Scalable Distributed Computing - Appunti

Francesco Lorenzoni

September 2024

Contents

Part I

Introduction to Scalable and Distributed Computing

Chapter 1

Basic Concepts

1.1 Introduction

Definition 1.1 (Distributed) *Spreading tasks and resources across multiple machines or locations*

Distribution enhances resilience and efficiency through decentralization.

Example 1.1.1 *Google's global data centers ensuring users across the world get fast search results.*

Definition 1.2 (Scalable) *Ability to grow and handle increasing workload without compressing performance*

Scaling is about growth and expansion while maintaining efficiency

Example 1.1.2 *Netflix scaling its services to accommodate millions of users streaming content simultaneously*

1.2 Scalability and Motivations

Scalability refers to a system's ability to handle increased load by adding resources (e.g., servers, nodes, or storage). Typically we have to two types of scalability:

- ◇ **Vertical** - Adding resources to a single node
- ◇ **Horizontal** - Adding more nodes to a system

In relation to scalability, systems can hence be characterized by **performance**, intended as the ability to handle a growing number of requests, and **elasticity**, intended as the ability to scale up or down based on current needs. On this matter, scalability may come in two flavours:

- ◇ **Strong Scalability** - The system's performance increases linearly with the number of resources.
  This is ideal for systems where the workload can be evenly distributed across many nodes
- ◇ **Weak Scalability** - The system's performance does not degrade as the number of users increases
  This fits best systems where the total workload grows alongside the system's resources

1.2.1 But why? - Motivation for Scalable Distributed Systems

- ◇ **Growing data** - large datasets from applications like social media, IoT, AI, etc.
- ◇ **Global Users** - Billions of users worldwide
- ◇ **Performance** - Reducing latency, increasing throughput, and improving reliability
  Sometimes latency is *not* a priority: in some systems it is okay to have high latency to guarantee high throughput and reliability

The challenges are mostly to **manage resources** across geographically distributed systems, and ensuring **low latency** and **high availability**.

1.2.2 Target Architectures

Some architectures which require scalable distributed systems are IoT networks, High-Performance Computing (HPC), and Cloud/Edge Computing.

Example - Cameras in a district

What if I send all the data gathered from cameras to a *single cloud*?

Pros
- ◇ Unlimited storage and processing power
- ◇ Centralized management
- ◇ Simplicity

Cons
- ◇ High latency
- ◇ High bandwidth usage
- ◇ Single point of failure (Scalability and Reliability)

But also simpler applications may considerably benefit from scalable and distributed architectures.
- ◇ Large graph analysis
- ◇ Stream processing
- ◇ Streaming services
- ◇ Machine Learning
- ◇ Big Data
- ◇ Computational Fluid Dynamics
- ◇ Web and online services

1.2.2.1 Parallel computing

Can't we rely on parallel computing to solve these problems?
Not really, parallel fits different needs and works in a slightly different way.

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

Figure 1.1: From Parallel to Distributed

1.2.3 Distribution is cool, but...

Consider that local computation is always faster than remote computation. (*Waaay faster*)
From the CPU perspective, time passes *very slowly* when the data travels outside the machine.

If one CPU cycle happened every second, sending a packet in a data center would take 20 hours. Sending it from NY to San Francisco would take 7 years.

1.3 Assessment Method - Exam

There are three options, but note that **in every case an oral exam will follow.**

1. *Writing a Survey or a Report* students can conduct a comprehensive survey or prepare an in-depth report on a topic or technology related to the course.
2. *Individual or Group Project* ($\leq$ 3 members) Designing, implementing and presenting a solution or prototype related to scalable distributed computing.
3. *Traditional Written Exam* "Questions, answers...you know the drill." Very sad option, in my opinion, but prof. Dazzi did not completely discourage it.

Prof. Dazzi is very open to proposals for the exam, he'd like to stimulate our creativity and curiosity.

Prof. Dazzi says that usually its oral examinations last from 30 to 35 minutes, even though there may be exceptions. Clearly, if the student chooses the report or the project, part of the oral will be about the proposed work, but also questions about the course will be asked.

Chapter 2

Elements and Challenges in Distributed Systems

"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

Various definitions of Distributed Systems have been given in the literature, none of them **satisfactory**, and none of them in agreement with any of the others.

Let's try to accept a quite simple one:

" A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system. " — Andrew S. Tanenbaum

From this definition, we can derive two key points:

1. **Autonomy** - Each node in the system is independent and can make decisions on its own
2. **Transparency** - The system appears as a single entity to the user

2.1 Autonomous Computing Elements

Nodes can act independently from each other, so:

- ◇ nodes need to achieve common goals realized by **exchanging messages** with each other
- ◇ nodes **react to messages** leading to further communication through message passing

Having a collection of nodes implies also that each node should know which other nodes are in the system and should contact, and how to reach them. In particular it is important to have a **robust naming system** to identify nodes, and to allow scalability. The key point of a naming system and a DNS is to **decouple** the name of a node from its physical address.

Since each node has its own notion of time, it is not always possible to assume that there is a "global clock": there must be **synchronization** and **coordination** mechanisms to ensure that the system behaves correctly.

2.2 Transparency

The distributed system should appear as a single coherent system, so end users should not even notice that they are dealing with the fact that processes, data, and control are dispersed across a computer network.

This so-called **distribution transparency** is an important design goal of distributed systems.

A —perhaps not-so-perfect— example may be Unix-like operating systems in which resources are accessed through a unifying file-system interface Effectively hiding the differences between files, storage devices, and main memory, but also networks

However, striving for a single coherent system introduces relevant issues: e.g., **partial failures** are inherent to any complex system, in distributed systems they are particularly difficult to hide.

2.2.1 Middleware

To aid the need of easing the development of distributed applications, distributed systems are often organized to have a **separate layer** of software placed on top of the respective operating systems of the computers that are part of the

system.
This layer is called **middleware**. [Bernstein, 1996]

In a sense, middleware is the same to a distributed system as what an operating system is to a computer: a manager of resources offering its applications to efficiently share and deploy those resources across a network

We list below some example of possible middleware services.

2.2.1.1 RPC - Communication

Remote Procedure Call (RPC) allows an application to invoke a function that is implemented and executed on a remote computer as if it was locally available.

Developers need merely specify the function header and the RPC subsystem generates the necessary code that establishes remote invocations.

Notable examples: Sun RPC, Java RMI, Google RPC (gRPC)

2.2.1.2 Service Composition

Enabling Service composition allows to develop new applications by taking existing programs and gluing them together, e.g. Web services.

An example: Web pages that combine and aggregate data from different sources, e.g. GMaps in which maps are enhanced with extra information from other services.

2.2.1.3 Improving Reliability

Horus toolkit allows to build applications as a group of processes such that any message sent by one process is guaranteed to be received by all or no other process.

As it turns out, such guarantees can greatly simplify developing distributed applications and are typically implemented as part of the middleware.

A more down-to-earth example of improving reliability is the use of RAID systems to ensure that data is not lost in case of disk failure.

Figure 2.1: Horus Toolkit

2.2.1.4 Supporting Transactions on Services

Many applications make use of multiple services that are distributed among several computers.

Middleware generally offers special support for executing such services in an all-or-nothing fashion, commonly referred to as an **atomic transaction**.

Recall all the mess that can happen with threads, race conditions, and deadlocks? Middleware can help with that.

The application developer need only specify the remote services involved.

By following a standardized protocol, the middleware makes sure that every service is invoked, or none at all.

2.3 Building Distributed...is it a good idea?

Building distributed systems is a challenging task, and it is not always the best choice. There are **four important goals** that should be met to make building a Distributed System worth the effort:

1. ***Resource*** *accessibility*
2. ***Hide*** *Distribution*
3. *Be* ***open***
4. *Be* ***scalable***

2.3.1 Resource Accessibility

In a distributed system the access, and share, of remote resources is of paramount importance. **Resources** can be virtually anything (peripherals, storage facilities, data, ...).

Connecting users and resources makes easier to collaborate and exchange information (hint: look at the success of the Internet).

File-sharing used for distributing large amounts of data, software updates, and data synchronization across multiple servers.

2.3.2 Hide Distribution

Hiding distribution is a fundamental goal in the design of distributed systems, and is related to an already mentioned key point, *distribution transparency*, but does not limit to it.
It means to hide that processes and resources are physically distributed across multiple computers possibly separated by large distances.
More precisely, it means to enforce the following properties in **??**:

`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 have different ways to represent and access data. A well designed distributed system needs to hide differences in:

- ◇ physical machine architectures
- ◇ data representation by different operating systems

A distributed system may have computer systems that run different operating systems, each having their own file-naming conventions.

