Scalable Distributed Computing

1. Basic Concepts

Distributed - Sharing tasks and resources across many computers or places.

Distribution makes systems stronger and faster by not depending on just one place. **Example:** Google's worldwide data centers let users everywhere get quick search results.

Scalable - The ability to grow and handle more work without losing speed or quality. Scaling means expanding while still working efficiently.

Scalability means a system can handle more work by adding resources (like servers, nodes, or storage). There are two main types of scalability:

- **Vertical** – adding more power (CPU, memory, etc.) to one machine.
- **Horizontal** – adding more machines (nodes) to the system.

When talking about scalability, we look at two things: **Performance –** how well the system handles more requests. **Elasticity** – how easily the system can grow or shrink based on current demand.

Scalability can be of two kinds:

- **Strong Scalability** – performance grows almost equally with the number of resources. Best when work can be shared evenly across many machines.
- **Weak Scalability** – performance stays the same even as users grow. Best when workload grows together with resources.

Motivation for Scalable Distributed Systems

- **Growing data –** huge datasets from social media, IoT, AI, etc.
- **Global users –** billions of people around the world.

- **Performance –** need to lower latency (response time), increase throughput (work done per second), and improve reliability.

⚠️ Note: Sometimes **latency** (delay) is not the main goal — some systems prefer higher throughput and reliability even if latency is higher.

The main challenges are managing resources across worldwide systems and ensuring both **low latency** and **high availability**.

Target Architectures: Examples of architectures that need scalable distributed systems:

- IoT networks
- High-Performance Computing (HPC)
- Cloud / Edge Computing

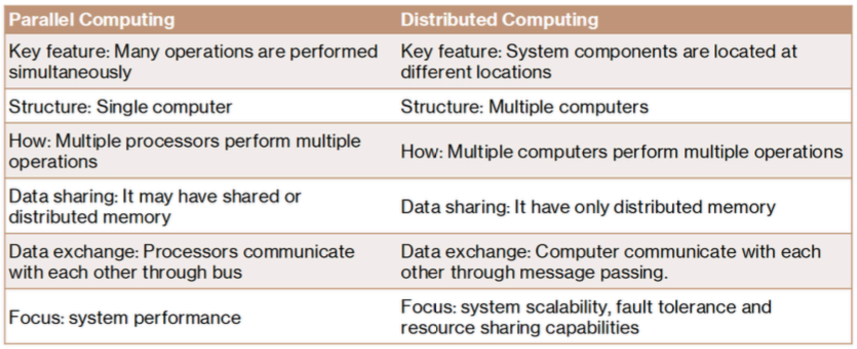
Even simpler applications can benefit, such as: Large graph analysis, Stream processing, Streaming services (Netflix, YouTube, etc.), Machine Learning, Big Data, Computational Fluid Dynamics, Web and online services

Parallel Computing

Question: Can parallel computing solve these problems?

Answer: Not really — parallel computing is useful, but for different goals and works differently than distributed systems.

Distribution is powerful, but remember: **Local computation is always much faster than remote computation.** From the CPU's view, waiting for data from another machine feels extremely slow.



Parallel Computing	Distributed Computing
Key feature: Many operations are performed simultaneously	Key feature: System components are located at different locations
Structure: Single computer	Structure: Multiple computers
How: Multiple processors perform multiple operations	How: Multiple computers perform multiple operations
Data sharing: It may have shared or distributed memory	Data sharing: It have only distributed memory
Data exchange: Processors communicate with each other through bus	Data exchange: Computer communicate with each other through message passing.
Focus: system performance	Focus: system scalability, fault tolerance and resource sharing capabilities

2. Elements and Challenges in Distributed Systems

Definition: "A distributed system is one in which the failure of a computer you didn't know existed can render your own computer unusable." - Leslie Lamport.

A simpler and widely used one is from Andrew S. Tanenbaum:

"A distributed system is a collection of independent computers that looks to users like a single system." **T**wo key points:

1. **Autonomy** – Each computer (node) works independently and can make its own decisions.
2. **Transparency** – The whole system looks like one single unit to the user.

Autonomous Computing Elements

- Each node is independent but must cooperate with others to reach common goals.
- Nodes communicate by **sending and receiving messages**, which may lead to further communication.
- Since multiple nodes exist, each must know:
  - which other nodes are in the system,
  - how to reach them.

This requires a **robust naming system** (like DNS), so that nodes can be identified and scalability is possible.

Key idea: **A node's name should not depend on its physical location.** Also, because each node has its own sense of time, there is usually **no single global clock**. Systems need synchronization and coordination to work correctly.

Transparency

A distributed system should **look like one single system** to users. Users should not notice that data, processes, and control are spread across many machines.

This goal is called ***distribution transparency***.

📌 Example: In Unix-like operating systems, many resources are accessed through a single file-system interface. This hides differences between files, devices, memory, and even networks.

⚠️ Challenge: While transparency is a goal, **partial failures** (where only some nodes fail) are very common in distributed systems and much harder to hide compared to simple centralized systems.

Middleware

To make developing distributed applications easier, distributed systems often include a special software layer on top of the operating systems of the computers. This layer is called **middleware** [Bernstein, 1996].

👉 Middleware in a distributed system is like an **operating system for a single computer**:

- it manages resources,
- shares them efficiently,
- and allows applications to use them across the network.

Some examples of middleware services are:

2.2.1.1 RPC – Communication

- **Remote Procedure Call (RPC):** lets a program call a function on another computer **as if it was local**.
- Developers just write the function header; the RPC system handles the rest (sending the call, getting the result).
- Examples: **Sun RPC, Java RMI, gRPC (Google RPC)**.

2.2.1.2 Service Composition

- Lets developers build new applications by combining existing ones.
- Example: **Web services** that combine data from multiple sources.

- A classic case: **Google Maps** enhanced with info from restaurants, traffic, or reviews.

2.2.1.3 Improving Reliability

- Example: **Horus toolkit** allows building apps as groups of processes where messages are either received by **all processes or none**.
- This guarantee makes distributed systems more reliable and easier to program.
- Such reliability features are usually provided by middleware.

2.2.1.4 Supporting Transactions on Services

- Many apps use multiple services spread across different computers.
- Middleware supports **atomic transactions** (all-or-nothing execution).
- This avoids issues like **race conditions, deadlocks, and partial updates**.
- Developers just specify the services; middleware ensures either all succeed or none do.

Creating **Distributed Systems** is difficult, and not always the best option. To make the effort worthwhile, four main goals should be achieved:

1. Resource accessibility
2. Hide distribution
3. Be open
4. Be scalable

2.3.1 Resource Accessibility

In a distributed system, being able to **access and share remote resources** is extremely important. Resources can be anything: storage, devices, data, or applications.

- Connecting users and resources makes collaboration and information sharing easier (the Internet is the best example).

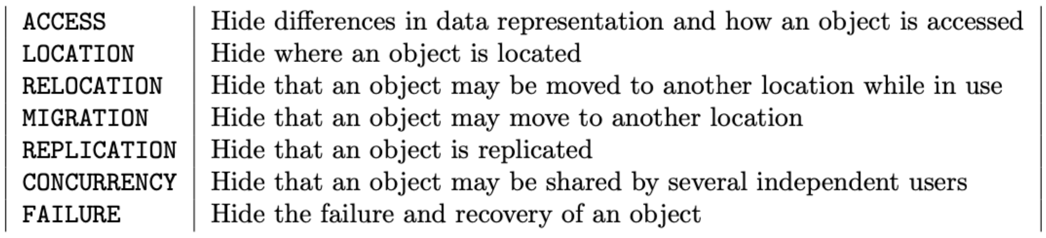
- File-sharing systems are also a good case: they distribute large files, software updates, and synchronize data across many servers.

2.3.2 Hide Distribution

Hiding distribution is a key design goal. It is related to **distribution transparency** (explained earlier), but goes further.

It means the system should **hide the fact** that processes and resources are physically spread across many computers, possibly far apart.

More precisely, it should provide properties that make the system look and act like a **single unified system** — even though it is distributed.



ACCESS	Hide differences in data representation and how an object is accessed
LOCATION	Hide where an object is located
RELOCATION	Hide that an object may be moved to another location while in use
MIGRATION	Hide that an object may move to another location
REPLICATION	Hide that an object is replicated
CONCURRENCY	Hide that an object may be shared by several independent users
FAILURE	Hide the failure and recovery of an object

Table 2.1: Hide Distribution properties

2.3.2.1 Access Transparency

Different systems use different ways to represent and access data. A good distributed system should **hide these differences**, such as:

- different machine architectures,
- different operating systems,
- different naming conventions or file operations,
- different ways of communication between processes.

👉 Example: accessing files on different OSs (Windows, Linux, Mac) without caring about how each internally names or stores them.

2.3.2.2 Location and Relocation Transparency

- **Location transparency:** Users can access resources without knowing where they are physically stored.
    - Achieved with **logical names** (indirection).
    - Example: A URL like `http://www.unipi.it` doesn't reveal the actual server location.
- **Relocation transparency:** Even if a resource moves to another machine or data center, users should not notice.
    - Especially important in **cloud computing**.

2.3.2.3 Migration Transparency

- Goes beyond relocation.
- Means resources or processes can **move across the system when initiated by users** without breaking communication.
- Example: Mobile phones – even if people move around, calls or video conferences continue smoothly.

2.3.2.4 Replication Transparency

- Resources may be **replicated** to improve performance or reliability.
- The system should **hide the fact that there are multiple copies**.
- All replicas must share the same name so the user doesn't notice.
- Replication transparency usually depends on **location transparency**.

2.3.2.5 Concurrency Transparency

- Multiple users may access the same resource at the same time (e.g., same file, same database). The system ensures:
    - **Consistency** of the resource,
    - through **locking** or **transactions**.

- Users should not notice other users working on the same resource.

2.3.2.6 Failure Transparency

- The system should **hide failures** of components from users.
- If something fails, the system should **automatically recover** without the user noticing.
- Very difficult to achieve in practice.
- Main issue: it's often impossible to tell whether a process has **crashed** or is just **slow**.

2.3.3 Degree of Distribution Transparency

In theory, **more transparency is good** for distributed systems, but in practice there is a **trade-off**:

- **Too much transparency** can hurt **performance** or make the system harder to design.
- Sometimes it is better **not to fully hide distribution**.

⚠️ Cases where hiding too much is a problem

- **Internet applications:** If a server is down, the system may keep trying to hide it, wasting time before switching to another server. This slows things down.
- **Replicas worldwide:** Keeping replicas in sync across continents may take seconds for each update. Hiding this delay from users is not realistic.

📍 Cases where hiding distribution is not always good

- **Location awareness:** For mobile devices (phones, wearables), knowing *where* the device is matters (e.g., finding the nearest restaurant). Hiding this would be harmful.
- **Real-time collaboration:** In shared document editing, too much concurrency transparency can block smooth cooperation (users need to see who is editing what).

🔑 Key point

Full transparency is impossible.

Sometimes it is better to **make distribution explicit** so that:

- Users and developers understand the system's real behavior.
- They are better prepared to deal with failures, delays, or limits.

2.4 Openness

An **open distributed system** is one where components can:

- be **easily used or integrated** into other systems,
- and often consist of components created elsewhere.

Being open enables two main features:

1. **Interoperability, composability, and extensibility**
2. **Separation of policies from mechanisms**

How openness works

- Components follow **standard rules** for how they look (syntax) and how they work (semantics).
- These rules are usually described with an **Interface Definition Language (IDL)**.
- With an IDL, one process can use a service provided by another, even if they were developed independently.

Example: Two companies can build different systems, but if both follow the same IDL, they can still work together.

Specifications

Complete: everything needed to implement the interface is described.

Neutral: they don't force a specific implementation style.

⚠️ In practice, many interface definitions are incomplete, meaning developers must add their own details.

Key benefits of openness

Interoperability – systems from different vendors can:

- work together,
- use each other's services,
- as long as they follow the same standard.

Portability – an application made for one distributed system can run on another **without changes**, if both implement the same interfaces.

Extensibility – it should be easy to:

- configure a system from components (even from different developers),
- add new components,
- replace existing ones without breaking the rest.

2.4.1 Policy and Mechanism Separation

To make **open distributed systems flexible**, they should be built as a set of **small, replaceable, and adaptable components**.

- This means defining not only the **top-level interfaces** (used by applications and users), but also the **internal interfaces** (between system components).
- By doing this, components can interact in a **modular way**.

This design is the opposite of the **monolithic approach**, where the whole system is one big program.

- In a monolithic system, replacing or changing one part is very hard because it affects everything else.
- In a modular, open system, you can replace or adapt one component without breaking the rest.

2.5 Scalability

Originally, applications were run on powerful desktops. Now, many services run **in the cloud**, and users connect through smaller devices (phones, tablets).

👉 Because of this, **scalability** is now one of the most important goals in distributed systems.

2.5.1 Dimensions of Scalability

Scalability can be looked at from three perspectives:

- **Size scalability** – The system can handle more users and resources **without performance loss**.
- **Geographical scalability** – The system works well even when users and resources are **far apart**, without noticeable delays.
- **Administrative scalability** – The system can still be managed even if it spans **many organizations** with different policies.

2.5.2 Size Scalability

Centralized services (one server or one cluster) often become a **bottleneck** due to:

- **CPU limits** → computation bound,
- **Storage/I/O limits** → I/O bound,
- **Network limits** → network bound.

2.5.3 Geographical Scalability

- Systems designed for **local-area networks (LANs)** don't always work well on **wide-area networks (WANs)**.
- In LANs, requests take microseconds, but in WANs, they can take hundreds of milliseconds.
- WANs also have higher packet loss and lower bandwidth.

  👉 This makes **synchronous communication (blocking)** problematic at large scale.

