Uni Distributed Systems Condensed Summary

Condensed summary for the distributed systems course at HdM Stuttgart

Felicitas Pojtinger

2023-02-05

Contents

1 Meta

1.1 Contributing

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

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

Figure 1: QR code to source repository

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

1.2 License

Figure 2: AGPL-3.0 license badge

Uni Distributed Systems Notes (c) 2023 Felicitas Pojtinger and contributors

SPDX-License-Identifier: AGPL-3.0

2 Introduction to Distributed Systems

2.1 Laws and Terms

2.1.1 Metcalfe's Law

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

Metcalfe's law is often cited as a factor that can contribute to the emergence of scale-free, or power law, distributions in networks. This type of distribution is characterized by a few nodes (or users) with many connections, and many nodes with only a few connections. The existence of network effects, in which the value of a network increases with the number of users, can help to explain why we don't see many Facebooks or Googles – it can be difficult for new networks to gain traction and achieve the same level of utility as established networks with a large user base.

2.1.2 Generalized Queuing Theory Terms (Henry Liu)

- **Server/Node**: A combination of a wait queue and a processing element
- **Initiator**: The entity that initiates a service request
- **Wait time**: The time a request or initiator spends waiting in line for service
- **Service time**: The time it takes for the processing element to complete a request
- **Arrival rate**: The rate at which requests arrive for service
- **Utilization**: The percentage of time the processing element spends servicing requests, as opposed to being idle
- **Queue length**: The total number of requests waiting and being serviced
- **Response time**: The sum of the wait time and service time for a single visit to the processing element
- **Residence time**: The total time spent by the processing element on a single transaction, if it is visited multiple times
- **Throughput**: The rate at which requests are serviced, or how fast requests can be processed without long wait times.

2.1.3 Little's Law

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

2.1.4 Hejunka

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

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

- **Setting limits** on the number of tasks or requests that can be processed at any given time
- **Balancing the workload** across different servers or processing elements
- **Prioritizing tasks** based on their importance or impact on the overall system
- Using techniques such as **batching or grouping** similar tasks together to reduce variability

By leveling the production process and reducing variability in task sizes, Hejunka can help to improve the efficiency and flow of work through a system, and reduce the risk of bottlenecks or delays caused by large differences in task size.

2.1.5 Amdahl's Law

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

$$speedup = \frac{1}{(1 - parallelfraction + \frac{parallelfraction}{numberofprocessors}}$$

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

Amdahl's Law is often used to understand the potential benefits and limitations of optimizing or improving specific parts of a system. It can be a useful tool for determining how much resources should be invested in improving a particular part of the system, and for understanding the potential impact of those improvements on overall system performance.

2.2 Process and I/O Models

2.2.1 Different Process Models

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

2.2.2 Questions for Process Models

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

2.2.3 Different I/O Models

- **Synchronous Blocking (Java before NIO/AIO)**: Prior to the introduction of the Java New I/O (NIO) and Asynchronous I/O (AIO) APIs, Java had a different model for handling input/output

(I/O) operations. This model involved using threads to block and wait for I/O operations to complete, which could be inefficient and consume a lot of system resources.
- **Synchronous Non-Blocking (Polling pattern)**: The polling pattern is a way of handling I/O operations in which a central component periodically checks for the completion of I/O operations. This can be done by repeatedly calling a function that checks the status of the operation, or by using a timer to trigger the check at regular intervals.
- **Asynchronous Blocking (Reactor pattern)**: The Reactor pattern is a way of handling I/O operations in which a central component is notified when an I/O operation is completed, rather than periodically checking for its completion. This can be more efficient than the polling pattern, as it allows the system to respond to I/O operations as soon as they are completed, rather than waiting for a periodic check.
- **Asynchronous Non-Blocking (Proactor pattern)**: The Proactor pattern is similar to the Reactor pattern, but it is designed to handle high-concurrency environments where many I/O operations are occurring simultaneously. It uses a combination of asynchronous I/O and event-driven design to allow for efficient handling of multiple I/O operations at once.

2.2.4 Questions for I/O Models

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

3 Message Protocols

3.1 Delivery Guarantees

3.1.1 The Role of Delivery Guarantees

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

- **TCP Communication properties**: TCP (Transmission Control Protocol) is a networking protocol that is used to establish and maintain communication between devices over a network. It

has several key properties that are relevant to the scenario described, including reliability, flow control, and congestion control.

- **At-least-once**: The "at-least-once" delivery guarantee means that a message may be delivered more than once, but it will always be delivered at least once. This can be useful in situations where it is more important to ensure that a message is delivered, even if it may be duplicated, than it is to prevent duplicates from occurring.
- **At-most-once**: The "at-most-once" delivery guarantee means that a message will be delivered at most once. This can be useful in situations where it is more important to prevent duplicates from occurring than it is to ensure that a message is always delivered.
- **Exactly once**: The "exactly once" delivery guarantee means that a message will be delivered exactly once, with no duplicates. This can be more difficult to achieve than the other delivery guarantees, as it requires additional complexity and overhead to ensure that duplicates are prevented.
- **Message complexity**: The number of messages sent refers to the total number of messages that are transmitted as part of the communication process. In the scenario described, the number of messages sent may affect the efficiency and reliability of the communication process, and may need to be taken into account when determining the appropriate delivery guarantee to use.