Differences in naming conventions, file operations, or in low-level communication mechanisms with other processes, etc

2.3.2.2 Location and Relocation Transparency

Location transparency means that users can access resources even are not aware where an object is physically located in the system.

Naming plays an important role by assigning logical names to resources, i.e. names not providing information about physical location.

Basically, naming is an indirection process!

Shardcake is a Scala library that provides location transparency for distributed systems, by exploiting "Entity Sharding".

An example of a such a name is the uniform resource locator (URL) http://www.unipi.it, which gives no information about the actual location of University's main Web server.

The entire site may have been moved from one data center to another, yet users should not notice, that is an example of relocation transparency, which is becoming increasingly important in the context of cloud computing.

Actually this is not the case for UniPi hihi ☺.

2.3.2.3 Migration Transparency

Relocation transparency refers just to being moved across the distributed system, migration transparency is:

- ◇ offered by a system when it supports the mobility of processes and resources initiated by users
- ◇ does not affect ongoing communication and operations

A common example is communication between *mobile phones*: regardless whether two people are actually moving, mobile phones will allow them to continue their conversation teleconferencing using devices that are equipped with mobile Internet.

2.3.2.4 Replication Transparency

Resources may be replicated to increase availability or to improve performance by *placing a copy close* to the place where it is accessed.

Hide the existence of copies of a resource or that processes are operating in some form of lockstep mode so that one can take over when another fails.

To hide replication to users, it is necessary that all replicas have the same name.

Systems that support replication transparency should support location transparency as well.

2.3.2.5 Concurrency Transparency

Two independent users may each have stored their files on the same file server or may be accessing the same tables in a shared database.

It is important that each user does not notice that the other is making use of the same resource, this phenomenon is called concurrency transparency

- ◇ concurrent access to a shared resource leaves that resource in a **consistent** state
- ◇ consistency achieved through locking mechanisms to give users **exclusive access** to a resource

A more refined mechanism is based on **transactions**, but may strongly impact on scalability.

2.3.2.6 Failure Transparency

Failure transparency is the ability of a system to mask the failures of components from users, so a user —or an application— does not notice that some piece of the system failed and that the system subsequently (and automatically) **recovers** from that failure.

Masking failures is one of the hardest issues in distributed systems and is even impossible when certain assumptions are made.

The main difficulty in masking and transparently recovering from failures is in the inability to distinguish between a dead process and a slowly responding one.

"Is the server is actually down or is the network too badly congested ?" It is often not easy to tell the difference.

2.3.3 Degree of distribution transparency

Distribution transparency is generally considered preferable for any distributed system, however there is a **trade-off** between a *high degree of transparency* and the *performance* of a system.

There are situations in which attempting to blindly hide all distribution aspects from users is *not* a good idea.

examples

- ◇ Internet applications repeatedly try to contact a server before finally giving up, as attempting to mask a transient server failure before trying another one may slow down the system as a whole.
- ◇ Guarantee that replicas, located on different continents, must be consistent all the time, may be costly
  a single update operation may take seconds to complete, that cannot be hidden from users

In other situations it is *not at all obvious* that hiding distribution is "not" a good idea.

- ◇ devices that people carry around and where the very notion of location and context awareness is becoming increasingly important e.g., finding the nearest restaurant
- ◇ when working real-time on shared documents concurrency transparency could hinder the cooperation

There are also other arguments against distribution transparency. Recognizing that full distribution transparency is *impossible*, we should ask ourselves whether it is wise to pretend to achieve it.

In some cases, it may be better to make distribution **explicit** so that: the user and application developer are never tricked into believing that there is such a thing as transparency, resulting in users much better understanding the behavior of a distributed system, and thus prepared to deal with its behavior.

2.4 Openness

An open distributed system is essentially a system that offers components that can easily be used-by, or integrated, into other systems. An open distributed system itself will often consist of components that originate from elsewhere.

Being open enables two key features:

- ◇ Interoperability, composability, and extensibility
- ◇ Separation of policies from mechanisms

Open means that components adhere to standard rules describing the syntax and semantics of those components usually by relying on an Interface Definition Language (IDL). An interface definition allows an arbitrary process that needs a certain interface, to interact with another process that provides that interface. This allows two independent parties to build completely different implementations of those interfaces.

Proper specifications are **complete** and **neutral**. *Complete* means that everything that is necessary to make an implementation has indeed been specified. However…many interface definitions are not at all complete so that it is necessary for a developer to add implementation-specific details. This is just as important is the fact that specifications **do not prescribe what an implementation should look like**, they should be *neutral*.

As pointed out in Blair and Stefani, completeness and neutrality are important for **interoperability** and **portability**.

- ◇ **Interoperability** - characterizes the extent by which two implementations of systems from different manufacturers can work together
  - by relying on each other's services
  - as specified by a common standard
- ◇ **Portability** - characterizes to what extent an application developed for a given distributed system can executed
  - without modification
  - on a different distributed system implementing the same interfaces

Another important goal for an open distributed system is that it should be easy to **configure** the system out of different components (possibly from different developers). Also, it should be easy to **add** new components or replace existing ones without affecting those components that stay in place.
In other words, an open distributed system should also be **extensible**.

For example, in an extensible system, it should be relatively easy to add parts that run on a different operating system, or even to replace an entire file system

2.4.1 Policy and Mechanism Separation

To achieve flexibility in open distributed systems, it is crucial that the system be organized as a collection of relatively small and easily replaceable or adaptable components.

A fundamental principle for achieving this flexibility is the **separation of policy and mechanism**:

- ◇ **Mechanism** defines *how* something is done - the low-level implementation that provides capabilities and operations
- ◇ **Policy** defines *what* and *when* to do - the high-level decision-making rules and strategies

By separating these concerns, the system becomes more flexible: policies can be changed or adapted without modifying the underlying mechanisms, and different policies can use the same mechanisms.

Examples in distributed systems:

- ◇ *Load balancing*: The mechanism provides request routing and node communication, while the policy decides which server receives each request (e.g., round-robin, least-loaded, geographic proximity)
- ◇ *Caching*: The mechanism implements the cache data structure and access operations, while the policy decides which items to evict (e.g., LRU, LFU, FIFO)
- ◇ *Resource allocation*: The mechanism manages resource availability and assignment, while the policy determines priorities and quotas for different users or applications
- ◇ *Security*: The mechanism implements authentication and access control infrastructure, while the policy defines who can access what resources and under which conditions

This implies providing definitions not only for the highest-level interfaces (those seen by users and applications), but also for interfaces to internal parts of the system, describing how those parts interact. This approach is an alternative to the classical monolithic approach in which components are implemented as one huge program, making it hard to replace or adapt a component without affecting the entire system.

2.5 Scalability

We were used to having relatively powerful desktop computers for office applications and storage.

We are now witnessing that such applications and services are being placed "in the cloud",leading in turn to an increase of much smaller networked devices such as tablet computers.

With this in mind, scalability has become one of the most important design goals for developers of distributed systems.

2.5.1 Dimensions of Scalability

Scalability is a complex issue and can be seen from different perspectives, such as:

- ◇ **Size** - A system can be scalable with respect to its size i.e., we can add more users and resources to the system without any noticeable loss of performance.
- ◇ **Geographical** - A geographically scalable system is one in which the users and resources may be distant, but communication delays are hardly noticed.
- ◇ **Administrative** - An administratively scalable system is one that can still be easily managed even if it spans many independent administrative organisations

2.5.2 Size scalability

Many **users** need to be supported → limitations of centralized services.

Many services are **centralized** → implemented by a single server running on a specific machine or in a group of collaborating servers co-located on a cluster in the same location.

The problem with this scheme is obvious: the server, or group of servers, can become a **bottleneck** due to three root causes:

- ◇ The computational capacity, limited by the CPUs [CPU bound]
- ◇ The storage capacity, including the I/O transfer rate [I/O bound]
- ◇ The network between the user and the centralized service [Network bound]

2.5.3 Geographical scalability

TLDR: Solutions developed for local-area networks cannot always be easily ported to a wide-area system.

This kind of scalability relates on the difficulties in scaling existing distributed systems that are **designed for local-area networks**, many of them even based on synchronous communication e.g., a party requesting service blocks until a reply is sent back from the server implementing the service.

Communication patterns are often consisting of many client-server interactions as may be the case with database transactions; this approach generally works fine in LANs where communication between two machines is often just a few hundred microseconds, but does not scale to WANs where communication may take hundreds of milliseconds. Besides in WANs, the probability of packet loss is much higher than in LANs, and the bandwidth is much lower.

