The receiver acknowledges each message delivery, and the sender waits for acknowledgment. If acknowledgment is not received within a timeout period, the sender retransmits the message. The sender only forgets a message when it receives the acknowledgment.

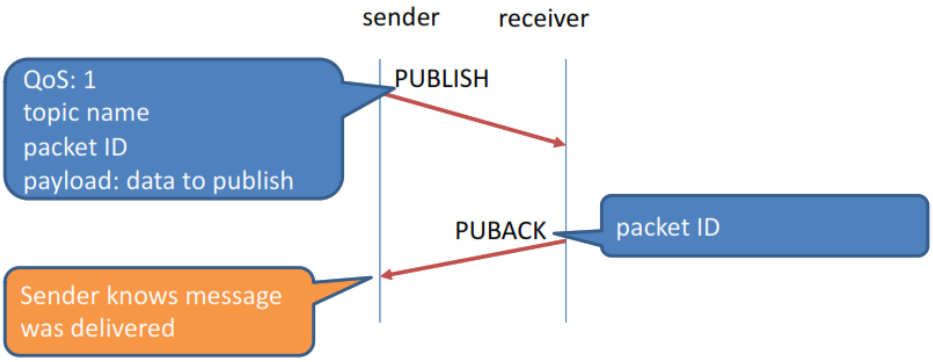

Figure 67: QoS 1

3. **QoS 2**: Exactly once delivery. This level includes delivery acknowledgment as in QoS 1, but also incorporates a mechanism to prevent duplicate messages. QoS 2 ensures message delivery exactly once, with no duplication. Both QoS 1 and QoS 2 levels have unbounded delays, meaning the delivery may experience arbitrary delays depending on the network conditions.

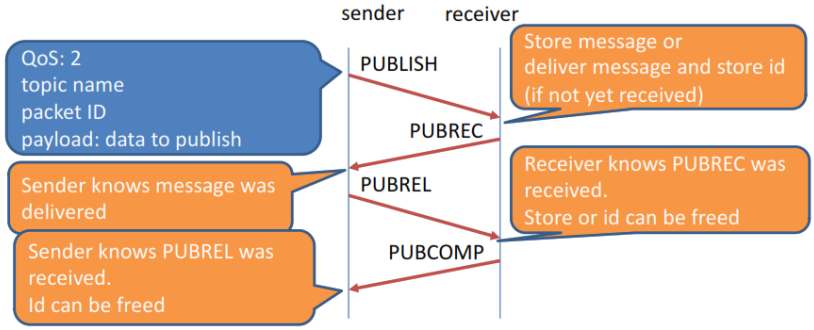

Figure 68: QoS 2

14.4.2 Subscription and Unsubscription

Subscription and unsubscription in MQTT involve the process of a client subscribing or unsubscribing to one or more topics using topic filters. Here is a description of the process:

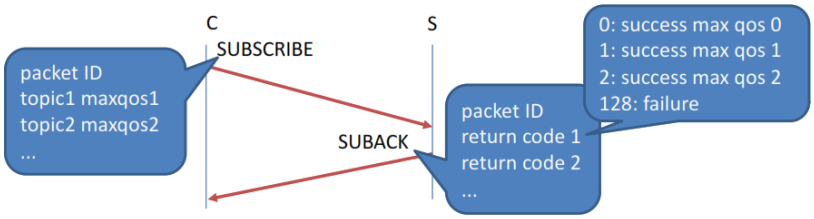

Figure 69: Subscription

- **Subscription**: When a client wants to receive messages published on specific topics, it sends a subscription request to the MQTT broker. The subscription request includes one or more topic filters that define the topics of interest. Each topic filter may also specify a maximum Quality of Service (QoS) level that the client desires for that particular topic.
- **Topic Filters**: Topic filters are used to define the set of topics to which the client wants to subscribe. The topic filters can include wildcard characters for flexible matching. For example, a topic filter like "home/+/temperature" would match topics such as "home/livingroom/temperature" and "home/bedroom/temperature", where the "+" wildcard represents a single level in the topic hierarchy.
- **Granting Subscriptions**: The MQTT broker receives the subscription request from the client and evaluates each topic filter. The broker has the ability to selectively grant subscriptions based on its policies. It can choose to accept or reject specific topic filters requested by the client. Additionally, the broker can modify the maximum QoS level for each topic filter, potentially downgrading it if needed.
- **Downgrading QoS**: In some cases, the broker may not be able to fulfill the client's requested maximum QoS level for a particular topic filter. For example, if the broker has limitations due to resource constraints or QoS prioritization, it can lower the maximum QoS level for that topic filter. This ensures that the client still receives messages on that topic but at a lower QoS level than initially requested.

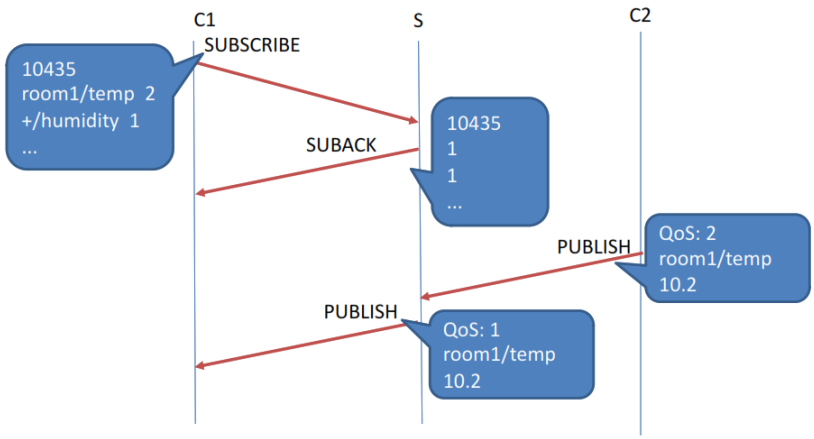

Figure 70: Example: QoS Downgrading

When a client no longer wishes to receive messages on specific topics, it can send an unsubscription request to the MQTT broker. The unsubscription request specifies the topics from which the client wants to unsubscribe.