3.1.2 Why is TCP not Enough?

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

- **Lost messages retransmitted**: TCP includes mechanisms for retransmitting lost messages, which can help to improve the reliability of communication. However, if messages are frequently lost or the network is particularly unreliable, the overhead of retransmitting lost messages may become a burden on the system.
- **Re-sequencing of out of order messages**: TCP includes mechanisms for reordering out-of-order messages, which can help to ensure that messages are delivered in the correct order. However, if messages are frequently delivered out of order, this can be inefficient and may cause issues with the overall communication process.
- **Sender choke back (flow control)**: TCP includes flow control mechanisms that allow the sender to adjust its rate of transmission based on the capacity of the receiver. However, if the sender is sending messages too quickly, this can lead to congestion on the network and reduced performance.
- **No message boundary protection**: TCP does not provide any protection for message boundaries, which means that messages may be broken up or combined during transmission. This can

make it difficult to ensure that messages are delivered in their entirety and can cause issues with the overall communication process.
- **Timeout problem**: TCP includes mechanisms for detecting and handling connection failures, but these mechanisms may not be sufficient in all situations. For example, if a connection is lost due to a timeout, it may take some time to detect and recover from the failure, which can lead to delays and disruptions in the communication process.

To address these and other issues, it may be necessary to use additional protocols or techniques, such as message-oriented middleware or application-level protocols, to ensure proper behavior in case of connection failures and to provide additional features and guarantees for message delivery.

3.1.3 Different Levels of Timeouts

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

3.1.4 Delivery Guarantees for RPCs

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

- **Once and only once/exactly once**: The “once and only once” or “exactly once” delivery guarantee means that a request will be delivered exactly once, with no duplicates. This can be more difficult to achieve than the other delivery guarantees, as it requires additional complexity and overhead to ensure that duplicates are prevented.

3.2 Idempotency

3.2.1 Overview

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

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

3.2.2 Server State and Idempotency

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

- **No need to remember a request and its result**: If an operation or request is idempotent, the server does not need to remember the request or its result. This can be useful in situations where the server’s storage is limited or unreliable, as it means that the server does not need to maintain a record of all previous requests and their results.
- **Server can lose its storage**: If the server’s storage is lost or becomes unavailable, it should not affect the overall result of the operation or request, as long as the operation or request is idempotent. This can help to ensure that the operation or request is able to be safely repeated even if the server’s storage is lost.
- **Concurrent updates might be consistent without concurrency control**: If an operation or request is idempotent, it is possible that concurrent updates to the same data may be consistent without the need for concurrency control mechanisms such as locks or transactions. This can help to improve the efficiency and performance of the system.

3.2.3 Implementing Delivery Guarantees for Idempotent Requests

- **"At least once"** implementation for idempotent requests: For idempotent requests, the "at least once" delivery guarantee can be implemented by simply sending an acknowledgement (ack) to the client after the request has been received. This approach does not require any updates to the server state, and is suitable for requests that do not have any critical side effects.
- **"At most once"** implementation for nonidempotent requests: For nonidempotent requests, the "at most once" delivery guarantee can be implemented by storing a response on the server until the client confirms that it has been received. This approach requires the server to maintain state for each response, and may involve adding a request number to each request to help the server detect and discard duplicate requests.
- **"Exactly once"** implementation: The "exactly once" delivery guarantee is not possible to achieve in asynchronous systems with network failures. However, it can be approximated using techniques such as two-phase commit and epoch numbers, which allow the client and server to coordinate their actions and ensure that they do not forget their decisions. This approach may involve maintaining an atomic log on both the client and server, and storing responses on the server until they are confirmed by the client

3.2.4 Repeating Non-Idempotent Operations

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

- **Use a message ID** to filter for duplicate sends: By including a unique message ID in each request, the server can filter out duplicates and only execute the request once.
- **Keep a history list of request execution results** on the server: If the reply to a request is lost, the server can retransmit the result from its history list. This helps to ensure that the client receives the correct result even if the initial reply was lost.
- **Lease resources on the server**: In some cases, it may be necessary to keep state on the server in order to facilitate communication. For example, a client may "lease" resources on the server for a specific period of time. This can help to ensure that resources are used efficiently and released when they are no longer needed.

It is important to carefully manage state on the server in order to avoid overloading the system with old replies. It may be necessary to set limits on how long to store old replies and when to discard them.

3.3 Order

3.3.1 Request Order in Multi-Point-Protocols

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

3.3.2 Request Ordering with Multiple Nodes

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

- **Reliable Broadcast**: Reliable broadcast is a protocol that ensures that a message is delivered to all nodes in the system, and that it is delivered in the same order to all nodes. This can help to ensure that requests are processed in the correct order, even if they are sent from different nodes or if there are delays or other issues with the network.
- **FIFO Cast**: FIFO cast is a protocol that ensures that messages are delivered in the order in which they were sent, with the first message sent being the first one to be delivered. This can help to ensure that requests are processed in the correct order, even if they are sent from different nodes or if there are delays or other issues with the network. This can still lead to causal inconsistencies!
- **Causal Cast**: Causal cast is a protocol that ensures that messages are delivered in a causally consistent order, based on the dependencies between the messages. This can help to ensure that requests are processed in the correct order, even if they are sent from different nodes or if there are delays or other issues with the network.
- **Absolutely Ordered Casts**: Absolutely ordered casts is a protocol that ensures that messages are delivered in a totally ordered sequence, with no uncertainty about the order in which the

messages were sent. This can help to ensure that requests are processed in the correct order, even if they are sent from different nodes or if there are delays or other issues with the network.