2.5.4 Administrative scalability

This addresses how to scale a distributed system across multiple, independent administrative domains.

A major problem that needs to be solved is that of conflicting policies with respect to resource usage (and payment), management, and security

2.6 How to scale?

Scalability problems in distributed systems appear as performance problems caused by limited capacity of servers and network. Improving their capacity (e.g., by increasing memory, upgrading CPUs, or replacing network modules) is referred to as **scaling up**, while **scaling out** refers to deploying more servers.

There are three main strategies to *scale out*, they are discussed in the following sections.

2.6.1 Hiding Communication Latencies

Applies in the case of geographical scalability, and aims at avoiding waiting for responses to remote-service requests. Instead waiting for a reply, do other useful work at the requester side, and when the reply arrives invoke a special *handler*; in other words, implement **asynchronous communication**.
This is very much used in batch-processing systems and parallel applications, contexts where indenpendent tasks can be scheduled for execution while another task is waiting for communication to complete.

In some scenarios asynchronous communication does not fit; a solution is to move part of the computation from server to client. This motivates *"hierarchical approaches"*. This is the foundation of **edge-computing**.

2.6.2 Distributing Work

Another scaling technique is partitioning and distribution: taking a component, splitting it into smaller parts, and subsequently spreading those parts across the system.

A very simple example is the World Wide Web: to most users the web appears to be an enormous document-based information system, but in reality it is a distributed system in which the documents are stored on many different servers.

2.6.3 Replication - Caching

Scalability problems often appear in the form of **performance degradation**, thus it is generally a good idea to actually replicate components across a distributed system. Replication increases **availability** and also helps to **balance the load** between components, leading to better performance.

Also, in geographically widely dispersed systems, having a copy nearby can hide much of the *communication latency* problems mentioned before.

2.6.3.1 Caching

Caching results in making a copy of a resource, generally in the proximity of the client accessing that resource. In contrast to replication, caching is a decision made by the *client* of a resource and **not** by the *owner* of a resource.

The most serious drawback to caching and replication in general is handling the **inconsistency** that may arise when a copy of a resource is updated. Incostincency occurs always, and to what extent it can be tolerated depends highly on the usage of a resource.

Seeing a cached web page, old of a few minutes, is acceptable. Old Stock-exchanges are not.

Non-scalability

Strong-consistency is difficult to enforce. If two updates happen *concurrently*, it is required that updates are processed in the same **order** everywhere, introducing an additional global ordering problem. Besides, combining strong consistency with high availability is, in general, impossible.

Global synchronization mechanisms are typically not scalable. So…

Scaling by replication may introduce inherently non-scalable solutions.

This is a fundamental paradox: replication is used to *improve* scalability, but maintaining strong consistency across replicas requires global coordination, which is *inherently non-scalable*. The more replicas you add, the more coordination overhead is needed:

- ◇ Every update must be propagated to all replicas and confirmed before completion
- ◇ With N replicas, coordination involves N nodes, increasing communication overhead
- ◇ Latency is determined by the slowest replica (geographical distance matters)
- ◇ The coordination mechanism itself becomes a bottleneck

Example: A database replicated across 3 continents to improve availability. With strong consistency, each write must wait for confirmation from all data centers (200-300ms inter-continental latency $\times$ 3), making writes very slow. Adding a 4th replica makes it worse.

This leads to the **CAP theorem**: you cannot simultaneously have strong *Consistency*, high *Availability*, and *Partition tolerance*. Most modern distributed systems choose eventual consistency instead of strong consistency to maintain scalability and availability.

2.6.4 Pitfalls

Developing a scalable and distributed system is a formidable task. Resources **dispersion** must be taken into account at design time, otherwise the system will be complex and flawed.

Whenever someone approaches designing a distributed system for the first time, it is easy to make mistakes. In fact, Peter Deutsch, in his famous fallacies of distributed computing, lists the following fallacies that are often made by developers of distributed systems:

- ◇ Network is reliable, secure and homogeneous
- ◇ The topology does not change
- ◇ Latency is zero, bandwidth is infinite
- ◇ Transport cost is zero
- ◇ There is one administrator

This happens because this issues, when developing non-distributed applications, most likely do not show up.

Some latency-sensitive applications are:

- ◇ Online gaming
- ◇ Video conferencing
- ◇ Remote surgery
- ◇ …

2.7 Types of distributed systems

2.7.1 HP(D)C - Clusters

Collection of similare workstations closely connected by means of a high-speed network, used to solve large-scale problems. Nodes typically run the same operating system and are somehow **homogeneous**.

Clusters are used in the most important supercomputers in the world.

Clusters are always *preferrable* to a single super-powerful machine, but not only because they are cheaper and reliable, but because they easily allow for **horizontal scalability**.

Beowulf clusters

Linux-based Beowulf clusters are cheap and easy to build, and are used in many scientific applications.

2.7.2 HP(D)C - Grid

Still HPC, but here the nodes belong to different administrative domains, and may be geographically distributed. Grid computing consists of distributed systems that are often constructed as a **federation** of computer systems. Clearly, the nodes in a grid are **heterogeneous** and may run different operating systems.

Heterogeneity actually may be advanteageous, since different workloads may be better suited to different types of machines.

Globus is an architecture initially proposed by Foster and Kesselman, and is one of the most widely used grid middleware systems.

2.7.3 Cloud

Cloud computing is a model outsourcing the entire infrastructure. The key point is the providing the facilities to dynamically construct an infrastructure and compose what is needed from available resources.
This is not really about being HPC, but about being able to scale up and down as needed, and in general providing lots of resources.

The father of *cloud* was the the concept of **utility computing**[1]., by which a customer could upload tasks to a data center and be charged on a per-resource basis.

Types

- ◇ **IaaS** - Infrastructure as a Service
- ◇ **PaaS** - Platform as a Service
- ◇ **SaaS** - Software as a Service
- ◇ **FaaS** - Function as a Service
- ◇ *Many-other-stuff as a service*, such as Backend aaS, Database aaS, etc…

Cloud computing is very popular, but there a few issues concerning it:

- ◇ Lock-in - Once you start using a cloud provider, it is hard to switch to another one
  Prof. Dazzi says that in general, it is cheaper to push data in the cloud, but more costful to pull it.
- ◇ Security and privacy issues - You are giving your data to someone else
- ◇ Dependence on the network - If the network resources —or simply the network— go down, you are in trouble

2.8 Pervasive systems

The distributed systems discussed so far are largely characterized by their stability: nodes are fixed and have a more or less permanent and high-quality connection to a network. To a certain extent, this stability is realized through the various techniques for achieving distribution transparency

[1] *"utilities"* in english are water, gas, electricity, etc…

However, matters have changed since the introduction of mobile and embedded computing devices, leading to what are generally referred to as **pervasive systems**.
The separation between users and system components is much more *blurred*. Typically there is no single dedicated interface, such as a screen/keyboard combination, and in the system there may be many **sensors** picking up various aspects of a user's behavior.

Many devices in pervasive systems are characterized by being **small**, battery-powered, mobile, and a wireless connection.

These are not necessarily *restrictive* aspects, consider Smartphones, for instance.

We may distinguish three types of pervasive systems, which may overlap:

1. Ubiquitous computing systems
2. Mobile systems
3. Sensor networks

Chapter 3

Time

These aspects have been already mentioned, but let's recall some Time-related issues in distributed systems:

- ◇ No global clock, every node has its own
- ◇ Asynchronous best-effort communication, messages may be lost
- ◇ No central authority, but coordination is needed

Time is fundamental for two reasons:

1. **Ordering**: to order events, we need to know when they happened
2. **Causal Relationships**: to determine cause-effect relationships, we need to know when they happened
   - ○ Handle **inconsistencies** between nodes
   - ○ Handle **conflicts**, such as multiple updates to a shared resource

Going a bit deeper, there are distributed systems which are highly time-dependant:

- ◇ Distributed **databases** - data must be consistent and transactions must either entirely succeed or entirely fail
- ◇ **IoT** systems - issuing commands to devices may be related to measurements taken at a certain time, besides, command receival and execution is always time-sensitive (e.g. suppose you get a "turn left" command too late)
- ◇ Cloud computing **auto-scaling** and **load balancing** - if you measure that the need for resources is increasing and you instance more VMs, but actually the need measurement is 2 hours old, you wasted resources and *money*

3.1 Challenges

First of all, **Clock Drift** is one the key problems. Every machine has its own *physical* clock, which may gradually drift over time due to hardware imperfections. It matters because over time, unsynchronized clocks on different nodes will show different times, leading to inconsistent timestamps for events.