2.5.4 Administrative Scalability

When a system spans **different organizations**, problems arise with:

- conflicting **resource usage rules**,
- **costs**,

- and **security policies**.

2.6 How to Scale?

Scalability issues often appear as **performance problems**.

- **Scaling up** = making one server stronger (faster CPU, more memory, better network).
- **Scaling out** = adding more servers.

There are **3 main strategies to scale out**:

2.6.1 Hiding Communication Latencies

- Instead of waiting for replies from remote servers, do **other useful work** while waiting. (→ asynchronous communication).
- Example: batch processing, parallel apps.
- If async doesn't fit, move part of the computation closer to the client → **edge computing** (hierarchical approach).

2.6.2 Distributing Work

- Break components into smaller parts and spread them across the system.
- Example: **The Web –** looks like one giant system, but data is actually stored on millions of servers worldwide.

2.6.3 Replication and Caching

Replication

- Make multiple copies of components/resources.
- Benefits: improves availability, balances load, reduces latency (copies closer to users).
- Challenge: keeping replicas **consistent** when updates happen.

Caching

- A special form of replication decided by the **client** (not the resource owner).

- Example: browsers cache web pages.
- Problem: **inconsistency** when cached copies are outdated.

2.6.4 Pitfalls

Designing scalable distributed systems is **hard** — mistakes often happen because new developers assume wrong things. In reality, networks fail, change, cost money, and are heterogeneous.

2.7 Types of Distributed Systems

2.7.1 HPC (High-Performance Computing) – Clusters

- Collection of similar workstations connected via **high-speed networks**.
- Homogeneous (same OS, similar hardware).
- Used in supercomputers → scientific & large-scale problems.
- **Advantages over one big machine**: cheaper, reliable, easy to scale horizontally.
- **Beowulf Clusters**: Linux-based, cheap, popular in science.

2.7.2 HPC – Grid Computing

- Also for HPC, but:
    - Nodes belong to **different organizations**.
    - Geographically distributed.
    - Heterogeneous (different OS, machines).
- Often a **federation** of computer systems.
- **Advantage**: heterogeneity can help (different workloads → different machines).

2.7.3 Cloud Computing

- Not HPC, but **on-demand resources** from a provider.
- Users **outsource infrastructure** and compose resources dynamically.

- Origin: **utility computing** → pay-per-use model.

Service models:

- **IaaS** – Infrastructure as a Service
- **PaaS** – Platform as a Service
- **SaaS** – Software as a Service
- **FaaS** – Function as a Service
- And more... (DBaaS, BaaS, etc.)

Issues:

- **Vendor lock-in** → hard to switch providers
- **Cost** → uploading data is cheap, downloading (pulling) is expensive
- **Security/privacy** → provider holds your data
- **Network dependence** → no network = no service

2.8 Pervasive Systems

Unlike "stable" distributed systems (fixed nodes, strong connections), pervasive systems are **mobile, dynamic, context-aware**.

- Devices: small, battery-powered, wireless (e.g., smartphones, wearables, IoT).
- Interaction: no single interface → sensors, context-awareness.

Types of pervasive systems (can overlap):

1. **Ubiquitous computing systems** → computers embedded everywhere in daily life.
2. **Mobile systems** → smartphones, tablets, laptops.
3. **Sensor networks** → many small devices collecting environment/user data.

Key Performance Laws

Amdahl's Law: Limits of Parallelism

Amdahl's Law explains that the parts of a task that must run in sequence limit the total speed improvement, even if many processors are used.

Parallel Computing Speedup: Amdahl's Law explains how tasks can run faster with parallel computing, but only up to a certain limit.

Diminishing Returns: Adding more processors gives smaller improvements in speed because some parts of the task cannot run in parallel.

$$S(p) = \frac{T(1)}{T(p)} = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

Gustafson's Law: Scalability

Gustafson's Law says that as problems get larger, they can achieve more speedup with parallel processing, showing how scalability works.

Scalability in Parallel Computing: Gustafson's Law explains how parallel computing systems can scale better as more processors are added.

Workload Expansion Principle: Unlike Amdahl's Law, Gustafson's Law says that the total work can grow when more processors are available, making the system more efficient.

Speedup Formula Explained: The formula **$S = P + (1 - P)N$** shows how the system's speed increases based on the number of processors and the amount of work that can run in parallel.

Little's Law: Queuing Insight

Little's Law connects the average number of items in a system, how often new items arrive, and how long each item stays in the system.

Key Principle of Little's Law: Little's Law connects the average number of items in a system with how often they arrive and how long they stay. It is a key idea in queueing theory.

Applications in Business: The law helps businesses improve processes and reduce waiting times, leading to better efficiency and customer experience.

Foundation in Operations Management: Little's Law is an important concept in operations management and service design, helping make service systems more effective.

3. Time

These aspects were already mentioned, but let's recall some time-related issues in distributed systems:

- There is no global clock, every machine has its own.
- Communication is asynchronous and "best effort" → messages may be delayed or lost.
- There is no central authority, but coordination between nodes is still needed.

Time is important in distributed systems for several reasons:

1. Ordering → to know in which order events happened.
2. Causal relationships → to understand cause-and-effect between events.
3. Handling inconsistencies between nodes.
4. Handling conflicts, such as multiple updates to a shared resource.

Some systems are very time-dependent:

- Distributed databases → data must stay consistent, and transactions must either fully succeed or fully fail.
- IoT systems → commands (e.g., "turn left") are useless if they arrive too late.
- Cloud systems (auto-scaling, load balancing) → decisions based on outdated measurements (e.g., 2 hours old) waste resources and money.

3.1 Challenges

One key issue is **clock drift**: each machine has its own physical clock, and due to hardware imperfections they slowly become unsynchronized. This can cause different timestamps for the same event.

Another issue is **network latency**: messages between nodes may be delayed, leading to events being observed in the wrong order.

3.1.1 NTP and PTP – Protocols to Solve Drift

NTP (Network Time Protocol) synchronizes computer clocks over a network.

- It works in "stratum levels": stratum 0 = reference clock, stratum 1 = server using stratum 0, and so on up to stratum 15.
- NTP can keep clocks synchronized within milliseconds, which is enough for many systems, but not for high-precision needs.

PTP (Precision Time Protocol) provides higher accuracy, down to sub-microseconds.

- Used in areas like high-frequency trading, telecom, and industrial automation.
- Uses a master-slave model with a "grandmaster clock" giving time to all others.
- Often relies on special hardware (like NICs with timestamping) to achieve this precision.
- Without good hardware support, even PTP cannot guarantee perfect accuracy.

3.1.2 Logical and Physical Clocks

Logical clocks do not show real-world time but are used to order events correctly when physical clocks are not synchronized.

Examples: Lamport timestamps and vector clocks.

3.1.2.1 Lamport Timestamps

1. Each process keeps a counter that increases with every event.
2. When sending a message, the process adds its counter value to the message.
3. The receiving process updates its counter to be larger than both its own and the received counter.

Example: If Client A writes to a file and then Client B reads it, Lamport timestamps ensure B sees the updated content in the right order.

3.1.2.2 Vector Clocks

1. Each process keeps a vector (list) of counters, one for each process.
2. When sending a message, the process increases its own counter and sends the whole vector.
3. The receiving process updates its vector by taking the maximum of each position between its own and the received vector.

Example: In collaborative editing (like Google Docs), vector clocks help detect and merge edits. By comparing vectors, the system knows the order of edits and keeps documents consistent across users.

4. Synchronizing

Synchronization means coordinating the actions of multiple processes that share resources. In short, it makes sure that processes using the same resources do not interfere with each other.

- Need to avoid inconsistencies or conflicts → Prevent errors when multiple processes try to use or change the same data at the same time.
- Managing access to shared resources → Do it in a way that keeps data correct and the system efficient.

In distributed systems, processes communicate over networks by sending messages. This makes synchronization harder because:

1. Network delays → messages can take different times to arrive, causing inconsistencies.
2. Process failures → a process may crash anytime, and the system must handle it.
3. No fairness → we must avoid situations where some processes wait forever (starvation).

4.1 Mutual Exclusion

Mutual exclusion is a common way to control access to shared resources in distributed systems.

Definition: Mutual exclusion ensures that only one process can enter a **critical section** at a time.

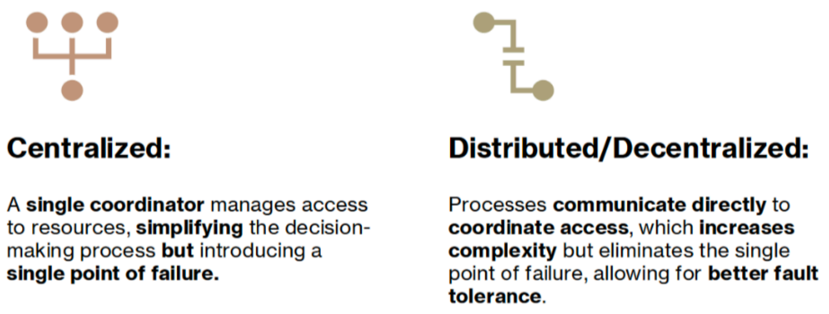
- A critical section is the part of code where shared resources are used.
- This guarantees atomic access (all-or-nothing, one at a time) to resources, avoiding conflicts and inconsistencies.

Goals of Mutual Exclusion

- **Safety** → only one process at a time in the critical section.
- **Liveness** → every process that wants to enter will eventually succeed (no starvation).
- **Fairness** → processes enter the critical section in the order they requested it.

Types of Mutual Exclusion

1. **Software-based** → Algorithms like Lamport's Bakery, Peterson's, Dekker's.
2. **Hardware-based** → Using atomic operations or locks supported by hardware.
3. **Hybrid approaches** → Combine software and hardware methods for better performance and reliability.



Centralized:

A **single coordinator** manages access to resources, **simplifying** the decision-making process **but** introducing a **single point of failure.**

Distributed/Decentralized:

Processes **communicate directly** to **coordinate access**, which **increases complexity** but eliminates the single point of failure, allowing for **better fault tolerance**.

Figure 4.1: Mutual exclusion - Centralized vs Distributed

4.1.2 Lamport's Bakery Algorithm (LBA)

LBA is a software-based mutual exclusion algorithm that uses a **ticket system** to guarantee fairness and prevent starvation. It works like a bakery: processes are like customers, and they take numbers to be served in order.

Steps:

1. A process takes a ticket number that is one more than the highest ticket currently in use.
2. The process waits until all processes with smaller tickets finish their critical sections.
3. The process with the smallest ticket number enters the critical section.
4. After finishing, the process resets its ticket to show it is done.

Sometimes, two processes may get the same ticket number if they request it at the same time. In this case, the one with the smaller process ID (pid) goes first. This ensures fairness.

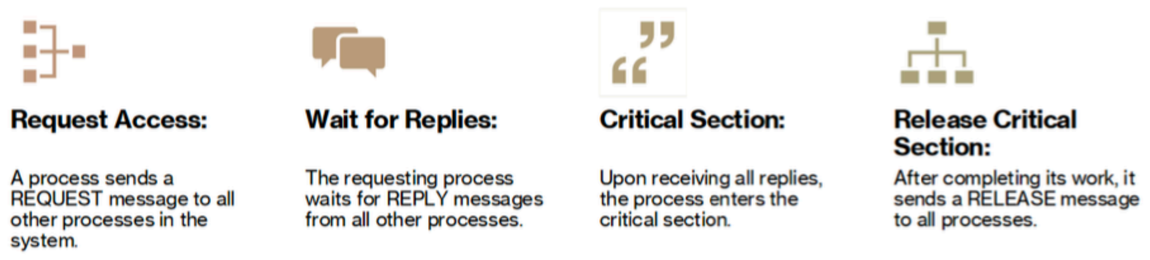
LBA does not rely on special hardware instructions (like spinlocks or test-and-set). It only uses simple ticket logic, which makes it portable and easy to implement in different systems.

Drawback: In large systems, LBA can cause performance problems because processes must constantly check (poll) other tickets. It works best in environments with shared memory and a small number of processes.

4.1.3 Lamport's Distributed Mutual Exclusion Algorithm (LDMEA)

LDMEA is a distributed version of the Bakery Algorithm. It allows processes on different nodes to access shared resources fairly by using **logical clocks**.

- Each process has a logical clock that increases with every request or message received.
- Each request gets a timestamp from this clock.
- Requests are ordered by timestamp, and the one with the smallest timestamp gets access to the critical section.



Request Access:

A process sends a REQUEST message to all other processes in the system.

Wait for Replies:

The requesting process waits for REPLY messages from all other processes.

Critical Section:

Upon receiving all replies, the process enters the critical section.

Release Critical Section:

After completing its work, it sends a RELEASE message to all processes.

Figure 4.2: LDMEA Key Steps

4.1.3.1 Entering the Critical Section

- When a site **Si** wants to enter, it sends a request message `Request(tsi, i)` (with its timestamp) to all other sites.
- It also adds this request to its local request queue.
- Site **Si** can enter the critical section only if:
    1. It has received replies from all other sites with timestamps larger than its own request.
    2. Its request is at the top of its local queue.

4.1.3.2 Leaving the Critical Section

- When **Si** leaves, it removes its request from the top of its queue.
- It sends a `Release(tsi, i)` message to all other sites.
- Other sites then remove **Si**'s request from their queues.

4.1.3.3 Drawbacks

Example flow:

1. P1 sends `REQUEST` to P2 and P3.
2. P2 and P3 reply.
3. Once P1 gets all replies, it enters CS.
4. After finishing, P1 sends `RELEASE` to P2 and P3.

Message cost:

- (N − 1) request messages
- (N − 1) reply messages
- (N − 1) release messages

  = **3N messages per CS entry**.