3.3.3 Implementing Causal Ordered Broadcasts

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

3.3.4 Implementing Absolutely Ordered Casts

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

There are however disadvantages with these implementations:

- **May have high overhead**: This solution may have high overhead, as it requires all nodes to send and receive messages from every other node in the system. This can be particularly problematic

in large systems with many nodes, as it may result in many messages being transmitted and processed.
- **May have high latency**: This solution may also have high latency, as it requires all nodes to wait for the total order to be determined before delivering their messages. This can be particularly problematic in systems where low latency is critical.

3.3.5 Reliable Messaging

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

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

SOAP and HTTP partially achieve this like so:

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

3.3.6 Transaction Models

Atomic transactions:

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

Activity transactions:

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

4 Theoretical Foundations of Distributed Systems

4.1 Foundational Concepts

4.1.1 The Eight Fallacies of Distributed Computing

- **The network is reliable**: This fallacy assumes that the network will always be available and free of errors or failures, which is not always the case.
- **Latency is zero**: This fallacy assumes that communication between nodes in a distributed system will be instantaneous, but in reality, there is always some latency due to the time it takes for a request to be processed and a response to be received.
- **Bandwidth is infinite**: This fallacy assumes that there is unlimited bandwidth available for communication between nodes, but in reality, bandwidth can be limited by factors such as network congestion or hardware limitations.
- **The network is secure**: This fallacy assumes that the network is completely secure and free from threats such as hackers or malicious software, but this is not always the case.
- **Topology doesn't change**: This fallacy assumes that the topology of the network, or the way that nodes are connected, will remain constant, but in reality, the topology can change due to factors such as node failures or changes in network configuration.
- **There is one administrator**: This fallacy assumes that there is only one person or entity responsible for managing the network, but in distributed systems, there may be multiple administrators or stakeholders with different roles and responsibilities.
- **Transport cost is zero**: This fallacy assumes that there are no costs associated with communication between nodes, but in reality, there may be costs such as network fees or hardware expenses.
- **The network is homogeneous**: This fallacy assumes that all nodes in the network are identical and operate in the same way, but in reality, nodes can have different hardware, software, and configurations.

4.1.2 Analyzing Latency

- **Know the long-term trends in hardware**: Latency can be influenced by the performance of the hardware being used, so it's important to be aware of trends in hardware development and how they may impact latency.
- **Understand the problem of deep queuing networks and the solutions**: Deep queuing occurs when there are many requests waiting to be processed, leading to longer latency. Understanding this problem and implementing solutions such as load balancing can help reduce latency.

- **Know your numbers with respect to switching times, router delays, round-trip times, IOPS per device, and perform "back of the envelope" calculations**: It's important to have a good understanding of the specific numbers and metrics related to latency in your system, such as switching times and router delays, and to perform calculations to estimate the impact of these factors on latency.
- **Understand buffering effects on latency**: Buffering, or the temporary storage of data, can impact latency by adding additional processing time. Understanding the effects of buffering on latency can help you optimize your system to minimize this impact.
- **Include the client side in your calculations**: Latency is often impacted by factors on the client side, such as the client's hardware and network connection. It's important to consider these factors when calculating latency and optimizing your system to minimize it.

4.1.3 Characteristics of Distributed Systems

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

4.2 Consistency

4.2.1 Liveness vs. Correctness

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

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

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

4.2.2 Liveness and Correctness in Practice

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

Correctness:

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

Liveness:

- **Start receiving notifications some time after a subscription was made**: This ensures that a node will eventually start receiving notifications after it has subscribed to them.

Failure Assumptions: Fail-stop model with fairness

In this example, the system is designed to ensure correctness by limiting the notifications a node receives to those it is subscribed to and ensuring that notifications are received only once. It is designed to ensure liveness by ensuring that a node will eventually start receiving notifications after subscribing. The system also makes assumptions about failures, such as the fail-stop model with fairness, which are used to ensure the correctness and liveness of the system.

4.2.3 Timing Models

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

- **Synchronous timing models**: In synchronous timing models, transmit times are strictly defined and events happen at specific moments. Nodes in a synchronous system can immediately detect when another node has crashed because the system relies on a clock to synchronize events. Examples of synchronous systems include CPUs and other types of hardware.

- **Asynchronous timing models**: In asynchronous timing models, there is no exact time between sending and receiving messages. Messages will "eventually" arrive, but there is no guarantee about when. Because there are no strict timing constraints, a node in an asynchronous system cannot tell whether another node has crashed or is simply very slow to respond. There are no timeouts in asynchronous systems because they would require a clock.
- **Partial synchronous timing models**: Partial synchronous timing models are a combination of synchronous and asynchronous models. These systems are asynchronous, but they are enhanced with local clocks that provide some level of synchronization. This is the model that is typically used for real-world distributed systems, as it allows for the flexibility of asynchronous communication while still providing some guarantees about timing.

4.2.4 The Fischer, Lynch and Patterson Result

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

4.3 The CAP Theorem

4.3.1 Overview

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

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

4.3.2 Preconditions for the CAP Theorem

- To be considered consistent, the system must ensure that **all operations are atomic and linearizable**, meaning that they can be thought of as occurring at a single instant in time and have

a total order.
- For a system to be considered available, it must **ensure that every request received** by a non-failing node **is met with a response**, and that every request terminates.
- Partition tolerance: The network will be **allowed to lose arbitrarily many messages** sent from one node to another.