Also **network latencies** are an issue Messages between nodes can be delayed due to network congestion, causing events to be perceived in a different order than they occurred.

In DS nodes may **fail** or be **unreachable**, making it difficult to maintain synchronized time across all nodes.

3.1.1 NTP and PTP - Protocols Solving Drift

Network Time Protocol (NTP) is a protocol used to synchronize the clocks of computers over a network.

It works on stratum levels, where a stratum 0 device is a reference clock, a stratum 1 device is a server that gets time from a stratum 0 device, and so on up to stratum 15.

NTP can synchronize clocks to within milliseconds over the internet, but it is insufficient for environments needing higher precision.

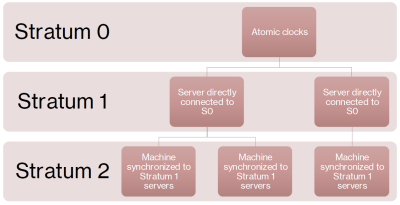


Figure 3.2: NTP Stratum architecture

PTP (Precision Time Protocol) is a protocol used to synchronize clocks in a network with sub-microsecond accuracy. Typically used in high-frequency trading, telecommunications, and industrial automation.

P TP uses a master-slave architecture, with a grandmaster clock providing time to all slaves. PTP timestamps are often hardware-assisted for higher precision (e.g., using Network Interface Cards (NICs) with timestamping capabilities).

According to prof. Dazzi, PTP allows for more precision than NTP, but it still limited by the underlying hardware:

if there isn't good hardware to support the protocol, then we can't achieve the high precision.

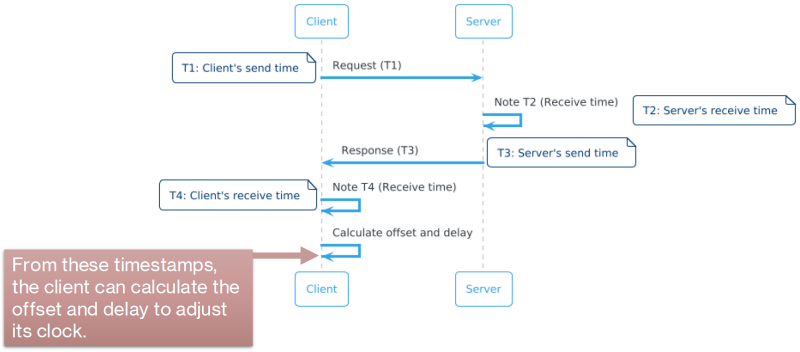


Figure 3.1: NTP protocol example

$$\theta = \frac{(T_2 - T_1) + (T_3 - T_4)}{2} \qquad \textit{Offset between clocks}$$
$$\sigma = (T_4 - T_1) - (T_3 - T_2) \qquad \textit{Round-trip delay}$$

NTP performs also some statistical analysis to filter out outliers and improve accuracy. Note that the first term $(T_2 - T_1)$ would be positive, while the second term $(T_3 - T_4)$. This is to account for the time it takes for messages to travel in both directions.
Note also that if the server's clock is ahead of the client's clock, the offset θ will be positive, indicating that the client needs to advance its clock, while if the server's clock is behind, θ will be negative, indicating that the client needs to set its clock back.

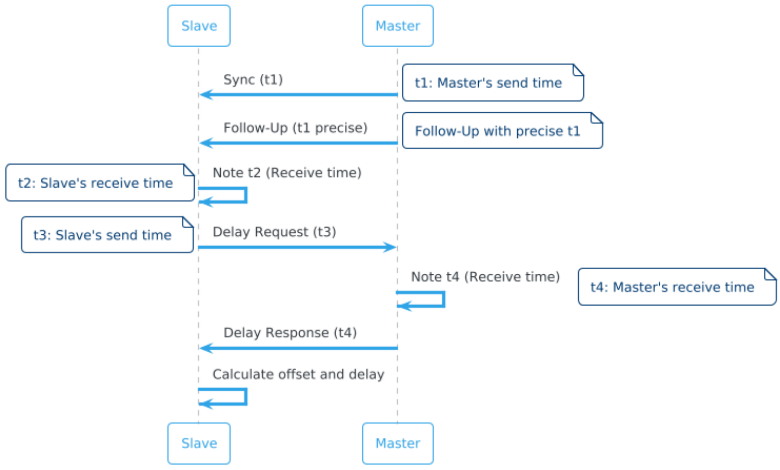


Figure 3.2: PTP protocol example

$$\theta = \frac{(t_2 - t_1) + (t_3 - t_4)}{2} \qquad \textit{Offset between clocks}$$
$$\sigma = \frac{(t_4 - t_1) - (t_3 - t_2)}{2} \qquad \textit{Round-trip delay}$$

In a simple architecture there are **slaves** and a **master**. In case of more complex architectures, there are multiple masters in a hierarchy. There is a **grandmaster clock** which is the most accurate clock in the network, and all other clocks, including regular masters, synchronize to it. **Boundary clocks** are used to connect different segments of the network, acting as both slaves to the grandmaster and masters to other slaves. **Transparent clocks** do not participate in the synchronization process but instead measure and account for the time taken by packets to traverse them, helping to improve overall accuracy.
PTP requires specialized hardware support to achieve high precision, such as network interface cards (NICs) that can timestamp packets at the hardware level.

PTP is used in specialized environments such as financial trading systems, telecommunications networks, and industrial automation where high precision time synchronization is critical. NTP is preferred for general-purpose applications where millisecond accuracy is sufficient, such as logging and internet services.

3.1.2 Logical and Phyisical Clocks

Logical clocks do not represent the actual time, but they are used to order events in a distributed system, ensuring that causally related events are ordered correctly even if the actual phyisical clocks are not synchronized.
Common examples are **Lamport** timestamps and vector clocks.

3.1.2.1 Lamport Timestamps

Lamport

1. Each process within this system maintains a counter which is incremented with each event.
2. When sending a message, a process includes its current counter value.
3. The receiving process then updates its counter to be greater than the highest value between its own counter and the received counter, ensuring a consistent sequence of events.

Lamport timestamps are useful in distributed file systems to maintain consistency across multiple clients accessing and modifying files concurrently. Notice however that Lamport timestamps do not capture causality completely; they only provide a partial ordering of events. This means that they cannot determine an ordering between two not-causally-related events.
Furthermore, adding Lamport timestamps to messages introduces some overhead, as each message must carry the timestamp information. This may become relevant in systems with high message throughput or limited bandwidth.

If Client A writes to a file and then Client B reads from the same file, Lamport timestamps can help ensure that Client B sees the updated file contents. Client A's write operation is assigned a timestamp, and when Client B reads the file, it updates its counter to reflect the most recent write. This ensures that all subsequent operations by Client B are correctly ordered after Client A's write, maintaining the consistency of the file system.

Being C_i the clock value of process P_i:

1. Before executing an event e, P_i sets $C_i = C_i + 1$
2. When P_i sends a message m, it sets $m.timestamp = C_i$.
3. Upon receiving a message m, process P_j sets $C_j = \max(C_j, m.timestamp) + 1$
   - ◇ These rules ensure that the logical clocks respect the causal relationships between events.

Lamport Example

The events a, b, c, d occur in the following sequence:

- ◇ P_1 executes event a.
- ◇ P_1 sends a message m (containing event b) to P_2.
- ◇ P_2 executes event c upon receiving m.
- ◇ P_2 sends a message n (containing event d) to P_3.

The logical clocks evolve as follows:

- ◇ P_1 executes event a: $C_1 = 1$
- ◇ P_1 sends m to P_2: $C_2 = 2$, $m.timestamp = 2$
- ◇ P_2 receives m and executes c: $C_2 = \max(C_2, 2) + 1 = 3$
- ◇ P_2 sends n to P_3: $C_2 = 4$, $n.timestamp = 4$
- ◇ P_3 receives n and updates its clock: $C_3 = \max(C_3, 4) + 1 = 5$

This ensures that the events are ordered as $a \to b \to c \to d$ across the processes.

3.1.2.2 Vector Clocks

Vector

1. Each process maintains a vector of counters, one for each process in the system.
2. When sending a message, a process increments its own counter in the vector, and includes the entire vector in the message.
3. The receiving process updates its vector by taking the element-wise maximum between its own vector (clock) and the received vector.