Problems:

- High number of messages.
- Needs extra mechanisms to handle process failures and network issues.

But it is **better than LBA** in distributed systems because it avoids constant polling.

4.1.4 Other Approaches

Mutual exclusion algorithms are grouped into:

1. **Token-based** algorithms
2. **Non-token-based** algorithms
3. **Quorum-based** algorithms

4.2 Token-Based Algorithms

- A unique token circulates among processes.
- Only the process with the token can enter CS.

Pros:

- Efficient when managed well.
- Low communication overhead (if no contention).

Cons:

- Token loss can disrupt the system.
- Delays in passing token increase waiting time.

4.2.1 Suzuki-Kasami Algorithm

- Works with a token.
- Requests are sent to all processes when needed.

- If a process already has the token, it enters immediately.

Data structures:

- Each process has an array **RN_i[N]** (Request Numbers): tracks the last request number received from each process.
- Token stores:
    1. **LN[N]** (Last Request Numbers granted).
    2. **Queue Q** of processes waiting for the token.

Performance:

- Needs **0 or N messages** per CS entry.
- Delay is **0 or N** (depending on whether the process has the token).

Issues:

- Must detect outdated requests.
- Must decide fairly who gets the token next.
- If token is lost, processes can hang → recovery needed.

Still, it is **efficient for high-latency networks** since message cost is low compared to non-token algorithms.

4.3 Non-token-based Algorithms

- No token used.
- Processes request permission from others directly.
- Uses more messages than token-based systems.

4.3.1 Ricart-Agrawala Algorithm

- Each process sends a **REQUEST** to all others when it wants CS.
- It waits until all replies are received.
- Priority is based on **timestamps** (earlier timestamp = higher priority; ties broken by process ID).

Message cost:

- **2(N – 1)** messages per CS entry (request + reply).

Pros:

- Fair ordering using timestamps.

Cons:

- High message cost.
- Not good for slow or unreliable networks.

4.4 Quorum-based Algorithms

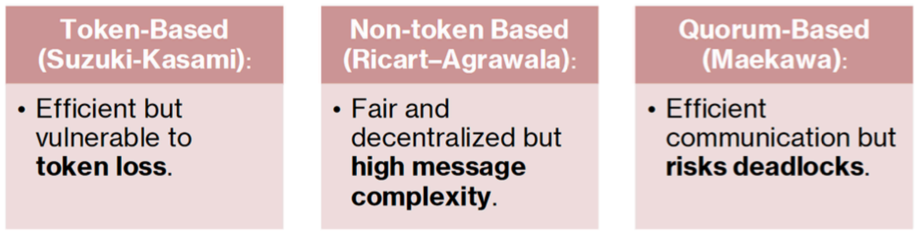
- Instead of contacting **all processes**, a process asks only a **subset (quorum)** for permission to enter the critical section.
- Rule: any two quorums must overlap → ensures at least one common process, so mutual exclusion is guaranteed.
- Advantage: uses fewer messages than Ricart-Agrawala, so it is good for systems with **many processes** and **high contention**.

4.4.1 Maekawa's Algorithm

- Each process belongs to one or more **quorums**.
- To enter CS, a process requests permission only from its quorum.
- Because quorums overlap, no two processes can both get access at the same time.
- **Pros:** reduces message complexity.
- **Cons:** quorum design must be careful; otherwise, deadlocks can happen.

4.5 Wrap Up

Summary of Key Features



Token-Based (Suzuki-Kasami):	Non-token Based (Ricart–Agrawala):	Quorum-Based (Maekawa):
• Efficient but vulnerable to **token loss**.	• Fair and decentralized but **high message complexity**.	• Efficient communication but **risks deadlocks**.

Message Complexity

- **Suzuki-Kasami:** Low message cost (just token passing).
- **Ricart-Agrawala:** High message cost (must contact all processes).
- **Maekawa:** Fewer messages than Ricart-Agrawala (asks only quorum).

Scalability

- **Token-based:** Scales well for smaller systems, but token management is harder with many processes.
- **Non-token (Ricart-Agrawala):** Does not scale well → too many messages.
- **Quorum-based (Maekawa):** Good for large systems, but quorum size must be managed properly.

Strategies for Problems

- **Suzuki-Kasami:** Use token recovery (if token is lost).
- **Ricart-Agrawala:** Use timeouts for unresponsive processes.
- **Maekawa:** Adjust quorum sizes dynamically to avoid bottlenecks.

Takeaway

- **Suzuki-Kasami:** Efficient but token loss is risky.
- **Ricart-Agrawala:** Fair and decentralized but heavy communication.
- **Maekawa:** Lower communication overhead, good for large systems, but careful quorum management needed.

5. Deadlocks

Definition 5.1 (Deadlock)

A deadlock happens when a group of processes are waiting for resources that are already held by others in the same group (quorum based algorithm). This creates a circular wait where no one can continue.

Four Necessary Conditions (Coffman Conditions):

1. **Mutual Exclusion:** A resource can only be used by one process at a time.
2. **Hold and Wait:** A process holding resources can also request more.
3. **No Preemption:** Resources cannot be taken away by force.
4. **Circular Wait:** A closed chain exists where each process waits for a resource held by the next.

👉 If any one of these conditions is missing, deadlock cannot happen.

Ways to Handle Deadlocks

1. **Prevention:** Make sure at least one Coffman condition never happens.
2. **Avoidance:** Check before granting a resource → will it cause a deadlock?
3. **Detection and Recovery (main real solution):** Allow deadlocks, then detect and fix them.
    - Methods: abort processes, preempt resources, or roll back processes.

5.1 Resource Allocation Graph (RAG)

- Graph with two types of nodes: processes and resources.
- Edges show requests and assignments.
- A cycle in the graph → deadlock.

5.1.1 Wait-For Graph (WFG):

- Simplified version of RAG.

- Nodes = processes only.
- Edge from P1 → P2 if P1 waits for a resource from P2.
- Cycle = deadlock.

5.2 Detecting Deadlocks

- **Centralized Detection:** One node collects all info and builds global WFG.
  - Simple, but not scalable, single point of failure.
- **Hierarchical Detection:** System split into regions, each region checks locally.
  - More scalable, but regional coordination is complex.
- **Distributed Detection:** All nodes share partial info and detect deadlocks together.
  - No single point of failure, scalable, but message passing is complex.

5.3 Resolving Deadlocks

- **Process Termination:** Abort one or more processes.
- **Resource Preemption:** Take resources away and give to others (needs rollback).
- **Rollback:** Return one or more processes to an earlier safe state.

System Model for Deadlock Detection:

- Asynchronous processes, message passing.
- WFG used to model system.
- Deadlock = cycle in WFG.
- Resources are reusable, exclusive, one copy each.
- Processes can be Running or Blocked.
- Challenges: keeping WFG updated, detecting cycles correctly, avoiding “phantom” deadlocks.

5.4 Deadlock Models

1. **Single Resource Model:**

    - Each process needs only one resource.
    - Deadlock = cycle in WFG.
    - Simple but unrealistic.
2. **AND Model:**
    - Process requests many resources at once, only runs if all are available.
    - Cycle in WFG → deadlock.
    - But not all deadlocks show up as cycles.
3. **OR Model:**
    - Process requests many resources, runs if at least one is given.
    - Cycle in WFG ≠ deadlock.
    - Deadlock = presence of a **knot** (a set of processes waiting only on each other).

5.5 Deadlock Detection Algorithms

5.5.1 Path-Pushing Algorithm

- Deadlocks are found by building a global **Wait-For Graph (WFG)** across all sites.
- Each site maintains its own local WFG.
- Sites send their local WFG info to others.
- The global WFG is updated, and cycles are checked → cycle = deadlock.

5.5.2 Edge-Chasing Algorithm

- Works by sending special messages called **probes** to track dependencies.
- If a blocked process sends a probe and it comes back to the initiator, a deadlock exists.
- Running processes discard probes (they don't forward them).
- Example:

    - P1 sends probe to P2 → P2 forwards to P3 → if it comes back to P1 → deadlock found.

5.5.3 Diffusion Computation Algorithm

- Based on sending **queries** when processes are blocked.
- A running process discards queries.
- Blocked processes forward queries and send replies back.
- Rules:
    1. First query → process waits until replies come back from all successors before replying.
    2. Later queries → reply immediately.
- Deadlock is detected when the initiator gets all replies back.

5.5.4 Global State Detection Algorithms

- Use the idea of taking a **consistent snapshot** of the distributed system.
- Snapshot does not freeze computation, but still captures a correct picture of system state.
- If deadlock exists before snapshot, it will appear in the snapshot.
- Then, detect deadlocks by analyzing the snapshot (checking for cycles).

5.5.5 Deadlock Detection in Dynamic and Real-Time Systems

- In **dynamic systems** (like cloud), resources change often → must constantly update detection info.
    - Problems: false detections, outdated info, frequent changes.
- In **real-time systems**, detection must be very fast and lightweight, because timing is critical.
    - Goal: detect and resolve deadlocks quickly with minimal delay.

6. Consensus

7.1 Introduction

Safety and liveness are two key properties of distributed systems.

- **Safety** means the system does not make wrong decisions.
- **Liveness** means the system eventually makes progress.

Consensus means reaching an agreement on a value.

7.1.1 Importance

In distributed databases and replication, it's important to keep replicas consistent, or at least make sure they eventually converge to the same state. The importance of consensus depends on perspective:

- An SQL programmer may not care about consensus.
- A database administrator (DBA) must care.

Consensus is also important for other coordination tasks, like leader election or resource allocation.

Consensus algorithms are widely used in **blockchains and cryptocurrencies**, where nodes must agree on whether a transaction is valid. Examples:

- **Proof of Work** (Bitcoin)
- **Proof of Stake** (Ethereum)

7.2 Mechanisms for Consensus

There are different ways to reach consensus:

- **Majority agreement**: Algorithms like **Paxos** and **Raft** require most nodes to agree on a value.
- **Leader-based coordination**: One leader drives the agreement.
- **Voting and Quorums**: Nodes vote, and a quorum (subset of nodes) must agree. A quorum is not always the full majority, just enough to guarantee overlap.
- **Handling failures**: Consensus algorithms must tolerate node crashes or network issues using timeouts and retries.

7.3 Safety and Liveness

- **Safety** → "Nothing bad happens." The system must stay correct, even if failures happen.
- **Liveness** → "Something good eventually happens." The system must make progress and reach a decision, even under failures or delays.

7.3.1 FLP Impossibility Theorem

Definition: In an asynchronous system (messages can be delayed, nodes may fail), it is impossible to design a consensus algorithm that guarantees **both safety and liveness** at the same time. This was proven in 1985 by Fischer, Lynch, and Paterson.

Proof idea

1. **Start**: Nodes need to agree on a value.
2. **Indecision**: At some point, no node has enough information to decide. Delays or failures can prolong this.
3. **Failure case**: If one node crashes or its messages are delayed forever, other nodes may wait forever.
4. **Conclusion**: Nodes cannot tell the difference between a crashed node and a slow one.
    - If we want **liveness** (progress), we may sacrifice **safety**.
    - If we want **safety** (correctness), we may lose **liveness**.

Implications

FLP shows that consensus algorithms must make **trade-offs**. Real-world systems like Paxos and Raft handle this by making practical compromises.

7.5 Paxos

Paxos is a family of consensus algorithms for unreliable networks. First proposed by Leslie Lamport in 1989, fully described in 1998. It assumes a fully asynchronous system, meaning nodes may start in different states.

Roles in Paxos:

- **Proposers**: Suggest values.
- **Acceptors**: Accept values if they follow the rules.

- **Learners**: Learn the final agreed value.

1. Phase 1

- **Prepare**: Proposer picks a proposal number and sends a *Prepare* message to a quorum (majority) of Acceptors.
- **Promise**: Acceptors reply with a *Promise* if the number is higher than any seen before.

2. Phase 2

- **Accept**: If the Proposer receives Promises from a majority, it sends an *Accept* request with the value.
- **Accepted**: Acceptors accept the value if they haven't already promised a higher number.

Fault tolerance: If there are `n` nodes and up to `f` faulty nodes, Paxos works as long as `n ≥ 2f + 1`.

7.5.2 Challenges and Applications

- Paxos is **hard to understand and implement**.
- Requires many **messages**.
- Network partitions and failures can block progress.
- Used in real systems (e.g., Google's Chubby lock service).

7.6 Key Takeaways

- Consensus is essential for coordination in distributed systems.
- It is a **fundamental problem** in distributed computing.
- Real-world algorithms (like Paxos and Raft) balance between **safety and liveness** because of FLP impossibility.

7. Raft

8.1 Logs use case

A **log** is just a sequence of records. It only makes sense if the records are in order. The order matters: if we apply operations in one order we may get one result, but if we change the order, we may get a different one.

In distributed systems, keeping logs consistent is not easy. On a single computer, timestamps are usually enough, but in distributed systems clocks are not perfectly synchronized. This means we need a more advanced way to decide the order of operations, usually by sequencing them (not just relying on timestamps).

8.2 Consensus (Again)

Consensus is the foundation of **consistency**, **fault tolerance**, and **coordination** in distributed systems. It allows the system to work reliably and predictably, even when there are failures or network problems.

8.2.1 Why is it relevant

One main reason is **convergence**. Consensus prevents replicas from diverging (becoming different). Instead, replicas must eventually converge to the same value.

But the question is: *which replica should others follow?*

The answer is: **Consensus decides that**.

Paxos and paper: For a long time, Paxos was the standard solution. But developers realized it was too complex to implement correctly. It was very detailed on paper, but in practice it was hard to use because many important aspects were left to the implementer.

8.3 Raft

Raft solves consensus in a simpler way by using an **elected leader**.