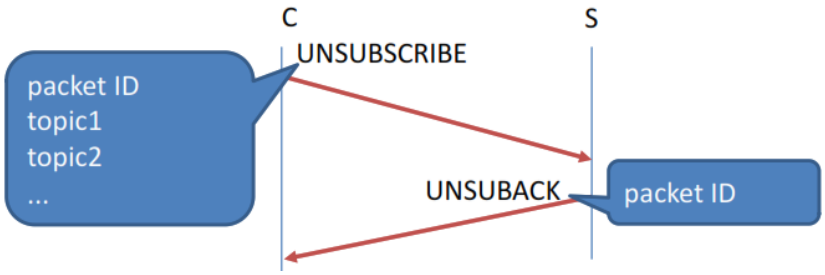

Figure 71: Unsubscription

By allowing the broker to selectively grant subscriptions and potentially downgrade the requested QoS level, MQTT provides flexibility in handling subscriptions based on the capabilities and limitations of the broker and the overall system.

Persistent Storage:

The MQTT broker maintains certain state information for disconnected subscribed clients, ensuring continuity of their sessions. This includes:

- Persistent sessions: The broker keeps track of session information for disconnected clients, allowing them to resume their subscriptions and QoS 1 and QoS 2 publish operations when they reconnect.
- RETAINED flag: Publishers can mark messages as retained, indicating that the last published value for a particular topic should be stored by the broker. When a client subscribes to a topic with a retained value, the broker immediately publishes the retained message to the client.
- Client state information: Clients also need to maintain their own state information for QoS 1 and QoS 2 publish operations to ensure reliable message delivery.

14.5 Last Will (Testament):

MQTT provides a mechanism called "Last Will" or "Testament" to handle ungraceful client disconnections. If a client unexpectedly disconnects, the broker can detect the disconnection and execute the client's last will. The Last Will feature includes the following fields when connecting to the broker:

- **Will topic**: Specifies the topic to which the last will message should be published.
- **Will message**: Defines the content of the last will message.
- **Will QoS**: Specifies the QoS level at which the last will message should be published.
- **Will retain flag**: Indicates whether the last will message should be retained by the broker.

14.6 MQTT over Websockets:

To enable browsers to act as MQTT clients, MQTT over Websockets was introduced. This feature allows MQTT communication to be carried over websockets, as browser APIs do not provide direct access to TCP. By leveraging websockets, browsers

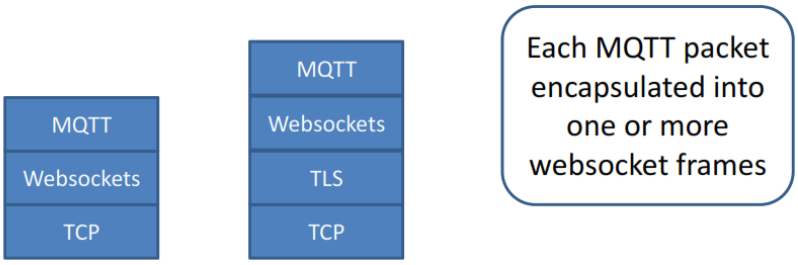

Figure 72: MQTT over Websockets

14.7 Pub/Sub Interface Design Guidelines

The Pub/Sub interface design is a messaging pattern that enables communication between components or systems in a decoupled manner. It follows a publish-subscribe model where publishers send messages (events or data) to a broker, and subscribers receive those messages based on their subscriptions. Basically the following things must be taken into account:

- **Granularity Level of Messages and Topics**: Fine granularity of topics can help reduce the load on the MQTT broker. The choice of granularity may depend on the desired level of interactivity in the application. However, it's important to note that high interactivity can come with additional costs.
- **How to return errors**: When handling errors in MQTT-based systems, it is recommended to use error topics and error messages. By defining specific topics and messages for error handling, it becomes easier to communicate and handle error conditions in a structured manner.
- **Idempotent operations**: Idempotent operations are particularly relevant when messages can produce state changes. Choosing appropriate topic semantics and designing messages in a way that allows them to be idempotent ensures that performing the operation multiple times has the same effect as performing it once.
- **Documentation**: It is essential to provide clear and comprehensive documentation for the MQTT-based system. This documentation should include information about message types, topic structure, and any other relevant details. Ideally, the topic structure should be self-explanatory, making it easier for developers to understand and use the system.

14.8 MQTT Specific Guidelines

Here are provided few MQTT guidelines:

- **Use plain ASCII in topic names**: It is recommended to use plain ASCII characters in MQTT topic names to ensure compatibility and interoperability across different systems and platforms.

- **Use topic names as identifiers to carry part of message** : MQTT topic names can be used as identifiers to convey relevant information about the message. By designing topic names appropriately, it is possible to provide additional context or metadata about the message without the need for additional fields or headers.
- **Leave topic structure open to extension**: It is advisable to design the topic structure in a flexible and extensible manner. This allows for future enhancements and additions to the system without significant changes to the existing topic structure. By keeping the topic structure open to extension, the system can adapt to evolving requirements and accommodate new features or functionalities.