This helps in merging conflicts in collaborative editing applications such as Google Docs. By examining the vector clocks, the system can determine the causality of edits and merge changes appropriately, ensuring that all users see a consistent view of the document

Vector Clock Rules

- ◇ **Vector Clock**: Each process P_i maintains a vector VC_i of length N, where N is the total number of processes.
- ◇ Each element $VC_i[j]$ represents the logical clock of process P_j as known by P_i.
- ◇ **Initialization**: All elements of VC_i are initialized to zero.
- ◇ **Event Occurrence**: When an event occurs at process P_i, it increments its own entry in its vector clock: $VC_i[i] = VC_i[i] + 1$
- ◇ **Message Sending**: When P_i sends a message to P_j, it includes a copy of its vector clock VC_i in the message.
- ◇ **Message Receiving**: When P_j receives a message from P_i with vector clock VC_i, it updates its own vector clock: $VC_j[k] = \max(VC_j[k], VC_i[k])$ for all k. Then, P_j increments its own entry: $VC_j[j] = VC_j[j] + 1$

Implementing and maintaining vector clocks is more complex than simpler mechanisms like Lamport timestamps. The complexity increases with the number of processes in the system.

The size of the vector clock grows linearly with the number of processes, leading to higher memory and communication overhead. This can be a concern in systems with a large number of processes.

Chapter 4

Synchronizing

Synchronization refers to coordinating the actions of multiple processes that share resources.

In short, it involves the coordination of processes that access shared resources

- ◇ Needs for avoiding inconsitencies or conflicts
  - Prevent conflicts and inconsistencies that arise when multiple processes access shared data or resources concurrently.
- ◇ Managing access to shared resources
  - In a way that maintains data integrity and system performance.

In Distributed Systems the communication happens through networks and message passing among multiple independent proceses.

Challenges

1. Network Delays - Variable communication times may lead to inconsistencies
2. Process Failures - Processes may fail at any time, so their failure must be handled
3. No Fairness - We must preventing starvation or indefinite delays for processes waiting to access resources.

4.1 Mutual Exclusion

Mutual exclusion is a well-known **solution** to access shared resources in a distributed system, avoiding interferences.

Definition 4.1 (Mutual Exclusion) *A property that ensures only one process can enter a critical section at any given time.*

The **Critical section** is a portion of code accessing shared resources.

In other words, Mutual Exclusion enforces **atomical** access to shared resources, avoiding conflicts and inconsistencies. Note that *atomic* does not imply the hardware support for atomic operations, we use it as a logical and semantical concept.

Mutual Exclusion Goals

- ◇ **Safety** - Only one process can access the critical section at a time
- ◇ **Liveness** - Ensures that every process that wishes to enter the critical section will eventually be able to do so, preventing starvation.
- ◇ **Fairness** - Requests for entry to the critical section are granted in the order they are made, ensuring no process is perpetually denied access.

4.1.1 Types of Mutual Exclusion

1. **Software-based** - Algorithms such as Lamport's Bakery Algorithm, Peterson's Algorithm, Dekker's Algorithm, etc.
2. **Hardware-based** - Atomic operations or locks
3. **Hybrid approaches** - Combining software and hardware-based solutions to achieve better performance and reliability.

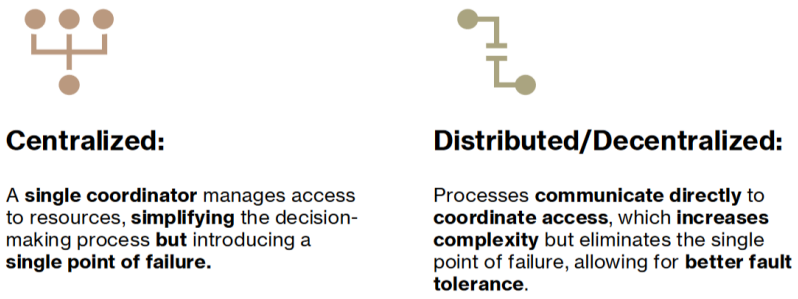


Figure 4.1: Mutual exclusion - Centralized vs Distributed

4.1.2 LBA - Lamport's Bakery Algorithm

LBA is a software-based mutual exclusion algorithm that uses a ticket system to ensure fairness and prevent starvation. LBA simulates a bakery where customers take a number and are served in order, clearly the processes are the customers.

1. A process picks a ticket number that is one greater than the maximum ticket number currently in use.
2. The process waits until all processes with smaller ticket numbers have completed their critical sections.
3. The process with the lowest ticket number enters the critical section.
4. The process resets its ticket number to indicate it has finished.

It is **unlikely** for two processes to pick the same ticket number due to the sequential nature of ticket assignment, but it is possible.

In the rare cases where multiple processes attempt to obtain a ticket simultaneously, they may end up with the same number. To solve this the algorithm specifies that the process with the smaller identifier (`pid`) has priority ensuring fairness among competing processes.

The process enters the critical section only after verifying that no other process with a smaller ticket number is currently in the critical section. Once the process completes its operations in the critical section, it releases its ticket by resetting its number to zero. The reset indicates to other processes that the critical section is available.

```
choosing[N] -> {false, false, ..., false} // Initialize choosing flags for each process
number[N] -> {0, 0, ..., 0} // Initialize ticket numbers for each process
...
Process (i):
   lock(i);
      critical session code
   unlock(i);

void lock(int i) {
   choosing[i] = true;
   number[i] = 1 + max(number[0], ..., number[N-1]);
   choosing[i] = false;
   for (int j = 0; j < N; j++) {
      // busy waiting
      while (choosing[j]); // Wait until other processes have chosen
      while (number[j] != 0 && (number[j] < number[i] || (number[j] == number[i] && j < i))
          );
   }
}
```

Note that unlike some hardware-based synchronization mechanisms (like spinlocks or test-and-set instructions), LBA does not require atomic operations or specific CPU instructions. Besides, LBA relies on simple ticketing logic and does not depend on hardware features, it can be implemented in various environments, enhancing its portability.

Its major **drawback** is that it may suffer from performance issues (e.g., high communication overhead) in large systems due to continuous polling (**busy waiting**) of ticket values. However, it nicely environments where shared memory is available and the number of processes is relatively small.

4.1.3 LDMEA - Lamport's Distributed Mutual Exclusion Algorithm

LBA *cannot* be directly applied in distributed systems, as there is no shared memory to store ticket values. In a distributed system, processes run on different nodes and must be aware of each other's states through message passing.

LDMEA is a distributed version of Lamport's Bakery Algorithm that allows processes to access shared resources across multiple nodes, exploiting logical clocks for coordination.

Figure 4.2: LDMEA Key Steps

Each process maintains a logical clock to timestamp its requests, which is incremented at each request or receive. Requests are ordered based on their logical timestamps, and the process with the smallest timestamp is granted access to the critical section.

Every site S_i, keeps a queue to store critical section requests ordered by their timestamps. $request\_queue_i$ denotes the queue of site S_i.

4.1.3.1 Entering the Critical Section

When a site S_i wants to enter the critical section, it sends a request message $Request(Ts_i, i)$ to all other sites and places the request on $request\_queue_i$. Here, Ts_i denotes the timestamp of Site Si. A site S_i can enter the critical section if it has received the message with timestamp larger than (Ts_i, i) from all other sites and its own request is at the top of $request\_queue_i$

Handling incoming requests: When a site S_j receives a $Request(Ts_i, i)$ from site S_i:

- S_j adds the request to its $request\_queue_j$ (ordered by timestamp)
- If S_j is **not** in the critical section or has a request with timestamp *larger* than (Ts_i, i), it immediately sends a *Reply* to S_i
- If S_j is **in** the critical section or has a pending request with timestamp *smaller* than (Ts_i, i), it **defers** the reply until it exits the critical section

This ensures that requests are processed in timestamp order, maintaining fairness and preventing deadlock.

4.1.3.2 Leaving the Critical Section

When a site S_i leaves the critical section, it removes its own request from the top of its request queue and it sends a release message $Release(Ts_i, i)$ to all other sites and removes its request from $request\_queue_i$.

When a site S_j receives the timestamped *Release* message from site S_i, it removes the request of S_i from its request queue.

4.1.3.3 Considerations and drawbacks

Example

1. **P1** sends a `REQUEST` message with its timestamp to **P2** and **P3**.
2. **P2** and **P3** reply to **P1**.
3. Once **P1** receives all `REPLY` messages, it enters the critical section.
4. After finishing, **P1** sends a `RELEASE` message to **P2** and **P3**

If **P1** sends a `REQUEST` message to **P2** and **P3**, it must wait for their replies before entering the critical section.
If both **P1** and **P2** send `REQUEST` messages, the process with the smaller timestamp will be granted access first, ensuring fairness.