4.3.3 Common Misconceptions of the CAP Theorem

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

4.3.4 The Modern View of the CAP Theorem

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

4.3.5 PACELC

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

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

4.4 Failure

4.4.1 Technical Failures

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

Unfortunately, in most cases there is no failure detection service that can identify when these failures occur and allow others to take appropriate action. However, it is possible to develop a **failure detection service** that could detect even partitioning and other types of failures through the use of MIBs (Management Information Bases) and triangulation. Applications could also be designed to track themselves and restart if necessary.

4.4.2 Failure Types

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

4.4.3 Failure Models

- **Crash-stop**: A process crashes atomically and stays down.
- **Crash-stop with recovery**: A process crashes and is down until it begins recovery, and is up again until the next crash occurs. For consensus, at least $2f+1$ machines are needed (quorum).

- **Crash-amnesia**: A process crashes and restarts without recollection of previous events or data.
- **Failstop**: A machine fails completely, and the failure is reported to other machines reliably.
- **Omission errors**: Processes fail to send or receive messages even though they are alive.
- **Byzantine errors**: Machines or parts of machines, networks, or applications fail in unpredictable ways and may recover partially. For consensus, at least $3f+1$ machines are needed.

Many protocols for achieving consistency and availability **make assumptions about failure models**. For example, transaction protocols may assume recovery behavior by its participants if the protocol terminates.

4.4.4 Failures and Timeouts

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

4.4.5 Failure Detectors

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

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

4.5 Clocks

4.5.1 Time in Distributed Systems

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

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

Logical time is modeled as partially ordered events within a process or between processes. It is used to **represent the relative order of events** in a distributed system, **rather than the actual clock time** at which they occurred.

Causal meta-data in the system can also order the events properly.

4.5.2 Consistent vs. Inconsistent Cuts

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

4.5.3 Event Clocks (Logical Clocks)

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

4.5.4 Lamport Logical Clock

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

4.5.5 Vector Clocks

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

4.5.6 Physical Interval Time (TrueTime)

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

4.5.7 Hybrid Clocks

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

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

4.5.8 Ordering in Distributed Event Systems

- **FIFO (first-in, first-out) ordering**: This refers to the requirement that a component must receive notifications in the order in which they were published by the publisher.

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

4.6 Consensus

4.6.1 Overview

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

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

4.6.2 Algorithms for Consensus

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

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

- **Dynamic Group Membership**: This class of algorithms is used to achieve consensus in systems with a dynamic membership. These algorithms include virtual synchrony and multicast-based approaches.
- **Gossip Protocols**: These algorithms are used to disseminate information between processes in a distributed system. They can be used to achieve consensus by allowing processes to exchange information and reach agreement on a decision.

4.6.3 Two-Phase Commit (2PC)

Steps:

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

Example:

1. Imagine that there are three processes in a distributed system: A, B, and C.
2. The coordinator process is A.
3. The processes are deciding whether to commit to a new software update.
4. In the **preparation phase**, A sends a "prepare to commit" message to B and C.
5. B and C send "prepare to commit" messages back to A.
6. In the **decision phase**, A sends a "commit" message to B and C.
7. In the **confirmation phase**, B and C send "commit confirmation" messages back to A.
8. In the **finalization phase**, A sends a "commit complete" message to B and C, indicating that the decision to commit to the software update has been successfully made.

Liveness and Correctness:

- The two-phase commit protocol (2PC) allows for **atomic (linearizable) updates** to be made by participating processes.

- 2PC is considered **relatively expensive** in terms of latency due to the requirement for a persistent log at each participant.
- A **crash failure** of the coordinator or a participant **can halt or disrupt** the execution of 2PC.
- If everything goes as planned, 2PC has a **clear and easy-to-understand** semantic for application developers.

4.6.4 Quorum-Based Consensus

Steps:

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

Example:

1. A group of five processes in a distributed system (A, B, C, D, and E) are **deciding whether to commit** to a new software update.
2. **Process A proposes the decision** to update the software.
3. **Processes B, C, D, and E all vote** on the proposed decision.
4. The **votes are counted and checked against the quorum requirement**, which is set at three votes.
5. Since a quorum of three votes has been received in favor of the decision, it is considered to **have been approved**.
6. The processes move forward with **implementing the software update**.

Liveness and Correctness:

- Quorum-based consensus protocols (QP) allow for **atomic (linearizable) updates** to be made by participating processes, but this depends on the type of implementation being used (e.g., whether read quoras are used).
- QP are known to be **relatively expensive in terms of latency**, as even simple reads may require a quorum for consistency. To improve performance, **many QP systems use a leader-based approach**, which involves routing client requests through a designated leader process.
- **Leader crashes** are usually **handled through the use of leases**, which can cause delays.
- In the case of a crash failure, **QP can continue as long as a quorum is still possible**.

- Network partitions may cause the system to **either** respond with all non-failing nodes (which may **sacrifice consistency**) or to stop responding to requests from minority nodes (which may **sacrifice availability**).
- QP offer a **reliable and efficient** method for achieving consensus in distributed systems, but they also come with some trade-offs in terms of performance and fault tolerance.

4.6.5 Raft

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

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

1. **Leader Election**:

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

2. **Log Replication**:

- The **Leader sends AppendEntries messages** to all other replicas to replicate its log.
- If a Follower's log is **missing an entry** preceding the one in the Leader's message, it **responds with a missing entry error**.
- If the Follower's log is up-to-date and the Leader's entry is valid, the **Follower appends the entry** and **responds with a success message**.
- If the **Leader receives a success message** from a quorum of replicas, it updates its commit index and **sends a Commit message** to all other replicas to apply the committed entry to their state machines.