- Each node in a Raft cluster can be either a **leader** or a **follower**.
- A follower can become a **candidate** and try to become leader through an election.
- The leader regularly sends a **heartbeat message** to followers to show it is still active.
- If followers stop receiving heartbeats within a timeout, they become candidates and start a new election.

Raft assumes that all nodes know about each other.

8.3.1 Key points

- **Leader election**: Raft uses randomized timeouts to avoid conflicts in elections.
- **Log replication**: The leader replicates its log to all followers.
- **Safety and liveness**: Raft ensures both correctness (safety) and progress (liveness), even when failures happen.

8.3.2 Log replication

Log replication is central to Raft. The goal is to make sure all nodes have the same log in the same order.

- The leader manages the log and sends updates to followers.
- The log contains commands that must be applied in the same order by all nodes.
- Only the leader can add new entries.
- An entry is considered *committed* once it is copied to the majority of nodes.
- If the leader crashes, Raft ensures a new leader is elected, and that it has the same log as the old one.

8.3.3 Leader election

Nodes in Raft can be in one of three states:

- **Leader**: Manages log replication.
- **Follower**: Accepts the leader's log.
- **Candidate**: Competes to become leader.

How a new leader is chosen:

1. If the current leader crashes or becomes unreachable, a new election starts.
2. Followers turn into candidates if they don't receive a heartbeat within the election timeout.
3. Once elected, the leader starts appending new log entries and replicating them.

A log entry is confirmed (committed) when it is stored by the majority of nodes. Followers always trust the current leader's log to keep consistency.

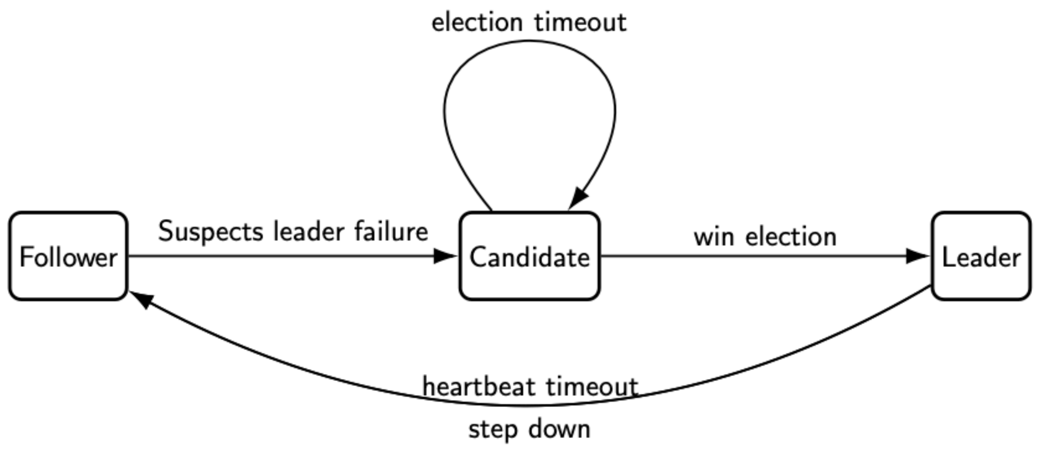


Figure 8.1: State diagram

8. CAP Theorem

Definition 9.1 (CAP Theorem)

A distributed data store cannot guarantee all three of the following at the same time:

- **Consistency:** Every read gives the latest write (or an error if it cannot).
- **Availability:** Every request gets a response, but it may not always be the most recent write.
- **Partition tolerance:** The system keeps working even if some messages between nodes are lost or delayed.

This is also called **Brewer's theorem**, introduced by Eric Brewer in 2000.

When a network partition (communication failure) happens, we must choose:

- **Cancel the operation** → reduces availability but keeps consistency.
- **Continue the operation** → keeps availability but risks inconsistency.

As Prof. Dazzi said: *“The CAP theorem implies that during a network partition, we must choose between consistency and availability.”*

9.1 Trade-off

The CAP theorem shows the compromises in distributed systems. Understanding it is important for good design. Misunderstanding can lead to poor or wrong design choices.

The theorem suggests three types of distributed systems:

- **CP** → Consistent when partitioned.
- **AP** → Available when partitioned.
- **CA** → Consistent and Available when no partitions exist.

The last one (CA) is not realistic in true distributed systems, because network partitions always happen at some point.

Both AP and CP systems can still offer some level of consistency, availability, and partition tolerance, but never all three fully at once.

9.2 Consistency

AP systems only give **best-effort consistency**. This means they will eventually become consistent, but not always immediately.

Examples: **web caching** and **DNS**.

Types of consistency:

- **Strong consistency** → All users see the same data at the same time.
- **Weak consistency** → Reads may give outdated (stale) data.
- **Eventual consistency** → If no new updates happen, eventually everyone will see the latest value.

9.3 Example Scenarios

9.3.1 Dynamic Tradeoff – Dynamic Reservation

In an airline reservation system:

- When many seats are still free, it's okay to work with slightly outdated data, because availability is more important.
- When only a few seats are left, accuracy (consistency) becomes critical to avoid overbooking.

This shows that sometimes we cannot have both strong consistency and guaranteed availability, but we can improve tolerance to network problems by adjusting based on the situation.

9.4 Partitioning

- **Data Partitioning** → Different data may need different levels of consistency and availability.

  Example:

  - Shopping cart → high availability, small mistakes are acceptable.
  - Product info → availability is important, minor differences are okay.
  - Checkout, billing, shipping → must be consistent.

- **Operational Partitioning** → Different operations may need different levels of consistency and availability.

  Example:

  - Login → high availability, minor mistakes are acceptable.
  - Transaction → must be consistent.

In general:

- Reads (queries) → need high availability.
- Writes (purchases/updates) → need high consistency, often require locks.

Other types:

- **Functional Partitioning**
- **User Partitioning**
- **Hierarchical Partitioning**

9.5 PACELC Theorem

Definition 9.2 (PACELC Theorem)

PACELC extends CAP by also considering **latency** when there is no partition.

- **P** = Partition tolerance
- **A** = Availability
- **C** = Consistency
- **E** = Else (when no partition happens)
- **L** = Latency
- **C** = Consistency

So PACELC says:

- **If there is a Partition** → must choose between **Availability (A)** and **Consistency (C)**.
- **Else (no partition)** → must choose between **Latency (L)** and **Consistency (C)**.

9. Replication

Definition 1 (Replication)

Replication means keeping multiple copies of the same data across different systems to improve reliability, fault-tolerance, and access. The goals are:

- reduce latency,
- increase availability,
- support more reads at the same time.

The challenge is keeping all replicas consistent when data changes. Main approaches are:

- single-leader,
- multi-leader,
- leaderless replication.

1 Leaders and followers

Each copy of the database is called a replica. Keeping replicas consistent is key.

- In leader-based replication, one replica is chosen as the leader.
- Clients send all writes to the leader, which updates its data.
- Followers then update their copies by reading the leader's replication log.

10.1.1 Synchronous vs Asynchronous Replication

- **Synchronous** → The leader waits for all followers to confirm before confirming the write.
    - Pro: followers always have the latest data.
    - Con: if a follower is slow, the system slows down or may stop.
- **Asynchronous** → The leader confirms the write immediately, without waiting for followers.
    - Pro: system stays fast and responsive.
    - Con: followers can lag behind, and if the leader crashes, some data may be lost.
- **Semi-synchronous** → The leader waits for at least one chosen follower (usually in another location) to confirm, while others are asynchronous.
    - Pro: ensures at least one follower is always up-to-date.
    - Con: more complex setup.

10.2 Failure

- **Node failures** → Nodes may fail due to errors or maintenance. The goal is to keep the system running smoothly.
- **Follower failure (catch-up recovery)** → Followers can catch up using the replication log.
- **Leader failure (failover)** → A follower is promoted to leader (manual or automatic).
    - Issue: in asynchronous systems, some data might be lost.

10.2.1 Replication logs

- **Statement-based replication** → Replicates SQL statements from leader to followers. Problem: not always consistent (non-deterministic functions, auto-increments).
- **Write-ahead log (WAL) shipping** → Replicates database change logs. Problem: requires same database engine and setup on all replicas.
- **Logical log replication** → Replicates logical data changes.
  - Pro: flexible.
  - Con: requires custom coding and expertise.
- **Trigger-based replication** → Uses database triggers to replicate changes.

10.3 Eventual Consistency

In eventual consistency, followers may lag behind the leader, but eventually all replicas match. This is useful when:

- many reads but few writes,
- we want to reduce load on the leader,
- we want fast local reads from nearby replicas.

10.3.1 Read-after-write consistency

Guarantees that a user can immediately see their own writes, even if others don't see them yet. Often done by reading user data directly from the leader.

10.3.2 Monotonic reads

Guarantees that once a user has seen a value, they won't later see an older one. Achieved by always reading from the same replica.

10.3.3 Consistent prefix reads

Guarantees that writes are seen in the same order they happened. Prevents "causality violations" (e.g., seeing an answer before the question).

10.4 Multi-Leader Replication

- Multiple leaders accept writes at the same time.

- Useful in global systems where each region has its own leader.
- Avoids bottlenecks of single-leader systems.

10.4.1 Collaborative editing

- Each user's changes are applied locally, then replicated.
- If two users edit the same part at the same time, conflicts may happen.
- System must resolve conflicts and notify users.

10.4.2 Conflict avoidance

- Route all writes for one user to the same leader.
- Use nearest datacenter as leader for lower latency.

10.4.3 Topologies

- **Circular** → each leader forwards to the next.
- **Star** → root leader forwards to all.
- **All-to-all** → every leader shares with every other leader.

  In all cases, replicas must be monitored to check how far behind they are.

10.4.4 Concurrent writes

When multiple leaders accept writes to the same data:

- **Last write wins** → keep the last one.
- **Most recent timestamp wins** → based on clocks (hard if clocks are not synced).
- **Merge values** → combine data using happens-before order.

10. Partitioning

Partitioning is needed when we deal with very large data sets and high query loads. It splits a big dataset into smaller, easier-to-handle parts. This makes the system faster and more efficient. Partitioning can be:

- horizontal (splitting rows),

- vertical (splitting columns), or
- functional (splitting by use or function).

Different technologies use different names:

Shard (MongoDB, Elasticsearch, Cassandra), Region (HBase), VBucket (Couchbase), Tablet (Bigtable), VNode (Riak, Cassandra).

Partitioning concepts

Each piece of data must belong to a partition. This can be done using hashing, ranges, or lists. Each partition works like a small database and can live on a different node, helping with scalability.

Combining partitioning with replication

Partitioning is often combined with replication to increase performance and reliability. Replication means storing multiple copies of data on different nodes, so it's still available if one node fails. Together, partitioning and replication provide high availability and fault tolerance.

Ways to combine them:

1. A node can store more than one partition, and each partition can be stored on multiple nodes.
2. Leader–follower setup: one node (leader) handles writes, followers copy the data.
3. Keep replication and partitioning as separate processes.

The goal is to spread data and workload evenly. If not, some nodes become overloaded ("hot spots"). Random partitioning can avoid hot spots, but it makes finding data harder, since you may need to query all nodes.

Key-range partitioning

A simple way is key-range assignment (like books in a library ordered by author). This makes it easy to find data, but may cause uneven distribution if some ranges are much more popular.

The ranges of keys are not necessarily evenly spaced, because your data may not be evenly distributed.

- One volume per two letters can lead to uneven sizes

- Partition boundaries need to adapt to data

11.2 Avoiding hot spots

With key-range partitioning, certain keys (like today's data in a sensor system) can overload one partition. A fix is to modify the key (e.g., add a prefix like sensor ID) to spread the load more evenly.

Hash partitioning

Hash functions help determine the partition for a given key. Good hash functions make skewed data uniformly distributed. This balances data better, but:

- It's bad for range queries, since related data is spread randomly.
- Poor hash functions can still create hot spots.

Secondary indexes

Secondary indexes help with hash partitioning. They add another layer to map secondary keys (like color, brand) to primary keys, and then locate data:

- Document-based (local indexes): partitioned by document ID, but queries may need to scan all partitions.
- Term-based (global indexes): global index across all partitions, more efficient for queries.

Rebalancing

Rebalancing means moving data between partitions to keep distribution fair, especially when new nodes are added or access patterns change.

Problems: if partitioning is done by simple `h(key) mod #nodes`, adding or removing a node forces almost all data to move. A better method is Distributed Hash Tables (DHTs) like Kademlia or Chord, which reduce movement by fixing key space ahead of time.

Dynamic partitioning Adapts to changes automatically or manually:

- Automatic: system rebalances itself. Less work for humans, but may be costly or unpredictable.
- Manual: admin controls rebalancing, often better for complex systems.
- Hybrid: mix of both.

Routing Routing decides which partition should handle a query. It can be done by:

- a routing tier (forwarding queries),
- partition-aware clients (clients know where data is),
- or querying all partitions (slow but sometimes needed).

Takeaways

Partitioning makes systems scalable by spreading data across nodes. With replication, it also improves reliability. But it introduces issues like hot spots and rebalancing. Solutions include hash partitioning, secondary indexes, and dynamic partitioning. A well-designed partitioning strategy helps achieve high availability, fault tolerance, and scalability.

11. Transaction

Definition 12.1 (Transaction)

A transaction is a group of operations that run together as **one single unit of work**.

- It can include reads, writes, and updates.
- A transaction must be **atomic**: either all operations succeed, or none do.
- While a transaction is running, other transactions cannot change the same data until it finishes.

Transactions are very important in distributed systems because they help keep **data consistent across multiple nodes**. But they also come at a cost, since even simple things like a lock (mutex) are harder to implement in a distributed system.

12.1 Error Handling

Transactions guarantee **all-or-nothing**, which makes error handling easier — no need to deal with partial failures.