Wrapping up we need

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

So, the first drawback is that LDMEA incurs a high number of messages ($3n$) for each entry into the critical section. Besides, LDMEA also has to handle **failures**: requires additional mechanisms to handle process failures and network partitions.

However, the message passing in LDMEA, aside from making it suitable for distributed systems, even though it introduces overhead, it is still better than the polling mechanism in LBA, where processes have to continuously check for ticket values.

4.1.4 Other approaches

Lamport LBA and LDMEA are just two examples of mutual exclusion algorithms. There are many other algorithms that provide mutual exclusion in distributed systems, each with its own strengths and weaknesses.

Typically they are classified in three categories:

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

4.2 Token-based Algorithms

A Token-Based Approach is a method used in distributed systems to manage access to a critical section. In this approach, a **unique token** circulates among the processes. Only the process that holds the token is allowed to enter the critical section, ensuring **mutual exclusion**.

Advantages

- ◇ **Efficiency** - When the token is well-managed, the system operates efficiently.
- ◇ Low Communication **Overhead** - There is minimal communication overhead when there is no contention for the token.

Drawbacks

- ◇ Vulnerability to **Token Loss** - If the token is lost, it can disrupt the entire system.
- ◇ Token Circulation **Delays** - Delays in token circulation can lead to inefficiencies and increased waiting times for processes.

4.2.1 Suzuki-Kasami Algorithm

The process holding the token has exclusive access to the critical section. Request messages are sent to all processes when a process wants to enter.
Keep in mind that Suzuki-Kasami does **not** exploit logical clocks.

Scenario

- ◇ P1 wants to enter the critical section.
- ◇ It sends a request to all other processes.
- ◇ If it holds the token, it enters the critical section.

4.2.1.1 Data Structures

Each **process** maintains one data structure:

- ◇ An array `RN_i[N]`(for Request Number), `i` being the `ID` of the process containing this array, where `RN_i[j]` stores the last Request Number received by `i` from `j`

The **token** contains two data structures:

- ◇ An array `LN[N]`(for Last request Number), where `LN[j]` stores the most recent Request Number of process `j` for which the token was successfully granted
- ◇ A queue `Q`, storing the `ID` of processes waiting for the token

4.2.1.2 Algorithm

Requesting the CS When process `i` wants to enter the CS, if it does not have the token, it:

- ◇ increments its sequence number `RN_i[i] := RN_i[i] + 1`
- ◇ sends a request message containing new sequence number to all processes in the system

Releasing the CS When process `i` leaves the CS, it:

- ◇ Sets `LN[i] := RN_i[i]`. This indicates that its request `RN_i[i]` has been executed.
- ◇ for every process k not in the token queue `Q`, it appends `k` to `Q` if `RN_i[k] == LN[k]+1`. This indicates that process `k` as a pending request
- ◇ if the token queue `Q` is not empty after this update, it pops a process ID `j` from `Q` and sends the token to `j`

- ◇ otherwise, it keeps the token

Performance

- ◇ Either 0 or N messages for CS invocation (no messages if process holds the token; otherwise N-1 request and 1 reply)
- ◇ Synchronization delay is 0 or N (N-1 requests and 1 reply)

The main two **issues** are discerning **outdated requests** from current ones, and determining which site is going to get the token next. Besides, if a token is lost, processes can hang; token recovery mechanisms or timeout strategies may be implemented to handle token loss.

On the other hand it is **efficient** in terms of message passing, as it only requires up to N messages for each CS invocation, and the synchronization delay is 0 or N (N-1 requests and 1 reply). It suits high-latency networks.

4.3 Non-token-based Algorithms

Processes communicate directly with each other to request permission to enter the critical section. There is ***no* central authority** or **token**, and there are messages exchanged to determine access rights.

4.3.1 Ricart-Agrawala Algorithm

Its goal is to achieve mutual exclusion by having processes send requests to each other: a process sends a request message to all other processes, waiting for replies to enter the critical section.

All processes communicate directly without a token in a decentralized fashion, and for each request $2(N-1)$ messages are exchanged.

Fairness is enforced by the timestamping mechanism, where the process with the smallest timestamp has priority: requests are served based on the order of arrival, governed by logical timestamps. If two processes have the same timestamp, the process with the lower ID gets priority.

Considering the previous points, the Cons are that message passing is high, and the algorithm is not suitable for high-latency and unreliable networks.

4.4 Quorum-based Algorithms

Instead of communicating with all processes, a process communicates with only a subset (quorum) of processes to get permission to enter the critical section.

4.4.1 Maekawa's Algorithm

A **quorum** is a subset of processes that must grant permission for mutual exclusion. Ensures that any two quorums overlap, guaranteeing access.

It has a reduced number of messages compared to the Ricart-Agrawala (Sec. 4.3.1 algorithm, and fits nicely environments with many **processes** and **high contention**.

4.5 Wrap Up

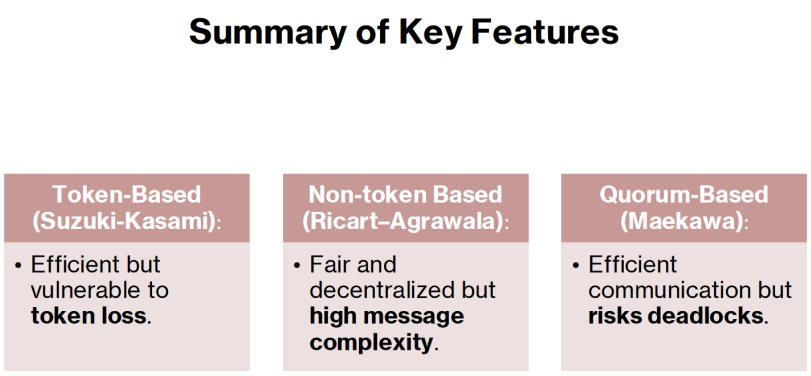


Figure 4.3: Summary of the Key features

Message complexity Message Complexity:

- ◇ **Suzuki-Kasami** - Low message overhead during token passing.
- ◇ **Ricart-Agrawala** - High message complexity due to multiple requests and replies.
- ◇ **Maekawa** - Reduced message count by only requiring quorum approval.

Scalability Considerations

- ◇ **Token**-Based - Scales well with fewer processes; token management becomes complex as the number of processes increases.
- ◇ **Non-token** Based - Scalability issues arise with an increasing number of processes due to high message overhead.
- ◇ **Quorum**-Based - Efficient for many processes, but the quorum size must be managed carefully to avoid performance degradation.

Potential Strategies

- ◇ **Suzuki-Kasami** - Implement token recovery mechanisms to recreate lost tokens.
- ◇ **Ricart-Agrawala** - Use timeouts to handle unresponsive processes.
- ◇ **Maekawa** - Adjust quorum sizes dynamically based on active processes.

Takeaway

- ◇ **Suzuki-Kasami** - Efficient token management but vulnerable to loss.
- ◇ **Ricart-Agrawala** - Fairness and decentralization but high message complexity.
- ◇ **Maekawa** - Reduced communication overhead with quorum mechanisms but requires careful management to avoid deadlocks.

Chapter 5

Deadlocks

Definition 5.1 (Deadlock) *A deadlock occurs when a set of processes is waiting for resources held by other processes in the set, resulting in a circular wait where no process can proceed.*

There are Four *Necessary* Conditions (*Coffman* Conditions) in order to have a deadlock:
- ◇ **Mutual Exclusion**: Resources cannot be shared.
- ◇ **Hold and Wait**: Processes holding resources can request new ones.
- ◇ **No Preemption**: Resources cannot be forcibly taken away.
- ◇ **Circular Wait**: A closed chain of processes exists, where each holds a resource needed by the next.

If any of the previous conditions is not met, a deadlock cannot occur.

There are three main approaches to handle deadlocks, which act at different points in the deadlock cycle, but in reality the last one is the only true and available solution; Prevention and Avoidance typically come at a way too high cost:
- ◇ **Prevention**: Ensures that at least one of the necessary conditions does not hold.
  - Acquire all resources before starting.
  - Preempt a process holding a needed resource.

  This approach typically leads to resource underutilization and low throughput.
- ◇ **Avoidance**: Checks dynamically if granting a resource would lead to a deadlock.
  - Banker's Algorithm (used in centralized systems).
  - Safe state evaluation in distributed environments.

  Avoidance requires *global state awareness*.
- ◇ **Detection and Recovery**: Identifies a deadlock state and then recovers from it.
  - Once detected, deadlocks can be resolved by aborting one or more processes or preempting resources
    - **Process Termination** - Abort one or more processes involved in the deadlock.
    - **Resource Preemption** - Forcefully take a resource from one process to give it to another. It is complex to implement without causing inconsistencies. It requires the ability to roll back (restore) the state of the preempted process.
    - **Rollback** - Roll back the state of one or more processes to a safe point before the deadlock occurred.