3. **State Machine Update**:

- If a **replica receives a Commit message**, it applies the committed entry to its state machine and **responds with an Apply message** to the Leader.
- If the **Leader receives an Apply message** from a quorum of replicas, it updates its commit index and **sends an Apply message to all other replicas**.

4.7 Broadcasting

4.7.1 Atomic Broadcast Conditions

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

It satisfies the following conditions:

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

It is widely used in distributed computing for group communication and defined as a reliable broadcast that **satisfies total order**.

4.7.2 Atomic Broadcast Protocol

Data:

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

Phases:

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

- **Frequent leader changes can cause overhead** and may be a potential denial of service. It is important to consider latency on the leader node when implementing an atomic broadcast protocol.
- **Paxos, Raft etc. are Atomic Broadcast Protocols!**

4.7.3 Gossip Protocols

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

There are several key characteristics of gossip protocols:

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

4.7.4 DWAL

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

4.7.5 Design Components of DWALs

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

- **Message order and latency hiding**: To ensure that the DWAL can function effectively, it is important to ensure that updates are delivered to all processes in a consistent order.

4.8 Replication

4.8.1 Properties of Replication Models

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

4.8.2 Single-Leader Replication

Steps:

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

Advantages:

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

Disadvantages:

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

4.8.3 Eventually Consistent Reads

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

Steps:

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

4.8.4 Multi-Master Replication

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

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

Steps:

1. Client 1 writes `x=5` to Node 1 (master)
2. Client 2 writes `x=10` to Node 2 (master)
3. The masters detect a conflict

Conflict types:

- **Update conflict**: Two identical rows changed on two servers
- **Uniqueness conflict**: Two identical primary (uniqe) keys added in same table on two servers
- **Delete conflict**: During delete of a row the same row is changed on a different server.

4.8.5 Leaderless Quorum Replication

Write:

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

Read:

- Some **systems may detect inconsistencies** during a read operation.
- These systems can **either automatically perform a cleanup** (e.g. using version numbers to return the correct value) or **offer both values for the client** to choose from.

4.8.6 Session Modes of Asynchronous Replication

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

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

We can also derive **session anomalies** from this:

- **Non-monotonic reads**: Reads that do not return increasingly up-to-date contents.
- **Non-monotonic writes**: Write operations that are not accepted in the order that they were made by the same user.
- **Non-monotonic transactions**: Transactions that do not preserve the order in which they were made by the same user.
- **Not-reading-my-writes**: Reads that do not include previous writes by the same user.

4.8.7 Global Modes of Replication

There are multiple different modes to choose from:

- **Strong consistency** ensures that **all previous writes are visible**, and is characterized by the following properties:
  - Ordered: Writes are accepted in the order that they were made.
  - Real: All writes are visible.
  - Monotonic: Write operations are accepted only after all previous write operations made by the same user are incorporated in the same replica.

  - Complete: All writes are included.

- **Consistent prefix** ensures that an **ordered sequence of writes is visible,** and is characterized by the following properties:

  - Ordered: Writes are accepted in the order that they were made.
  - Real: All writes are visible.
  - X latest missing: Some of the latest writes may be missing.
  - Snapshot isolation-like: The system behaves like snapshot isolation, where writes made by a transaction are not visible to other transactions until the transaction is committed.

- **Bounded staleness** ensures that all **writes older than X or every write except the last Y are visible**, and is characterized by the following properties:

  - Ordered: Writes are accepted in the order that they were made.
  - Real: All writes are visible.
  - X latest missing: Some of the latest writes may be missing.
  - Monotonic increasing due to bound: Write operations are accepted only after all previous write operations made by the same user are incorporated in the same replica, but the bound on staleness allows for some deviation from strict monotonicity.

- **Eventual consistency** ensures that a **subset of previous writes is visible**, and is characterized by the following properties:

  - Unordered: Writes may not be accepted in the order that they were made.
  - Un-real: Some writes may not be visible.
  - Incomplete: Some writes may be missing.

Each of them have their own trade-offs:

- **Strong consistency**:

  - Consistency: Excellent
  - Performance: Poor
  - Availability: Poor

- **Eventual consistency**:

  - Consistency: Poor
  - Performance: Excellent
  - Availability: Excellent

- **Consistent prefix**:

  - Consistency: Okay

  - Performance: Good
  - Availability: Excellent

- **Bounded stalenes**:
  - Consistency: Good
  - Performance: Okay
  - Availability: Poor

- **Monotonic reads**:
  - Consistency: Okay
  - Performance: Good
  - Availability: Good

- **Read my writes**:
  - Consistency: Okay
  - Performance: Okay
  - Availability: Okay

5 Distributed Services and Algorithms I

5.1 Types of Distributed Services

5.1.1 What is a Distributed Service?

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

- High scalability
- High availabilit

5.1.2 Services and Instances

- Distributed systems:
  - Comprised of **services**, such as applications, databases, caches, etc.
  - **Services are** made up of instances or nodes, which are **individually addressable hosts** (physical or virtual)

- Key observation:
  - Unit of **interaction is at the service level, not the instance level**

  - Concerned with **logical groups of nodes, not specific instances**
  - Example: Interacting with a database server, rather than a specific database instance.