12.2 Databases

Most SQL databases support transactions. Some NoSQL databases may not, or may support them in limited ways.

12.3 ACID Properties

- **Atomicity**: All operations succeed or none do.
- **Consistency**: Data is valid before and after the transaction.
- **Isolation**: Transactions run as if they are alone, not interfering with each other (like they run serially).
- **Durability**: Once committed, changes are permanent.

The term **ACID** was introduced by Jim Gray in 1983.

12.3.1 Durability

Durability means that once a transaction is committed, its results should not be lost.

- In reality, perfect durability is impossible (hardware faults, power loss, firmware bugs...).
- Systems reduce the risk by writing to disk, replicating to other machines, and making backups.

12.3.2 Single-Object and Multi-Object Transactions

- If a transaction affects **only one object**, it is simple.
- If it affects **many objects**, it can cause performance issues and deadlocks.

Databases solve concurrency problems using **isolation**. The strongest form, **serializable isolation**, makes transactions run as if they were one after another.

- This ensures correctness, but is slow.
- Weaker isolation levels are faster, but harder to reason about and may introduce subtle bugs.

12.4 Avoiding Transactions

12.4.1 Read Committed

In this isolation level, transactions only **read and overwrite committed data**.

- This is the **default** in most databases.

Definitions:

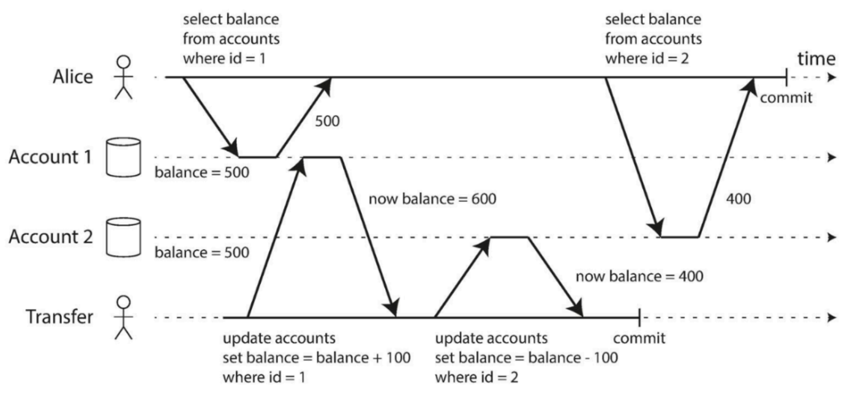
- **Dirty Read**: When a transaction reads uncommitted data from another transaction.
- **Dirty Write**: When a transaction overwrites uncommitted data from another transaction.

In **Read Committed**, dirty reads and dirty writes are **not allowed**.

- This makes it more reliable than no transactions at all, but still cheaper than full transactions.

However:

- **Non-repeatable Read**: A transaction may read the same data twice and get different results, because another transaction updated it in between. This can still happen in Read Committed.



12.4.2 Snapshot Isolation

Snapshot isolation is **less strict than serializable isolation**, but **stricter than Read Committed**.

- At the **start** of the transaction, the database takes a **snapshot** (a consistent view of the data).
- The transaction then **reads from this snapshot**, so it always sees the database as it was when it started.
- At the **end**, the transaction writes its updates to the database.

This way, transactions don't get confused by changes made by others while they are running.

12.5 Write Skew and Phantoms

Definition 12.5 (Write Skew)

A **write skew** happens when:

1. Two transactions read the same data.
2. Based on that data, they update **different objects**.
3. Both commit successfully.

   → The result is a **conflict**, because each transaction assumed the data hadn't changed.

12. Failure Models

- Algorithmic **correctness** usually assumes that communication works perfectly.
- **Robustness** means staying correct even when real failures happen.

To get both correctness and realism, we need clear models of how channels and nodes can fail.

- In token-based mutual exclusion, if a message is lost, the system can stop working.
- If a token is duplicated, correctness is broken.
- So, the behaviour of the channel affects what an algorithm can guarantee.

What are the Failure Models?

A **failure model** is a formal description of what kinds of incorrect behaviour are allowed for processes and channels, and what timing assumptions exist. It defines what an algorithm can and cannot detect when faults happen.

Use: it helps (i) prove correctness or impossibility, and (ii) decide the minimum coordination needed to reach a guarantee (such as delivery rules or consensus).

Why Failure Models?

- They make assumptions clear and easy to compare.
- Allows proofs and impossibility results.

- They combine the behaviours of nodes and channels.

Channel vs. Process Failures

- **Channel:** message loss, duplication, reordering, or corruption.
- **Process:** crash, missing sends/receives, timing problems, or arbitrary behaviour.
- The whole system model = channel model + process model.

Overview of 5 Failure Classes:

- **Crash-stop:** a process stops forever; temporary memory is lost.
- **Crash-recovery:** a process may restart; stable storage remains; repeating actions may happen.
- **Omission:** temporary drops of sends or receives without stopping (for example, congestion).
- **Timing/performance:** actions or messages take longer than expected; in a fully asynchronous system this looks the same as a failure.
- **Byzantine:** completely unpredictable or malicious behaviour, including giving different answers to different parties.

Crash Failures

- A process stops and does not come back.
- This is the simplest meaningful failure model and often used as a baseline.
- It interacts with time assumptions (asynchronous vs partially synchronous systems).

Why time assumptions matter?

- In a fully asynchronous system, a timeout cannot tell whether a process is slow or crashed.
- In partial synchrony, time limits eventually become reliable, so timeouts start working.
- Failure detectors (like Ω, ⋄S) capture these timing assumptions.

Omission Failures

- Temporary loss of sends or receives without the process stopping.
- Common causes: buffer overflow, congestion, network card or driver drops.
- Typical fix: at-least-once delivery with deduplication.

Timing Failures

- Operations or messages take longer than expected.
- This creates false suspicions in asynchronous systems.
- Partial synchrony restores reliable timeouts after the system stabilizes.

Byzantine Failures

- Completely unpredictable behaviour: sending conflicting messages, forging data, or replying selectively.
- Safety requires authenticated voting and larger quorum sizes.
- General rule: to handle f Byzantine nodes, use 3f+1 replicas.

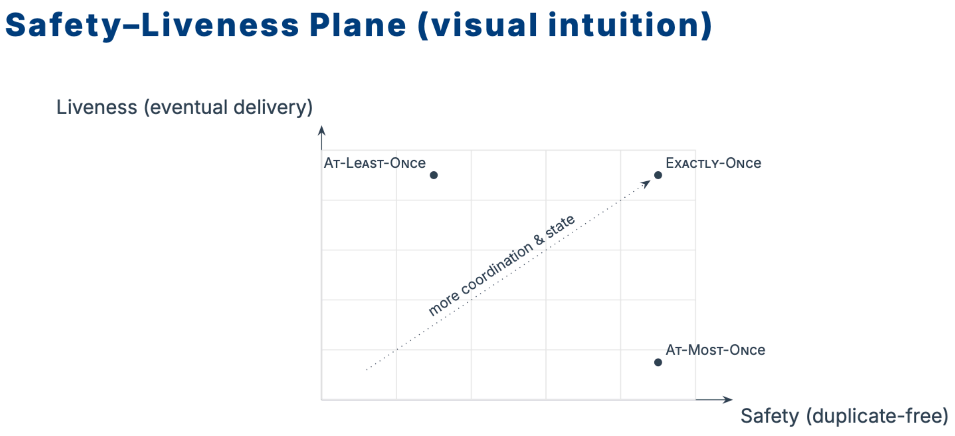
Channel Models (Attiya–Welch)

- Perfect: no loss, no duplication, no reordering (idealised case).
- Reliable FIFO: no loss or duplication, messages arrive in the same order per link.
- Reliable (non-FIFO): no loss or duplication, but order is not guaranteed (needs sequence numbers).
- Fair-loss: messages may be dropped, but if a sender retries forever, infinitely many will be delivered.
- Unreliable: loss, duplication, and reordering can all happen (common on wide-area networks).

What Do We Mean by "Delivery Semantics"?

- Message delivery (channel-level): whether a message arrives 0, 1, or more times.

- Effect delivery (end-to-end): whether the application's effect happens 0, 1, or more times.
- Scope: can be per-link, per-partition, per-key, or end-to-end. Guarantees are always tied to a scope.
- Terms like AT-MOST-ONCE, AT-LEAST-ONCE, EXACTLY-ONCE refer to message or effect delivery under a specific scope and failure/time model.


Safety–Liveness Plane (visual intuition)
Liveness (eventual delivery)
At-Least-Once
Exactly-Once
more coordination & state
At-Most-Once
Safety (duplicate-free)

Definition: AT-MOST-ONCE (specification and practice)

- Spec (channel): a message is delivered either 0 or 1 time; loss is allowed, no duplicates.
- Liveness: not guaranteed; a message might never arrive.
- Typical mechanisms: send once without retry; or dedup at the receiver without retrying.
- Use when: losing a message is acceptable or upper layers can handle missing data.

Definition: AT-LEAST-ONCE (specification and practice)

- Spec (channel): if the sender keeps retrying and the receiver stays up, delivery happens eventually; duplicates are allowed.
- Safety: duplicates must be handled at the receiver.
- Typical mechanisms: acknowledgements with retries, backoff, message IDs for deduplication.

- Use when: you need guaranteed delivery but can handle duplicates using idempotence or deduplication.

Definition: EXACTLY-ONCE (perceived, end-to-end)

- Spec (effects): the effect happens exactly once; eventually it must occur. Multiple message deliveries may happen, but the effect is applied once.
- Note: in a fully asynchronous system with failures, EXACTLY-ONCE cannot be a channel guarantee; it is achieved end-to-end through idempotence, deduplication, or coordinated commit.
- Typical mechanisms: idempotent operations, dedup keys, transactional inbox/outbox, atomic write and publish, barriers.

Ambiguity: Delay vs. Loss

- Without time bounds, delay looks the same as message loss.
- Result: proving progress requires partial synchrony or shifting guarantees to end-to-end idempotent effects.

Guarantee vs. Scalability

- Stronger guarantees require more coordination and more state.
- Coordination reduces parallelism and increases latency.

Why Idempotence and Commutativity?

In real distributed systems, messages can be retried, duplicated, or arrive in the wrong order.

The main idea: if operations are idempotent and commutative, the system stays correct and scalable even with retries, duplicates, and reordering.

Key questions:

- What happens if the same operation is applied many times?
- What if operations happen in a different order?

Definition: an operation is Idempotent if doing it more than once has the same effect as doing it once: `f(f(x)) = f(x)`

Commutativity: order does not change the result: `f(a, b) = f(b, a)`

Associativity: grouping does not change the result: `f(f(a, b), c) = f(a, f(b, c))`

CALM Connection: Monotonicity Enables Scalability

CALM Principle: a program is consistent and does not need coordination if and only if it is monotonic.

Monotonicity: once something becomes true, it stays true as more data arrives.

Result: monotonic programs do not require coordination to stay consistent.

Examples:

- monotonic: set union, max, min, logical OR, counting unique items
- non-monotonic: set difference, leader election, enforcing global uniqueness

Perfect order does not exist. Scalable distributed systems work by handling disorder through clear assumptions, controlled guarantees, and well-structured operations.

13. Distributed Messaging

Messaging systems allow **loose coupling** and **asynchronous communication** between distributed components. They act as the **glue** that connects different parts together.

In distributed computing, messaging systems can support many communication styles, such as request/reply, publish/subscribe, and event sourcing. A simple messaging system is often not enough, because it must also handle **delays, message ordering, lost messages, scalability, and fault tolerance**.

16.1 Message Passing Mechanisms

- **Request-response**: a client sends a request to a server, and the server replies.
- **One-way messaging**: a client sends a message, but the server does not reply.

- **Publish-subscribe**: clients subscribe to a topic, and the server sends all new messages on that topic to them.

Publish-subscribe is useful because it allows clients to receive updates from many services, group them into categories, and give them different meanings.

16.1.1 Adapting Publish-subscribe to Point-to-point

Using publish-subscribe for direct point-to-point communication is messy. If every client subscribes to every other client, the system becomes complex and inefficient.

A better way is to add a **server in the middle** to forward messages to the right recipient. This makes the design simpler but creates a **central bottleneck** that is not scalable and can fail.

Another option is to give each client its own queue, but this also doesn't scale well and may still cause single points of failure.

16.2 Message Brokers – Kafka

Message brokers are systems that receive messages from producers and deliver them to consumers. They add features like message storage, routing, and filtering.

Kafka is a distributed message broker designed for **high throughput, fault tolerance, and scalability**. It is used for log aggregation, stream processing, and event sourcing.

Messages in Kafka are stored in **topics**, which are like tables in a database. Each topic is split into **partitions**. A message has a **key, value, and timestamp**, and is written in an **append-only log** inside a partition.

Messages are written in **batches** (groups of messages) to improve throughput and reduce latency. Larger batches mean higher efficiency, but messages may take longer to be delivered.

Partitions allow a topic to be **spread across multiple servers**, making it scalable. However, this means Kafka does not guarantee a **global message order** across all partitions. Order is only guaranteed **within a single partition**. If global order is required, a topic can be limited to one partition, but this reduces scalability. To balance between order and scalability, partitions can be split logically (e.g., odd user IDs in one partition, even in another).

16.2.1 Producers and Consumers

- **Producers** write messages to topics.
- **Consumers** read messages from topics.

Both producers and consumers can be scaled horizontally.

Each message has an **offset**, a unique number in a partition that lets consumers track what they have read. Offsets always increase.

To keep order, only **one consumer in a group** can read from a partition at a time. This means the maximum scalability is limited by the **number of partitions**, not the number of consumers.