5.1 RAG - Resource Allocation Graph

The Resource Allocation Graph is a directed graph that models the allocation of resources to processes. It is composed of two types of nodes, processes and resources, and two types of edges, request and assignment. A cycle in the graph indicates a deadlock.

5.1.1 WFG - Wait-For Graph

A Wait-For Graph is a directed graph, a simplified RAG, that models the wait-for relationship between processes. It is composed of processes as nodes and edges representing the wait-for relationship. A cycle in the graph indicates a deadlock.

5.2 Detecting Deadlocks

There are some challenges in detecting deadlocks in distributed systems, such as the lack of a global state, communication delays, and the need for coordination among nodes. More specifically, we risk:

- ◇ False positives due to delays in the network or inconsistent state information
- ◇ Overhead due to message passing

- ◇ Dynamic changes in resource allocation

Deadlock detection in distributed systems involves monitoring process and resource states across different nodes and identifying circular wait conditions globally.
Correctness Conditions are Progress and Safety, which are, respectively:

- ◇ Progress: The algorithm must detect all deadlocks in finite time.
- ◇ Safety: No false deadlocks (called phantom or false deadlocks) should be reported.

5.2.1 Centralized Deadlock Detection

A central node gathers information from all other nodes and constructs a global wait-for graph. It is very simple, but not scalable, and the node can become a single point of failure.

5.2.2 Hierarchical Deadlock Detection

The system is divided into regions, with each region responsible for local deadlock detection. Regional coordinators communicate with higher-level nodes for global detection.

This approach provides better scalability than centralized detection, but, on the other hand, coordination between regions can be complex.

5.2.3 Distributed Deadlock Detection

Here all nodes participate in deadlock detection by sharing partial information about resources and processes, and each node detects deadlocks locally and communicates with others to detect global deadlocks.

There is no single point of failure, and the approach scales well with larger systems, but poses more complex message passing and increased overhead.

5.2.4 System Model for Deadlock Detection - WFG

The distributed system is composed of asynchronous processes that communicate via message passing. A WFG models the state of the system.

- ◇ Nodes:
  - Processes.
- ◇ Edges:
  - Directed edges from P1 to P2 indicate that P1 is waiting for P2 to release a resource.
- ◇ Deadlock occurs when there's a cycle in the WFG

Assumptions

- ◇ Only reusable resources.
- ◇ Exclusive resource access.
- ◇ One copy of each resource.
- ◇ Two process states: Running (active) or Blocked (waiting for resources).

The challenges in this system are **WFG maintenance**, which involves tracking resource dependencies across distributed nodes, and **cycle detection**, which involves searching cycles in the WFG. In particular, such search should find *all* cycles in *finite* time. Besides no false —aka *phantom*— deadlocks should be reported.

Having the Deadlocks detected, the system can then proceed to resolve them by **rollback**, **preemption**, or **termination**.

- ◇ Break the wait-for dependencies between deadlocked processes.
- ◇ Roll back or abort one or more processes to free up resources.
- ◇ Clean up the wait-for graph after resolution to avoid phantom deadlocks.
- ◇ Untimely and inappropriate cleaning of broken wait-for dependencies is the main reason why many deadlock detection algorithms reported in the literature are incorrect

5.3 Deadlock Models

5.3.1 Single Resource Model

In this model, each process requires a single resource to proceed. The deadlock detection algorithm is simple and efficient, but it is not very realistic. Since the maximum out-degree of a node in a WFG for the single resource model can be 1, the presence of a cycle in the WFG shall indicate that there is a deadlock

5.3.2 AND Model

A process can request multiple resources simultaneously, and the request is granted only if all resources are available. A cycle in the WFG —guess what— implies a deadlock.
In the AND model, if a cycle is detected in the WFG, it implies a deadlock but ***not*** *vice versa*. That is, a process may not be a part of a cycle, it can still be deadlocked.

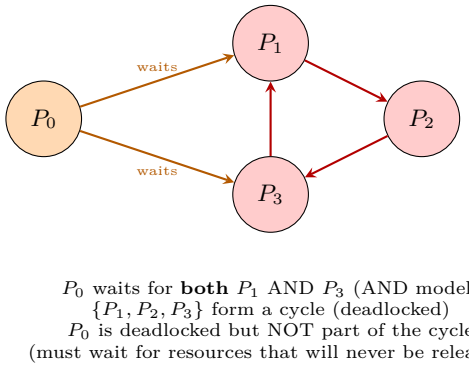


Figure 5.3: AND Model deadlock: P_0 (orange) is deadlocked without being part of the cycle. Since P_0 must wait for BOTH P_1 AND P_3 (AND semantics), and both are stuck in a cycle $\{P_1 \to P_2 \to P_3 \to P_1\}$, P_0 will never proceed even though it's not in the cycle itself.

5.3.3 OR Model

In the OR model, a process can make a request for numerous resources simultaneously and the request is satisfied if any one of the requested resources is granted.
Presence of a cycle in the WFG of an OR model does **not** imply a deadlock in the OR model.

In the OR model, the presence of a **knot** indicates a deadlock. A knot is a set of processes such that each process in the set is waiting for a resource held by another process in the set.
More formally, a knot K is a non-empty subset of processes such that:

- **Mutually reachable** - $\forall p, q \in K$ p is reachable from q and q is reachable from p (K is *strongly connected*).
- **No dead ends** - $\forall p \in K$, if p is waiting for a process r, then $r \in K$. In other words, there are no outgoing edges from processes in K to processes outside K (which could potentially lead to resource acquisition and deadlock resolution), so K is closed with respect to reachability towards processes that can provide resources

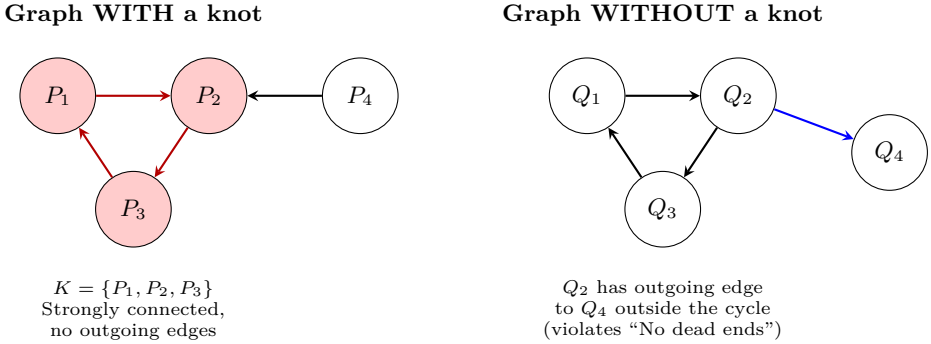


Figure 5.1: Comparison of graphs with and without a knot: Left shows a knot (red nodes form a strongly connected component with no outgoing edges). Right shows a cycle that is NOT a knot because Q_2 has an outgoing edge to Q_4 outside the cycle, potentially allowing deadlock resolution.

Note that, there can be a deadlocked process that is **not** a part of a knot.
So, in an OR model, a blocked process P is deadlocked if it is either in a knot or it can only reach processes on a knot.

Remember that graph is directed, so reachability is **not** symmetric. A process p can reach process $q \in K$ if there is a directed path from p to q, but the reverse may not necessarily be true, so the "**No dead ends**" condition would still hold

5.4 Deadlock Detection Algorithms

5.4.1 Path-Pushing Algorithm - Global WFG

Distributed deadlocks are detected by maintaining an explicit global WFG

- ◇ to build a global WFG for each site of the distributed system
- ◇ Nodes propagate local WFG to other nodes.
- ◇ After the local data structure of each site is updated, WFG are updated
- ◇ Deadlocks are detected by identifying circular dependencies.

5.4.2 Edge-Chasing - Probing

Here, a process sends probes to check if it is part of a deadlock. Whenever a process that is executing receives a probe message, it simply discards this message and continues. Only blocked processes propagate probe messages along their outgoing edges[1]. If the probe returns to the initiating process, a deadlock is detected.

- ◇ P1 sends a *probe* to P2, which forwards it to P3.
- ◇ When the *probe* returns to P1, a deadlock is detected

A probe is a message used in edge-chasing algorithms to trace the path of resource dependencies.