5.1.3 Core Distributed Services

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

5.2 Availability

Is defined as:

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

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

5.2.1 Typical Hardware Causes for Downtime

- Overheating
- PDU failure
- Rack-move
- Network rewiring

- Rack failures
- Racks go wonky
- Network maintenances
- Router reloads
- Router failures
- Individual machine failures
- Hard drive failures

5.2.2 Availability through Redundancy

Across groups of resources:

- Multi-site data center
- Disaster recovery
- Scalability

Within a group of resources:

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

Between two resources:

- High availability (HA)
- Centralized administration (CA)
- Automatic failover (CO)
- Load distribution to prevent overload in case of failure

For an individual resource:

- Single point of failure (SPOF)
- Easy updates
- Maintenance problems
- Simple reliability
- Limited vertical scalability

5.2.3 3 Copy Disaster Recover Solution

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

5.3 Load Balancing

5.3.1 Serial vs. Redundant Availability

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

5.3.2 Global Server Load Balancing

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

  - Relies on the accuracy of the DNS provider's IP and location guesswork
  - TTL setting may not accurately reflect the time-to-live of cached information.

- **Real User Measurements (RUM)**:
  - Uses real-time data from end-user devices to dynamically adjust traffic distribution for optimal performance.

5.3.3 Failover with Virtual IPs

Failover with one virtual IP:

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

Multi-site failover:

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

5.3.4 Failover, Load Balancing and Session State

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

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

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

5.3.5 P2P Load Balancing

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

5.4 Caching

5.4.1 CQRS (Command Query Responsibility Segregation)

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

5.4.2 Requirements of Caching Services

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

5.4.3 Handling Changing Machine Counts in Caching Services

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

- Machines can be assigned multiple virtual positions.

5.4.4 Consistent Hashing Algorithms

Simple Consistent Hashing Algorithm:

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

Dynamo Consistent Hashing Algorithm:

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

5.4.5 Cache Patterns

Pull:

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

Push:

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

Pre-warmed:

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

General consideration: Be aware of LRU or clocked invalidations as cache is mission critical.

5.4.6 Cache Design Considerations

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

5.5 Events

5.5.1 Local vs. Distributed Events

Local:

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

Distributed:

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

5.5.2 Asynchronous Event-Processing

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

5.5.3 Features of Event-Driven Interaction

- **Basic Event**: Any entity can send and receive events without restrictions or filtering.
- **Subscription**: A receiver can subscribe to specific events, making event delivery more efficient.

- **Advertisement**: The sender informs receivers about possible events, reducing the need for broadcasting.
- **Content-Based Filtering**: The sender, middleware, or receiver can apply filtering based on event content.
- **Scoping**: Administrative components can manipulate event routes, enabling invisible communication between components.

5.5.4 Types of Message Oriented Middleware (MOMs)

Centralized Message-Oriented-Middleware:

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

Clustered Message-Oriented-Middleware:

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

Simple P2P Event Libraries:

- Local libraries are aware of each other, but **components are de-coupled**
- Broker-less architecture is **faster** than brokered ones
- **Does not provide** at-most-once semantics or **protection against message loss**
- Guarantees **atomicity** and **possibly FIFO**
- Examples include ZeroMQ, Aaron, and Nanomsg

Flooding Protocols:

- Notifications travel towards **subscriptions**, which are only kept at **leaf brokers**
- Advantages include that **subscriptions become effective quickly**, and notifications are guaranteed to arrive everywhere
- Price is **many unnecessary notifications** to leaf nodes without subscribers

5.5.5 ZeroMQ

- **Brokerless** socket library for messaging, with message filtering
- Connection patterns include **pipeline, pub/sub, and multi-worker**

- **Various transports**, including in process, across local process, across machines, and multicast groups
- **Message-passing process model** without the need for synchronization
- **Multi-platform and multi-language** support
- "Suicidal snail" fail-fast mechanism to **kill slow subscribers**

5.6 Sharding

5.6.1 Horizontal vs. Vertical Sharding

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

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

5.6.2 Sharding Strategies

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

5.6.3 Horizontal Sharding Functions

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

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

5.6.4 Consequences of Sharding

- **No more SQL `JOIN`s**, leading to lots of copied data
- Increased need for **partial requests** for data aggregation
- **Expensive distributed transactions** required for consistency (if needed)
- Vertical sharding distributes related data types from one user, while horizontal sharding distributes related users from each other (**bad for social graph processing**)

- **SQL limitations** due to mostly key/value queries and problems with automatic DB-Sequences
- Every change **requires corresponding application changes**

5.7 Relationships

5.7.1 Overview

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

For example when an employee leaves:

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

5.7.2 Functional Requirements of Relationship Services

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

5.7.3 Why Relationship Services Failed

The good:

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

The bad:

- Tends to create **many small server objects**
- **Performance impact**

- Not supported by many CORBA vendors for a long time
- EJB only supported with local objects in the same container.

6 Distributed Services and Algorithms II

6.1 Problems with Classic Concurrency

6.1.1 Why Truth is Expensive

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

6.1.2 Aspects of Classic Distributed Consistency

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

6.2 Locking

6.2.1 Locking Against Concurrent Access

Binary locks:

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

Modal locks (read/write locks):

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

Lock Granularity: The granularity of locks (the scope of the resources being protected by the lock) affects the overall throughput of a system. **The smaller the lock granularity, the better the performance** will be.