16.2.1.1 Rebalancing

The number of partitions is normally fixed when the topic is created. Kafka supports **rebalancing**, which allows adding more partitions later. But this requires stopping, recalculating, and restarting, which is messy.

16.2.1.2 Brokers and Clusters

- A single Kafka server = a **broker**.
- A group of brokers = a **cluster**.

Each partition has a **leader broker**, which handles all reads and writes for that partition. Other brokers keep **replicas** as backups. If the leader fails, a replica becomes the new leader.

Consumers always read from the **leader broker** (not replicas), which ensures consistency.

16.2.1.3 Retention

Kafka allows setting a **retention policy** for each topic:

- Keep messages for a certain **time** (e.g., 7 days).
- Or keep messages until a **size limit** is reached.

When the limit is reached, old messages are deleted.

16.2.1.4 Multiple Groups

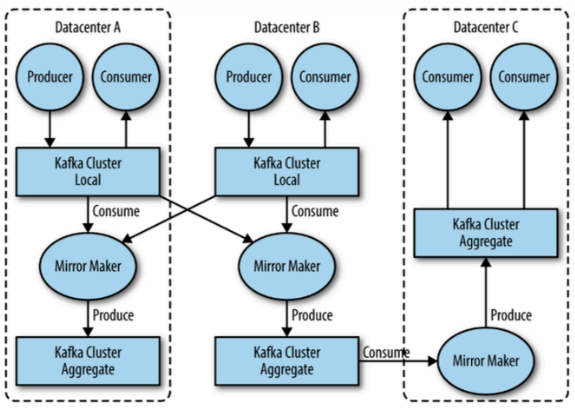


Figure 16.1: Clusters and `MirrorMakers` schema

As Kafka systems grow, it is often useful to have **multiple clusters**. This helps with **separation, isolation, and fault tolerance**.

When working with **multiple datacenters**, messages may need to be copied between them. Kafka provides a tool called **MirrorMaker** for this. MirrorMaker is basically a **Kafka consumer + producer connected together with a queue**: it consumes messages from one cluster and sends them to another.

16.3 Using Kafka

16.3.1 Producers

From the producer side, some important questions must be asked:

- Is every message critical?
- Can we accept duplicate messages?
- Do we need low latency (speed) or high throughput (volume)?

The answers decide how the producer API should be used. The producer can send messages in three main ways:

- **Fire-and-forget** – the producer sends the message and doesn't wait for a response. Kafka will retry failed messages, so most will arrive, but some may still be lost.
- **Synchronous** – the producer sends a message and waits for a reply. This is the **safest** but also the **slowest** method. The `send()` call returns a `Future()` object, and we can use `get()` to wait for confirmation.

- **Asynchronous –** the producer sends a message with a **callback** and continues working. The callback runs later, either on success or on error.

16.3.1.1 Message Record Format

A Kafka message (ProducerRecord) usually contains:

- **Topic**
- **Key** (optional)
- **Partition** (optional)
- **Value**

Before sending, the producer **serializes** the key and value so they can be sent over the network. If a **partition** is specified in the ProducerRecord, the partitioner uses it directly. A separate **thread** in the producer is responsible for sending batches of records to the correct Kafka brokers. After successful delivery, the broker returns **Metadata** about the record.

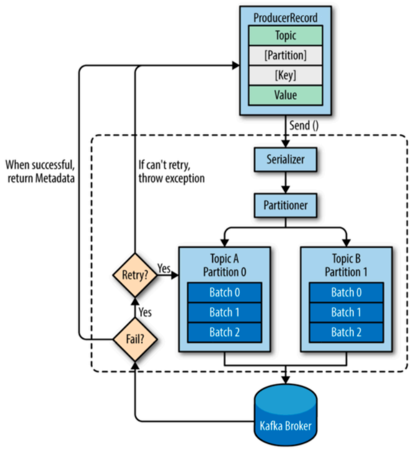


Figure 16.2: Sending `ProducerRecord`

16.3.2 Consumers

When working with Kafka messages, we must consider that **producers can write messages faster than consumers can read them**. If this happens, consumers may fall behind and even run out of memory. To avoid this, we need a way to **scale consumers**.

Kafka solves this with **consumer groups**. When multiple consumers belong to the same group and subscribe to a topic, Kafka splits the partitions among them. Each consumer gets a different subset of partitions. If there are **more consumers than partitions**, the extra consumers will just stay **idle** (doing nothing).

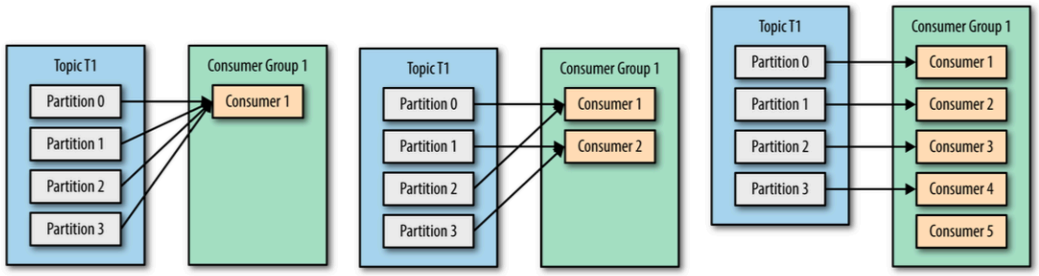


Figure 16.2: Partitions are assigned to Consumers to have a fair distribution.

A good practice is to create **many partitions** so the system can scale when more consumers are added later. Sometimes, **multiple applications** need to read the same data. This is solved by giving each application its **own consumer group**. That way, each group receives the full set of messages.

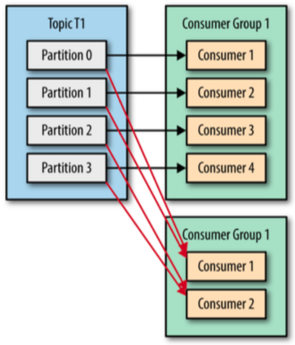


Figure 16.3: Multiple Consumer Groups

16.3.3 Rebalancing

Moving partitions from one consumer to another is called a **rebalance**. Rebalances are useful because they give the consumer group **high availability and scalability**. But they are also **undesirable** because, during a rebalance, **consumers stop reading messages**. This creates a short period where the whole group cannot process data. When partitions move, a consumer also **loses its current state**. If it was caching data, it needs to refresh it again, which can temporarily slow the application.

When is a rebalance needed?

- A new consumer joins the group
- A consumer leaves the group
- A consumer is marked as dead
- The number of partitions for a topic changes

The most important case is when a **consumer dies**. If it stops sending **heartbeats** for too long, its session will time out. After a few seconds, the group coordinator marks it as dead and triggers a rebalance.

During that gap, no messages from the dead consumer's partitions are processed. If a consumer closes **cleanly**, it notifies the coordinator before leaving, so the rebalance happens immediately (faster and cleaner).

16.4 Kafka Strengths

Kafka can handle:

- **Multiple producers**, writing to the same or different topics
- **Multiple consumers**, either in the same group or different groups

Unlike many queue systems, Kafka allows **several consumers to read the same stream independently**. Once one consumer reads a message, it is **still available** for others. Also, Kafka uses **disk-based retention**. This means messages stay stored for a while, so consumers don't need to work in real-time. They can **catch up later** at their own pace.

14. Distributed File System

Another key part is a **Distributed File System (DFS)**. This allows data to be stored across many machines while appearing to the user as if it's all in a single system. The DFS also provides **location transparency**, meaning the user doesn't need to know where the data is physically stored.

A distributed filesystem (DFS) provides a storage layer that:

- Shows one logical namespace even though data is stored on many machines;
- Separates metadata (names, permissions, directories) from data blocks or objects;
- Offers configurable levels of consistency, availability, and durability;
- Increases capacity and throughput by distributing both metadata and data across multiple servers.

Distributed Filesystem vs Object Storage

DFS: hierarchical directory tree, rename is a core operation, directory-level rules.

Object storage: flat namespace, simple PUT/GET/DELETE, directory listings often only eventually consistent.

Both use distributed metadata and distributed data, but follow different rules and guarantees. Modern systems often mix ideas (for example, CephFS on RADOS, HDFS on top of object stores).

Conceptual motivation

DFS show the main trade-offs of distributed systems:

- consistency vs availability vs performance,
- centralized vs distributed vs decentralized control,
- replication vs partitioning vs erasure coding.

They involve all architectural layers: control, state, topology, and time model. They show complex interactions between metadata, data placement, consistency, and recovery.

Distributed filesystems separate:

- **metadata:** directory structure, file attributes, permissions, layout;
- **data:** blocks, chunks, objects.

Metadata operations dominate performance: path lookup (often multiple steps), directory listing, rename and concurrency control, workloads with many small files. DFS scalability is mainly a **metadata scalability** problem.

Metadata Service Architectures. Common designs:

- **Single MDS:** simplest but becomes a bottleneck.
- **Active-standby MDS:** improves availability with fast failover.
- **Sharded MDS:** metadata split across several servers.
- **Dynamic subtree partitioning:** directories moved between MDS based on load.

Single Metadata Server (Single MDS)

Definition: One metadata server manages the whole namespace: directories, inodes, permissions, and layout.

Simple architecture: all metadata operations go to one server.

Strong semantics: global total order of metadata operations, easy to enforce tree structure and atomic rename, local locking is enough.

Bottleneck: limited by one CPU and one I/O path, high latency under heavy metadata load, more data servers do not help if the single MDS saturates.

Active-Standby Metadata Servers

Definition: One active MDS handles metadata; standby MDS replicas keep up-to-date copies using journal or log replication.

Goal: fault-tolerance with fast failover

If the active MDS fails:

1. a standby becomes the new active MDS;
2. it replays recent log entries;
3. clients reconnect to it.

Downtime is small (detection + promotion).

Trade-offs

- increases availability but not scalability; still one logical MDS.

- must keep active and standby journals consistent.
- needs reliable failure detection and protection against split-brain.

Sharded Metadata Servers

Definition: The namespace is **partitioned** across multiple metadata servers. Each server manages one shard.

Partitioning strategies

- **Subtree-based:** one directory subtree per MDS.
- **Inode-based:** inodes divided by number range.
- **Hash-based:** hashing the path or inode selects the MDS.

Advantages: higher throughput by adding more metadata servers, better distribution of load.

Challenges

- Path resolution becomes distributed; a single path like “/a/b/c” may involve multiple MDS;
- Rename across shards requires distributed transactions;
- Distributed locking and shared journaling are needed.

Dynamic Subtree Partitioning

Idea: Use subtrees (directories and all their children) as the unit for partitioning. Assign each subtree to one MDS, but allow it to move between MDS servers at runtime.

Why dynamic?

Workload is not even: some directories get very busy (many creates, stats, renames). Static partitioning can cause hotspots: one MDS overloaded while others are idle.

Mechanism

Monitor load per subtree (number of operations, latency, queue length). When a subtree is hot, move it to a less busy MDS:

- transfer metadata and ownership;
- update client routing information;
- keep namespace rules correct during migration.

Two main models: **Subtree-based and Hash-based.** Many systems use a hybrid of both strategies dynamically.

Leases and Delegations To reduce MDS load:

- Clients get **delegations** (temporary control over metadata).
- MDS issues **leases** to allow caching of directory entries or file attributes.

Data Placement & Replication:

Chunking & Striping

Files are divided into large chunks (e.g., 64–256 MiB).

- **Chunking** reduces metadata overhead and improves throughput: fewer RPCs per GiB and efficient sequential I/O.
- **Striping:** chunks (or blocks) spread across multiple data servers to use parallelism.

Trade-offs:

- Large chunks are good for big files but not for many small files.
- Striping increases throughput but also increases risk if servers fail.
- Stripe width must balance parallelism, recovery cost, and hotspot handling.

Topology-aware Placement. Placement algorithms consider:

- rack and node failure domains,
- network locality and bandwidth asymmetry,
- heterogeneous storage

Objective: maximize locality and throughput while minimizing correlated failures.

Typical rule (HDFS/GFS-style):

- 1 replica on the local rack,
- 1 replica on a remote rack,
- 1 replica on a third independent failure domain.

Topology-awareness is essential for durability, availability, and recovery speed.

CRUSH Placement (Ceph)

CRUSH = Controlled Replication Under Scalable Hashing.

Determines object placement without a central metadata table. Encodes hierarchy: host → rack → row → datacenter → region.

Advantages:

- No lookup tables, no MDS bottleneck.
- Stable mapping when the cluster changes slightly.
- Supports weighted placement and failure-domain rules.

CRUSH is an example of a decentralized placement algorithm.

Consistency & Concurrency: Consistency Dimensions in DFS

Consistency in a distributed filesystem covers:

- Metadata consistency, Data consistency, Visibility, Caching rules.

DFS consistency must balance:

- POSIX expectations (open, read, write, close)
- Distributed system realities (latency, network partitions, asynchronous replication)

Close-to-Open Semantics Write visibility rule:

- A client's writes are visible to others only after close().
- A newly opened file sees the latest committed version.

Properties:

- Avoids fine-grained distributed write serialization.
- Relies on cache invalidation at open().
- Weaker than POSIX but better for performance and stateless servers.

Weaknesses:

- Concurrent writes may interleave unpredictably.
- Stale caches if clients fail to close.
- Rename operations may cause unexpected reorderings.

DFS often provide session-level guarantees:

- **Read-your-writes:** a client sees its own writes immediately.
- **Monotonic reads:** once a client reads version v, it never sees v′ < v.
- **Monotonic writes:** a client's writes are applied in order.
- **Writes-follow-reads:** preserves causal order within a session

These guarantees create a causal consistency envelope around weaker global consistency.

DFS typically enforce:

Atomicity: a write of n bytes is all-or-nothing.