6.2.2 Optimistic Locking

Process:

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

Advantages:

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

6.2.3 Serializability with Two-Phase Locking

Process:

1. Allocate all locks

2. Manipulate the data
3. Release all locks

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

6.2.4 Deadlocks

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

6.2.5 Distributed Deadlock Detection

Local wait-for-graphs:

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

Detection servers:

- **Correctness**: Detection servers are designed to detect deadlocks in a distributed system, so this method should provide correct results if implemented correctly.
- **Liveness**: This method is designed to detect deadlocks in a distributed system, so it should have better liveness compared to local wait-for-graphs.
- **Cost/complexity**: The cost of implementing this method is higher than local wait-for-graphs, as it requires communication and coordination between multiple processes.
- **Failure model**: This method is susceptible to false negatives if one or more detection servers fail, or if there are errors in the communication between the servers.

- **Architecture type**: This method is suitable for systems with a decentralized architecture, where multiple processes are involved in the detection of deadlocks.

Distributed edge chasing algorithms:

- **Correctness**: This method is designed to detect deadlocks in a distributed system, so it should provide correct results if implemented correctly.
- **Liveness**: This method is designed to detect deadlocks in a distributed system, so it should have better liveness compared to local wait-for-graphs.
- **Cost/complexity**: The cost of implementing this method is higher than local wait-for-graphs, as it requires communication and coordination between multiple processes.
- **Failure model**: This method is susceptible to false negatives if there are errors in the communication between the processes.
- **Architecture type**: This method is suitable for systems with a decentralized architecture, where multiple processes are involved in the detection of deadlocks.

Stochastic detection:

- **Correctness**: The accuracy of this method depends on the parameters used, so it may provide incorrect results in some cases.
- **Liveness**: This method is designed to detect deadlocks in a distributed system, so it should have better liveness compared to local wait-for-graphs.
- **Cost/complexity**: The cost of implementing this method is relatively low, as it only requires monitoring resource requests and using randomization to make decisions.
- **Failure model**: This method may miss deadlocks if the randomization parameters are not set correctly.
- **Architecture type**: This method is suitable for systems with a decentralized architecture, where multiple processes are involved in the detection of deadlocks.

6.3 Transactions

6.3.1 Classic ACID Definitions

- **Durability**: Ensures that once a transaction is committed, its effects **persist even in the case of system failures** (e.g. a crash that causes you to lose changes made to a word file)
- **Atomicity**: Ensures that a transaction is treated as a single, indivisible unit of **work that either happens in its entirety or doesn't happen at all** (e.g. in the case of a birthday party reschedule where not all participants were caught in time)

- **Isolation**: Ensures that the concurrent execution of transactions results in a system state that would be obtained **as if transactions were executed serially** (e.g. if two people work on a shared file, their changes should not interfere with each other)
- **Consistency**: Ensures that the **system remains in a valid state after a transaction is executed** (e.g. after you complete a friend's work for the day, the tasks remain consistent, and the system remains in a valid state)

6.3.2 Transaction Properties and Mechanisms

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

6.3.3 Serializability and Isolation

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

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

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

6.3.4 Transaction API

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

Commit transaction:

- System has a new, consistent state
- Local objects are now invalid

- Changes are visible to others

On error: Rollback:

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

6.3.5 Components of Distributed Transactions

Process

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

RPCs:

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

Components:

- Transaction
- Transactional client
- Transactional servers (objects)
- TACoordinator
- XA resource managers

6.3.6 Service Context

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

6.3.7 Distributed Two-Phase Commit

Vote:

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

Completion:

- The **coordinator is the only entity that can commit or abort** a TA after the prepare phase.
- If the vote phase was successful and all participants have prepared for a commit, the **coordinator calls for a commit**.
- In case of an error (e.g. unreachable participant), the **coordinator calls for a rollback**.

6.3.8 Failure Models of Distributed Transactions

Work Phase:

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

Voting Phase:

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

Commit Phase (Server Uncertainty):

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

6.3.9 Special Problems of Distributed Transactions

Resources:

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

Coordinator as a Single Point of Failure:

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

Heuristic Outcomes for Transactions:

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

6.3.10 Transaction Types

Flat Transactions:

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

Nested Transactions:

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

Long-running Transactions:

- Challenge is resource allocation and increasing amount of work lost in case of rollback.
- Syncpoints move the fallback position closer to the commit point.

Compensating Transactions:

- Improves transaction throughput by making objects visible sooner, at the cost of sacrificing the ISOLATION property.
- Require compensation for previous TA which can no longer be rolled back.
- Depend on the application whether compensating transactions are possible.
- Can be hand-coded if no transaction monitor/manager is available.

6.3.11 ANSI Transaction Levels

Problems:

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

Transaction Levels:

- **Read Uncommitted**:
  - Prevents: Nothing
- **Read Committed**:
  - Prevents:
    - Dirty reads.
- **Repeatable Read**:
  - Prevents:
    - Dirty reads
    - Non-repeatable reads
- **Serializable**:
  - Prevents:
    - Dirty reads
    - Non-repeatable reads
    - Phantom reads

The higher the level, the more overhead is required.

6.4 NoSQL

6.4.1 Forces behind NoSQL

- Need for low-latency and high-throughput access to data
- Difficulty in managing and maintaining consistency in a distributed system
- Increased focus on scalability and flexibility
- Changing data requirements and needs for real-time processing
- Cost and complexity of traditional RDBMs in large-scale systems

- Inability of RDBMs to handle large amounts of unstructured data
- The need for horizontal scaling in storage
- Lack of support for real-time, complex data processing using RDBMs
- The need for automatic scaling in storage to keep up with rapidly growing data
- Relaxed data consistency requirements in some applications.

6.4.2 Scaling Problems of RDBMs

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

6.4.3 NoSQL Design Patterns

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

6.4.4 DynamoDB Design Principles

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

- Multi-version store with conflict resolution policies for better data management

6.5 Beyond Relaxed Consistency

6.5.1 Overview

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

6.5.2 CALM Principle

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

6.5.3 CALM Operations

Logically Monotonic:

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

Non-monotonic:

- Overwriting variables
- Set deletion
- Resetting counter
- Negation

6.5.4 CRDTs

State-based CRDTs:

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

Operation-based CRDTs:

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

6.5.5 Bending the Problem

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

6.5.6 Examples of CRDTs

Counters:

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

Registers:

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

Sets:

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

- Two-phase set: Consists of two sets, one for adding and another for removing, elements can be added once and removed once
- Unique set: Optimized version of the two-phase set
- Last write wins set: Merge operation is max(ts), payload is a set
- Positive-negative set: Consists of one PN-counter per set item
- Observed-remove set

6.5.7 Distributed Coordination

Features:

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

Liveness and Correctness:

- **Sequential Consistency**: Updates from a client will be applied in the order they were sent.
- **Atomicity**: Updates either succeed or fail. No partial results.
- **Single System Image**: A client will see the same view of the service regardless of the server it connects to.
- **Reliability**: Once an update has been applied, it will persist from that time forward until a client overwrites the update.
- **Timeliness**: The client's view of the system is guaranteed to be up-to-date within a certain time bound.

6.6 Distributed Coordination Services

6.6.1 Zookeeper API

- **create**: Creates a node at a specified location in the tree

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

6.6.2 Primary-Order Atomic Broadcast with Zab

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

6.6.3 Consistency Requirements for ABCast (Reliable Ordered Atomic Broadcast)

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

6.6.4 Primary Order

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

6.6.5 HA Transactions

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

7 Design of Distributed Systems

7.1 Design Principles for Distributed Systems

7.1.1 Overview

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

> Know Your No. 1 Enemy: **Latency**!

7.1.2 Sharing Ressources and Data

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

7.1.3 Connection Pooling

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

7.1.4 Horizonal Scaling/Parallelization

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

7.1.5 Caching and Replication

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

7.1.6 End-to-End Argument

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

7.1.7 Design Methodology

- **Back-of-the-envelope** calculations
- Decide on **geographical distribution and replication strategy**
- Determine **data segregation**, including single or multi-tenancy models and partitioning
- Divide **business requirements into REST**-like services

- **Define SLAs for services**, including availability, latency, throughput, consistency, and durability
- **Define security context** with IAAA (Identity, Authentication, Authorization, Audit) and perform risk analysis
- Complete **monitoring and logging setup**
- **Plan for deployment, release changes, testing, and maintenance** using fault-tolerant features.

7.1.8 Uncomfortable Real-World Questions

- How many application servers are needed to support the customer base?
- What is the optimal ratio of users to web servers?
- What is the maximum number of users per server?
- What is the maximum number of transactions per server?
- Which specific hardware configurations provide the best performance?
- What is the current production server capability?
- What do the users do? (These are business process definitions.)
- How fast do the users do it? What are the transaction rates of each business process?
- When do they do it? What time of day are most users using it?
- What major geographic locations are they doing it from?
- How many connections can the server handle?
- How many open file descriptors or handles is the server configured to handle?
- How many processes or threads is the server configured to handle?
- Does it release and renew threads and connections correctly?
- How large is the server's listen queue?
- What is the server's "page push" capacity?
- What type of caching is done?

7.1.9 Best Practices for Designing Services

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

- Manage **dependencies carefully**
- **Create APIs** for customer access to services

7.2 Architecture Fields

7.2.1 Overview

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

7.2.2 Information Architecture (to analyze Caching)

> Defines pieces of information to aggregate or integrate

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

7.2.3 Distribution Architcture

> Tells portal how to map/locate fragments defined in the information

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

Additional factors to consider:

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

Example results:

- Back-end server performance affecting home page construction time
- Huge latencies and variations in response times causing instabilities in the portal application
- Getting to the servers for every request is nearly impossible due to huge latencies and variations in response times from dependencies.

7.2.4 Service Access Layer

Is determined by the distribution architecture.

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

Simple Alternative: Sidecar, contains circuit breaker & service discovery

Advanced Alternative: Service mesh with separate data and control plane

7.2.5 Physical and Process Architecture

Physical Architecture:

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

Horizontally scalable application:

- Replicated on multiple hosts
- Avoids single point of failure

Vertically scalable application:

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

7.2.6 Architecture Validation

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

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

7.3 Fan-Out Architecture

7.3.1 Overview

Calls are parallel instead of serial.

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

7.3.2 Reliability Issues in Dependencies

- System load becomes worse due to **hanging requests** occupying resources and leading to heavy garbage collection
- Dead servers can cause a **buildup of threads** due to even short timeouts
- The portal was **frequently impacted by failing back-end servers**

- **Avoid lengthy waiting time** for sub-requests in the homepage action handler: Adopt the "Fail-fast" pattern today.

7.3.3 Fragments

Pages: Unique to customers, cannot be re-used

Page fragments:

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

7.3.4 Latency Reduction & Tolerance

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

7.3.5 Avoiding Getting Stuck

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

7.4 Containing Failures

7.4.1 Circuit Breakers

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

7.4.2 Bulkheads

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

7.4.3 Blast Reduction

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