Append atomicity: record appends (GFS/HDFS) ensure aligned multi-writer logs.

Write ordering: replicas apply writes in the same sequence.

Cache Invalidation Protocols. Two main models:

- Callback invalidation: MDS notifies clients when cached entries are stale.
- TTL-based caching: entries expire after a timeout; simpler but weaker.

With leases:

- Clients get temporary read or write control.

- MDS must revoke leases before conflicting operations.
- Revocation latency affects system responsiveness.

Correctness of leases depends on timely revocation and client liveness.

Distributed Locking Models. Locks may apply to:

- Inode-level: simple but coarse, serializes entire files.
- Directory-level: important for rename operations.
- Byte-range locking: needed for POSIX compliance; expensive in DFS.
- Path-level locking: treats whole path as a unit for safety.

Challenges:

Preventing distributed deadlocks via canonical ordering.

Migrating locks when directories move between MDS shards.

Interaction with leases and client-side caching.

POSIX assumes:
Single-system image. Strongly consistent file descriptors. Immediate visibility after writes. Synchronous directory updates.

Distributed filesystems provide:

Eventual or session consistency. Delayed visibility due to replication or leases. Asynchronous directory updates. Partial ordering across shards.

MDS typically uses a write-ahead journal:

- Every change (create, delete, rename) is logged before being applied.
- After a crash, replaying the journal rebuilds a correct metadata state.
- Multi-MDS operations use intent records to resolve incomplete work.

Recovery steps:

1. Read journal entries since the last checkpoint.
2. Apply mutations that can safely run multiple times.
3. Resolve partial operations such as rename or move.
4. Rebuild client lease information.

Goal: Reconstruct a state that includes all committed operations and nothing more.

Recovery Throttling Repair actions must not overload the cluster.

Mechanisms:

- Limit the bandwidth used for recovery.
- Prioritize tasks: metadata first, frequently accessed data next.
- Slow down recovery during heavy user I/O.
- Apply rack-level limits to avoid network congestion.

Insight: Uncontrolled recovery traffic can cause more disruption than the original failure.

Design Takeaways

- DFS design balances metadata rules with scalability needs.
- Placement methods impact both speed and durability.
- Recovery bandwidth must be treated as a key system resource.
- No single DFS works best for all workloads; design depends on access patterns.
- Knowing historical designs helps explain modern cloud storage systems.

Design according to constraints, not personal preference.

15. Map Reduce

Distributed computing is the main idea behind Big Data processing. It makes it possible to **split very large tasks into smaller ones**, run them in parallel across many machines, and then combine the results into a final output.

The key to Big Data is **horizontal scaling** – adding more machines instead of making a single machine stronger. This applies to both **strong scaling** (faster with more machines for the same data size) and **weak scaling** (handling more data with more machines).

Instances of **MapReduce** are **Hadoop** and **Spark** (we will discuss them later). Google was the first to introduce **MapReduce** in 2004. It is both a programming model and a system that allows large datasets to be processed and generated in a **parallel and distributed way** on clusters of machines.

Infrastructure Requirements

For Big Data processing, it's not enough to just have powerful CPUs and GPUs. You also need a **fast network** for quick data transfer between nodes. Examples include **Ethernet, Infiniband, or Myrinet**. Sometimes **RDMA (Remote Direct Memory Access)** is used to reduce latency and CPU usage.

14.1 MapReduce Framework

Map `Map((keya, value1)) → list((keyb, value2))`

The Map step processes the input data. It usually does **filtering, sorting, or transformation**.

Reduce `Reduce((keyb, list(value2))) → list((keyc, value3))`

The Reduce step **summarizes and combines** the results from the Map step.

Steps of MapReduce:

1. **Map** – Each worker node runs the Map function on its local data and writes the output to temporary storage.
2. **Shuffle** – Data is redistributed based on keys, so that all data with the same key is sent to the same worker node.
3. **Reduce** – Each worker node processes its group of data (per key) in parallel, producing the final result.

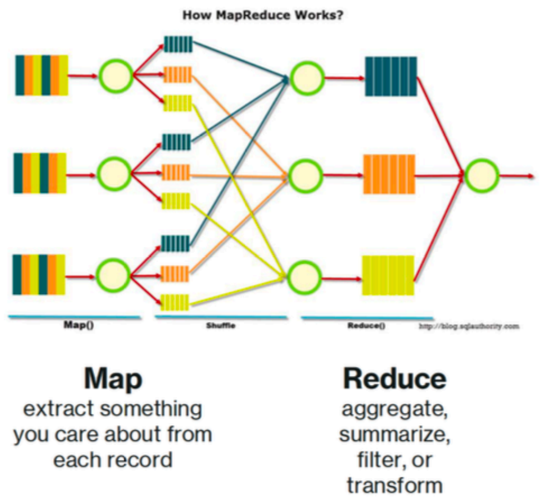

How MapReduce Works?
Map()
Shuffle
Reduce()
http://blog.sqlauthority.com
Map
extract something you care about from each record
Reduce
aggregate, summarize, filter, or transform

Figure 14.1: MapReduce overview

14.2 Running MapReduce

- There is **one JobTracker**, which receives MapReduce jobs from client applications.
- The **JobTracker assigns work to TaskTracker nodes** in the cluster, trying to keep work close to where the data is stored.
- With a **rack-aware file system**, the JobTracker knows which node has the data.
    - If it can't run the job on the exact node, it prefers another node in the same rack (closer = faster).
- If a **TaskTracker fails or times out**, the job is rescheduled on another node.
- TaskTrackers send **heartbeats** to the JobTracker every few minutes to show they are alive.
- The **status of JobTracker and TaskTrackers** can be monitored in a web browser.

14.2.1 Issues

- Each TaskTracker has a **fixed number of slots** for map and reduce tasks. Jobs are assigned based only on free slots, **not on how heavy the tasks are**.
- This can lead to **unbalanced workloads**, where some nodes do much more work than others.

- The system waits for the **slowest node** to finish, so one slow machine can slow down the whole job.
- To fix this, **speculative execution** can run the same task on multiple nodes, and the first result is used.

14.3 Hadoop

Hadoop At first, Big Data systems were built on **Apache Hadoop**, an open-source framework for distributed storage and data processing. Hadoop uses **MapReduce** to process data across many machines in a cluster.

- **Hadoop** is an open-source framework for **distributed storage and processing of big data**.
- It can run frameworks like **MapReduce** across large clusters of computers.
- A **small Hadoop cluster** has one **master node** and many **worker nodes**.
    - The **master node** includes: JobTracker, TaskTracker, NameNode, and DataNode.
    - A **worker node** usually acts as both a DataNode (stores data) and a TaskTracker (runs jobs).
    - Some worker nodes can be **data-only** or **compute-only**.

14.4 Spark

Spark is a **fast and general cluster computing system**. It supports APIs in **Java, Scala, Python, and R**, and uses an optimized engine that can handle many types of execution flows.

It also includes powerful tools:

- **Spark SQL** for SQL and structured data,
- **MLlib** for machine learning,
- **GraphX** for graph data,
- **Spark Streaming** (old, now replaced by **Spark Structured Streaming**).

Lazy vs Eager evaluation

- **Lazy evaluation**: an expression is only calculated when its value is actually needed.

- **Eager evaluation**: an expression is calculated as soon as it is assigned to a variable.

14.4.1 Architecture

- Every Spark application has a **driver program**, which runs the main function and coordinates parallel tasks on the cluster.
- The driver connects to Spark using a **SparkSession**, which represents the connection to the cluster.
- The main Spark abstraction is the **RDD (Resilient Distributed Dataset)**.

Definition (RDD): An RDD is a single, unchangeable (immutable) distributed collection of objects.

- The data is split into **partitions**, which can be processed on different nodes in parallel.
- RDDs can hold **any type of objects** in Python, Java, or Scala — even user-defined classes.

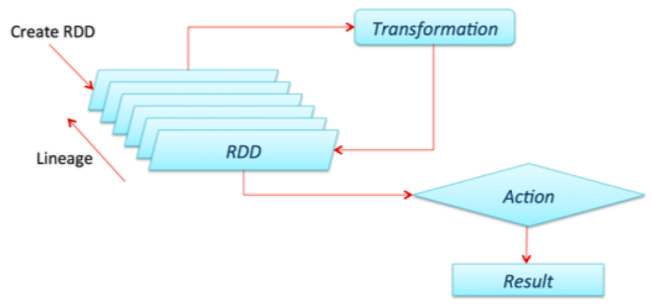


Figure 14.4: RDD schema

Listing 14.2: Creating an RDD

```
data = [1, 2, 3, 4, 5]
distData = sc.parallelize(data)
distFile = sc.textFile("data.txt")
```

- RDDs can be created in two ways:
  1. By **parallelizing** an existing collection in the driver program.

    2. By **loading data** from an external source like a file system, HDFS, HBase, etc.

- RDDs can be **kept in memory (persisted)**, so they can be reused efficiently across many operations.
- RDDs also **recover automatically** if a node fails.

Shared variables in Spark

Normally, when Spark runs a function in parallel on different nodes, it sends a copy of every variable used in the function to each task.

But sometimes, we need variables to be shared across tasks or between tasks and the driver program. Spark supports two special types of shared variables:

- **Broadcast variables**: read-only variables that are **cached on each machine**, instead of sending a copy to every task.
- **Accumulators**: variables that tasks can only **add to** (using an associative operation like sum). Useful for counters and aggregations.

RDD operations There are two main types:

- **Transformations**: create a **new dataset** from an existing one. They are *lazy* (Spark doesn't run them right away — it just remembers them).
- **Actions**: return a **value to the driver program** after running a computation. When an action is called, Spark executes the chain of transformations to produce the result.

*Because RDDs are **immutable** (they cannot be changed), we get three main benefits:*

1. **Consistency** – Since RDDs don't change, Spark can do optimizations and lazy evaluation. We can also always return to the original RDD without worrying about tracking all the changes.
2. **Resiliency** – If a node fails, Spark can rebuild the RDD from the original data, so nothing is lost.
3. **Concurrency** – Many tasks can safely use the same RDD at the same time, since the data will never change. This removes concurrency problems.

16. Actor Model

In 2024, computing is everywhere — on our phones, in our homes, in cars, and in the cloud. This is called the **Compute Continuum**, meaning that the digital and physical worlds are now deeply connected, and computing happens at any time and any place.

15.1 Towards the Actor Model

Because computing is now highly decentralized, we need decentralized models of computation.

The **Actor Model** is one such model. It views **actors** as the basic building blocks of concurrent (parallel) computing.

- When an actor gets a message, it can:
    - Make local decisions
    - Create new actors
    - Send more messages
    - Decide how to respond to the next message
- Actors can change their own private state, but they can't directly change others — they can only communicate through messages.
- This avoids the need for locks.

This model is very useful in modern systems like:

- Self-organizing programs
- Adaptive and collective systems
- IoT and cyber-physical systems
- Edge and fog computing

Fun fact: The Actor Model was introduced by **Carl Hewitt in 1973**, inspired by Alonzo Church's Lambda Calculus, and was originally thought for AI.

15.1.1 MapReduce – Why not?

MapReduce has limits. It doesn't work well when:

- We don't know the number of steps in advance

- We don’t know where computation will run
- The environment is heterogeneous (different kinds of devices/machines)

Still, MapReduce can be seen as a special case of a more general model called **Bulk Synchronous Parallel (BSP)**.

15.2 BSP – Bulk Synchronous Parallel Model

The BSP model is a framework for writing parallel algorithms. It has three global steps:

1. Computation happens in parallel
2. Communication between nodes
3. Barrier synchronization (everyone waits before moving to the next step)

The problem is that full synchronization is risky: if one machine fails or gets partitioned from the network, the whole system may freeze. This is often a bigger issue than latency or bandwidth.

15.3 Delegation

To achieve true scalability, we need models that are **decentralized by design**.

Examples of **Actor-based frameworks**:

- **Akka** – Java/Scala (not open source anymore)
- **Orleans** – .NET
- **CAF** – C++
- **Ray** – Python
- **ZIO Actors** – Scala (for single-machine computations) (*Fun fact: the Halo 4 game backend used this model.*)

Another approach is **Reactive Programming**, which uses asynchronous data streams to handle events and flows. (*Example: Netflix API is based on this idea.*)

In practice, actors are not run as separate threads, because a single machine might need to manage **millions of actors**. Instead:

- Actors can be placed in lists, and a single thread calls them one by one (round-robin).

- Or multiple threads can handle different actor lists.
- Many possible implementations exist.

15.4 The Actor Model – Key Concepts

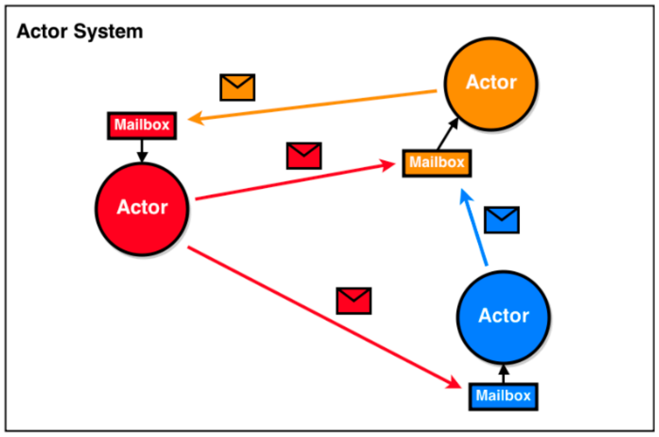


Figure 15.1: Actor model schema

The **actor model** is a simple and clear way of programming for highly concurrent and distributed systems. It focuses on **message passing** and **actor isolation**, which makes systems more modular (easier to break into parts) and scalable (able to grow).

Definition 15.1 (Actor) An actor is a computational unit that, when it receives a message, can at the same time:

- Send a limited number of messages to other actors
- Create a limited number of new actors
- Decide what behavior it will use when it gets the next message

Actors are **stateful** (not stateless) — they remember progress in their own computation.

Modularity in the actor model comes from one-way, asynchronous communication. Once a message is sent, it's the receiver's job to handle it.

Messages are separate from the sender and are delivered by the system on a **best-effort basis** (not guaranteed order or timing). Also, since actors can create new actors, they form a **hierarchy**. This can help with **locality of computation** (keeping related work close together).

15.4.1 Indeterminacy and Quasi-Commutativity

Messages are delivered asynchronously, and their **order is not guaranteed**. Because of this, the actor model is **not fully deterministic** (results may vary).

Quasi-commutativity means that in some cases, the order of messages between two actors **does not change the final result**.

This flexibility is important so that the system is not forced to follow a strict message order, which could be slowed down or broken by network delays, errors, or lost messages.

15.4.2 Fault Tolerance

The actor model is naturally **fault-tolerant**. If one actor fails, the rest of the system can still keep working. Failed actors can be restarted or moved to another machine without stopping the whole system.

17. TLAV

MapReduce with HDFS is a common solution for **Big Data** problems because it lets us process huge amounts of data across many machines. But it also has **limits**: it has **high latency** (slow response) and does **not support real-time processing**.

HDFS (Hadoop Distributed File System) is a storage system that spreads large datasets across many machines, so data is stored and processed in parallel.

18.1 Spark SQL

Spark SQL lets you run **SQL queries** on big data. It is a Spark module for structured data processing. These queries are automatically converted into Spark jobs (made of actions and transformations).

One use of SparkSQL is to execute SQL queries. When running SQL the results will be returned as a Dataset/DataFrame.

Dataset is a distributed collection of data. Data set is a new interface that provides the benefits of RDDs with the benefits of SparkSQL's optimized execution engine.

- It is built on top of **Spark Core**.
- It gives a more **user-friendly interface** for working with data.

- The output is a **DataFrame** (a distributed table with named columns).

With **SparkSession**, applications can create **DataFrames** form existing RDD, from Hive table, or from Spark data sources.

The SQL function of a SparkSession enables application to run SQL queries programmatically and return the result as DataFrame.

18.1.1 Structured Streaming

The main idea of structured streaming is to treat a **live data stream** as if it were a **table** that keeps getting new rows.

- This way, we can run SQL queries on the stream.
- The system updates results **in real-time** as soon as new data arrives.

It is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine.

The system ensures end-to-end exactly-once fault-tolerance guarantees through checkpointing and Write-Ahead Logs.

Output modes:

Complete Mode - The entire updated Result Table will be written to external storage. It is up to the storage connector to decide how to handle writing of the entire table.

Append Mode - Only the new rows appended in the Result Table. since the last trigger will be written to the external storage. This is applicable only on the queries where existing rows in the Result Table are not expected to change.

Update Mode - Only the rows that were updated in the Result Table sine the last trigger will be written to the external storage.

- Note that this is different from the complete mode that this mode is only outputs the rows that have changed since the last trigger.
- If the query doesn't contain aggregations, it will be equivalent to Append mode.

Vertex Centric Computing

19.1 Introduction to vertex-centric computing

Vertex-centric computing combines ideas from **distributed computing**, **graph theory**, and **big data**.

19.1.1 Graph Theory

A graph is usually written as **G = (V, E)**, where:

- **V** = set of vertices (nodes),
- **E** = set of edges (connections).

Edges can be **directed** (one-way) or **undirected** (two-way), and they may also have **weights** (values).

Each vertex has an **in-degree** (edges coming in) and an **out-degree** (edges going out).

Edges represent relationships between vertices.

Definition (Graph Theory): Graph theory is a branch of math that studies graphs, which are structures that model relationships between objects. It focuses on **connectivity, flows, topology**, and more.

Types of graph topologies:

- **Random graphs** – each edge exists with some probability *p*.
- **Small-world graphs** – most nodes can reach each other with only a few hops.
- **Preferential attachment** – new nodes prefer to connect to nodes that already have many connections (like social networks).

19.1.2 Towards Vertex-Centric

Definition (Vertex-Centric): Vertex-centric computing is a graph processing model where the programmer defines a function, and this function is run **on each vertex of the graph**.

This model (often called **Think-Like-A-Vertex, TLAV**) is simple and flexible. It can be used in:

- **Decentralized networks** (one machine = one node),
- **Distributed networks** (one machine = many nodes),
- **Multi-core machines** (one core = many nodes).

📜 Report:

🎚️ Slides

Data Processing

Data can be processed in **batch** or in **real-time (stream)**, or with a **mix of both**.

- **Batch processing**: when data is collected first and then processed in one go.
- **Real-time processing**: when data must be processed as soon as it is created.

17.1 Batch Processing

Batch processing **does not need immediate results**. Data is stored temporarily and then processed in groups.

- Usually **cheaper and easier** than real-time.
- Works well for tasks like **data transformation, reporting, analytics**.
- Has **high latency** (slow results) but **high throughput** (can handle a lot at once).
- Easier to design and maintain.
- Not good for real-time decision-making.

The main model here is **MapReduce**.

17.2 Stream Processing

Stream processing focuses on **handling data as soon as it arrives**.

17.2.1 Time semantics

- **Event time –** when the event actually happened.
- **Processing time** – when the system processes it.
- **Ingestion time** – when the system first receives it.

To manage timing, we use **watermarks**, which help detect **late-arriving data**.

Challenges: events may **arrive late** or **out of order**. Solutions:

- **Buffering** (wait for more data)
- **Allow lateness** in processing

Events are grouped with **windowing**:

- **Time-based windows**: fixed or sliding time periods.
- **Count-based windows**: groups based on a number of events.
- **Session-based windows**: groups based on activity and inactivity periods.

17.2.2 Lambda Architecture

Combines **batch + real-time**. Has 3 layers:

- **Batch layer –** processes large volumes of historical data, scalable and fault-tolerant.
- **Speed layer** – processes new data quickly (low latency).
- **Serving layer –** answers user queries by combining results from batch + speed layers.

Pros: unified view of old + new data, scalable.

Cons: complex, higher cost, may duplicate work.

17.2.3 Kappa Architecture

Created to simplify Lambda by **removing the batch layer**. Focuses only on **stream processing**.

- Handles both historical and real-time data through **streams only**.
- Uses **checkpointing and state backends** to manage progress.
- Very good for **IoT devices and analytics**.

- Still more complex than simple streaming, but **faster and more scalable** than Lambda.

17.2.4 CQRS (Command Query Responsibility Segregation)

Used when there are **many writes but few reads**.

Separates the system into two models:

- **Command model (write side)** – handles updates, inserts, deletes.
- **Query model (read side)** – handles data retrieval.

Pros: better scalability and performance.

Cons: can be hard to implement, risk of inconsistencies between read/write models, adds latency.

17.3 Data Processing Frameworks

17.3.1 Sharing or not

- **Shared-nothing architectures**: each node has its own memory and storage, communicates by message passing.
  - Good for **big data** and **web apps**, supports horizontal scaling.
  - Examples: **Hadoop, DynamoDB, Couchbase, Facebook TAO**.

17.3.2 Data Lakehouse

A modern architecture that **mixes data lakes + data warehouses**.

- Stores **raw data** (like data lakes).
- Processes data (like warehouses).
- Works with **structured, semi-structured, and unstructured data**.
- Supports both **batch and real-time processing**.
- Example: **Snowflake**.

17.3.3 Federated Learning

Here the **training of machine learning models** happens directly on edge devices (e.g., IoT), not on a central server.

- Only the **updates** are sent to a server for aggregation.

- Improves **privacy and security**, since raw data stays on devices.
- Good for **IoT environments**.

17.3.4 Serverless Data Processing

Here the infrastructure is **fully abstracted**. You don't manage servers, only the data pipelines.

- Great for scaling.
- Widely used in **modern data workflows**.

Self-Stabilization (DIDNT HAVE THIS LECTURE)

6.1 Self-stabilization

In a distributed system, many machines are spread out and often communicate. Because of this, the system can sometimes end up in a wrong or *illegitimate* state (for example, if a message is lost).

Think about token-based systems: what if the token is lost? That would cause problems.

Note that what is considered *illegitimate* or *legitimate* depends on the application.

Definition 6.1 (Self-stabilization):

No matter what the starting state is, the system will always reach a *legitimate* state in a finite number of steps, and it does this by itself without outside help.

6.1.1 Challenges

The main difficulty is that nodes in a distributed system do not share a single memory they can look at. Each node only knows its own state and what it receives from neighbors, but together their actions must still reach the global goal (a correct state for the whole system).

6.2 System Model

A distributed system (DS) can be described as:

- A set of n machines, called processors, that talk to each other.
- The ith processor is called Pi.

- A processor's neighbors are the processors directly connected to it.
- Neighbors communicate by sending and receiving messages.
- The DS can be represented as a graph:
    - processors are nodes,
    - links between neighbors are edges.
- Communication channels are modeled as FIFO queues: Qij is the queue of messages sent from Pi to Pj but not yet received.
- Each processor has its own state.
- The full description of the DS at any time is given by all processor states plus the content of all queues.

This model is simplified. In reality, messages are rarely delivered in exact FIFO order, since networks are unreliable and unpredictable.

6.2.1 System Configuration

The term *system configuration* is used to describe the DS at a certain moment.

It is written as:

c = s1, s2, ..., sn, q1,2, q1,3, ..., qn,n−1

where si is the state of processor Pi, and qi,j (i ≠ j) is the content of the queue from Pi to Pj.

6.2.2 Network Assumptions

- N is the maximum possible number of processors in the system (useful for dynamic systems).
- γ is the network diameter, meaning the maximum number of links in the shortest path between any two processors.
- A network is **static** if its communication structure (topology) never changes.
- It is **dynamic** if links or nodes can fail (go down) and later recover.
- In dynamic systems, **self-stabilization** is considered starting from the time after the *last* (or "final") failure of a link or node. The idea of "final failure" is often used in the self-stabilization literature.

- Because stabilization only happens eventually, we assume that faults will eventually stop. This means there must be a “long enough period” without new failures, so the system has time to stabilize.
- Finally, it is always assumed that the network stays **connected**—that is, there is still some path between every pair of nodes.

Self-stabilization (formal definition)

We define self-stabilization for a system **S** with respect to a condition (predicate) **P** over its set of global states.

- **P** represents the correct behavior of the system.
- States that satisfy **P** are called **legitimate (safe)** states.
- States that do not satisfy **P** are **illegitimate (unsafe)** states.

A system **S** is self-stabilizing with respect to **P** if it satisfies two properties:

1. **Closure** – Once the system is in a legitimate state, it will remain legitimate in all future executions.
2. **Convergence** – From any starting (possibly illegitimate) state, the system will eventually reach a legitimate state in a finite number of steps.

Issues in designing self-stabilizing systems

Some of the main challenges are:

- How many states each node (machine) must have.
- Uniform vs non-uniform algorithms in distributed systems.
- Centralized vs distributed uniformity.
- Reducing the number of states in token ring systems.
- Shared memory and mutual exclusion.
- Costs of self-stabilization.

Dijkstra’s Token Ring Algorithm

- The system has **n finite-state machines** connected in a ring.
- Each machine has a **privilege** (the right to change its state).
- A privilege is defined using its current state and the state of its neighbors.
- When a machine has a privilege, it can perform a **move** (state change).
- If multiple machines have privilege at once, a **central demon** (scheduler) decides which one moves first.

A **legitimate state** must follow these rules:

- There is at least one privilege in the system (no deadlock → liveness).
- Every move from a legal state produces another legal state (closure).
- Each machine eventually gets a privilege (no starvation).
- Any legal state can reach another legal state through a sequence of moves (reachability).

👉 Dijkstra defined a legal state as one where **exactly one machine has the privilege**.

This is like **mutual exclusion**, since only one process can be in its critical section at a time. After finishing, the privilege is passed to a neighbor.

First Solution

- Machine 0 is **special** (different).
- All other machines are identical.

Second Solution

- Improved version: uses only **three states (0,1,2)**.
- Here, two machines are special:
  - **Bottom machine (state 0)**
  - **Top machine (state n−1)**

The rules:

- **Bottom machine**: if `(S+1) mod 3 == R`, then `S := (S-1) mod 3`

- **Top machine**: if `L == R` and `(L+1) mod 3 != S`, then `S := (L+1) mod 3`
- **Other machines**:
  - if `(S+1) mod 3 == L`, then `S := L`
  - if `(S+1) mod 3 == R`, then `S := R`

This ensures the system always has at least one privilege. Over time, the number of privileged machines reduces to **one**, so the system self-stabilizes.

Observations

- The number of states per machine matters.
- Dijkstra showed three solutions for a ring of **n machines**, each with:
  - **K ≥ n states**
  - **K = 4 states**
  - **K = 3 states**

Uniformity

- In distributed systems, it is nice if all machines run the **same algorithm**.
- But for self-stabilization, sometimes different machines must run **different algorithms**.
- Uniformity can make it hard to decide which machine should move, leading to deadlocks.
- Therefore, a **central demon** is often assumed.

Costs of Self-Stabilization

- Goal: reduce **convergence span** (time to reach a legal state) and **response span** (time to respond).
- Time complexity is measured in **rounds**.

Designing Self-Stabilizing Systems

- Think of a **malicious adversary** that can disrupt the system.

- A self-stabilizing system can **recover** from disruption and return to a legal state.
- A common method is **layering** (like in internet protocols).

Because of the **transitivity property** of self-stabilization:

- If **P stabilizes Q** and **Q stabilizes R**, then **P stabilizes R**.