5.4.3 Diffusion Computation Algorithm - Query Propagation

- ◇ An algorithm for deadlock detection based on diffusion computation.
- ◇ When a process is blocked, it propagates queries to other processes, otherwise it drops queries
- ◇ Queries are discarded by a running process and are echoed back by blocked processes in the following way:
  - When a blocked process first receives a query message for a particular deadlock detection initiation, it does not send a reply message until it has received a reply message for every query it sent (to its successors in the WFG).
  - For all subsequent queries for this deadlock detection initiation, it immediately sends back a reply message.
  - The initiator of a deadlock detection detects a deadlock when it has received a reply for every query it has sent out.

5.4.4 Global state detection-based algorithms

These algorithms exploit the following facts:

- ◇ a consistent snapshot of a distributed system can be obtained without freezing the underlying computation
- ◇ a consistent snapshot may not represent the system state at any moment in time, but if a stable property holds in the system before the snapshot collection is initiated, this property will still hold in the snapshot.

Therefore, distributed deadlocks can be detected by taking a **snapshot** of the system and examining it for the condition of a deadlock.

5.4.5 Deadlock Detection in Dynamic and Real-time Systems

In distributed systems with dynamic resources (e.g., cloud systems), resource requests and releases are constantly changing. Deadlock detection algorithms must adapt to these changes and avoid outdated information.

This clearly poses consistent challenges such as false detections, the need for constant updates, and handling frequent resource changes.

In real-time distributed systems, deadlocks must be detected and resolved quickly to meet timing constraints. Detection algorithms must have low latency and minimal overhead

5.5 Assignment

Study and eventually prepare a few slides on

- ◇ Misra-Chandy-Haas Algorithm for AND model
- ◇ Misra-Chandy-Haas Algorithm for OR model

In Misra-Chandy-Haas algorithm a process suspecting a deadlock sends a **probe** $\langle P_0, P_i, P_k \rangle$, where:

- ◇ P_0 is the process that initiated the probe
- ◇ P_i is the process that at the current iteration sent the probe to P_k
- ◇ P_k is the process that P_i is waiting for

When the probe returns to P_0, a deadlock is detected and confirmed.

[1] I suppose of a WFG

The key difference between the AND and OR models is in the way the probe is propagated. In the AND model, the probe is propagated along **all** outgoing edges, while in the OR model, the probe is propagated along **only one** outgoing edge.

5.5.1 Misra-Chandy-Haas Algorithm - Detailed Explanation

The Misra-Chandy-Haas algorithm is an edge-chasing, probe-based deadlock detection algorithm that works in distributed systems. It is designed to detect cycles in the Wait-For Graph (WFG) by propagating probe messages along dependency edges.

5.5.1.1 Key Concepts

- ◇ **Probe Message**: A triplet $\langle P_0, P_i, P_k \rangle$ where:
  - P_0 is the *initiator* (the process suspecting it might be deadlocked)
  - P_i is the *sender* (the process currently forwarding the probe)
  - P_k is the *receiver* (the process that P_i is waiting for)
- ◇ **Probe Propagation**: Only blocked processes propagate probes; running processes discard them immediately.
- ◇ **Deadlock Detection**: If a probe returns to its initiator P_0 (i.e., P_0 receives a probe $\langle P_0, P_j, P_0 \rangle$), a cycle is detected and a deadlock is confirmed.

5.5.1.2 Algorithm for AND Model

In the AND model, a blocked process must wait for *all* resources it has requested. Therefore, when a blocked process P_k receives a probe $\langle P_0, P_i, P_k \rangle$:

1. If $P_k = P_0$, a deadlock is detected (the probe has completed a cycle).
2. If P_k is running, it discards the probe.
3. If P_k is blocked and waiting for processes $\{Q_1, Q_2, \ldots, Q_n\}$, it forwards the probe to **all** of them:

$$\langle P_0, P_k, Q_1 \rangle, \quad \langle P_0, P_k, Q_2 \rangle, \quad \ldots, \quad \langle P_0, P_k, Q_n \rangle$$

This exhaustive propagation ensures that all dependency paths are explored, which is necessary because in the AND model, *all* dependencies must be satisfied.

5.5.1.3 Algorithm for OR Model

In the OR model, a blocked process needs only *one* of the requested resources to proceed. Therefore, when a blocked process P_k receives a probe $\langle P_0, P_i, P_k \rangle$:

1. If $P_k = P_0$, a potential deadlock is detected.
2. If P_k is running, it discards the probe.
3. If P_k is blocked and waiting for processes $\{Q_1, Q_2, \ldots, Q_n\}$, it forwards the probe to **only one** of them (typically following a single outgoing edge):

$$\langle P_0, P_k, Q_j \rangle \quad \text{for some } j \in \{1, \ldots, n\}$$

This selective propagation reflects the semantics of the OR model: since any one resource can break the wait, exploring a single path is sufficient to detect a knot. However, detecting all deadlocked processes in the OR model is more complex, as a process can be deadlocked even if not part of a cycle (if all its paths lead to a knot).

5.5.1.4 Example: AND Model

Consider processes $P_1 \to P_2 \to P_3 \to P_1$ (a cycle).

- ◇ P_1 initiates a probe $\langle P_1, P_1, P_2 \rangle$.
- ◇ P_2 (blocked, waiting for P_3) forwards $\langle P_1, P_2, P_3 \rangle$.
- ◇ P_3 (blocked, waiting for P_1) forwards $\langle P_1, P_3, P_1 \rangle$.
- ◇ P_1 receives the probe with itself as the receiver $\Rightarrow$ deadlock detected.

5.5.1.5 Example: OR Model

Consider $P_1 \to P_2$, $P_1 \to P_3$, $P_2 \to P_3$, $P_3 \to P_1$ (a knot).

- ◇ P_1 initiates $\langle P_1, P_1, P_2 \rangle$ (choosing one edge).
- ◇ P_2 forwards $\langle P_1, P_2, P_3 \rangle$.
- ◇ P_3 forwards $\langle P_1, P_3, P_1 \rangle$.
- ◇ P_1 receives the probe $\Rightarrow$ potential deadlock (knot) detected.

5.5.1.6 Advantages and Limitations

- **Advantages**:
  - Distributed and scalable (no central coordinator).
  - Low message overhead compared to global WFG construction.
  - Detects deadlocks efficiently by tracing dependency chains.
- **Limitations**:
  - In the OR model, detecting all deadlocked processes (including those not in a cycle but reachable only to a knot) is more complex.
  - False positives (phantom deadlocks) can occur if the WFG changes during probe propagation.
  - Requires careful synchronization to avoid race conditions.

Part II

Stabilization

Chapter 6

Self-stabilization

6.1 Self-stabilization

In a distributed system a large number of systems are widely distributed and frequently communicate, so there is a chance to end up in an **illegitimate** state in case, for instance, a message is lost.
Among the previously mentioned algorithm and scenarios, consider *token-based* systems: what if the token is lost? Drama ☹

Note that the meaning of *illegitimate* and *legitimate* depends on the application.

Definition 6.1 (Self-stabilization) *Regardless of the initial state, the system is guaranteed to converge to a legitimate state in a* ***finite*** *number of steps by itself* ***without*** *any* ***outside intervention****.*

6.1.1 Challenges

The main challenge in self-stabilization is that nodes in a distributed system do not have a *global memory* that they can access instantaneously. Each node has to rely on local knowledge and messages from neighbors to make decisions, but their actions must achieve a *global objective*.

6.2 System Model

A Distributed system (DS) model comprises a set of n machines called *processors* that communicate with each other.

- ◇ Denote the i^{th} processor in the system by P_i.
- ◇ Neighbors of a processor are processors that are *directly* connected to it.
- ◇ Neighbors communicate by sending and receiving messages.
- ◇ DS is a **graph** in which each processor is represented by a node and every pair of neighbouring nodes are connected by link.
- ◇ FIFO queues are used to model channels for asynchronous delivery of messages: Q_{ij} contains all messages sent by a processor P_i to its neighbor P_j that have not yet been received.
- ◇ Each processor is characterized by its state.
- ◇ A full description of a DS at a particular time consists of the state of every processor and the content of every queue.

Note that this model is quite simplistic. In reality messages are almost *never* delivered in FIFO order: communications over networks are unpredictable and unreliable.

6.2.1 System Configuration

The term **system configuration** is used to describe a DS. A configuration is denoted by $c = s_1, s_2, \ldots, s_n, q_{1,2}, q_{1,3}, \ldots, q_{n,n-1}$ where s_i is the state of processor P_i and $q_{i,j}$ $(i \neq j)$ is the content of the queue from P_i to P_j.

6.2.2 Network Assumptions

- ◇ Let N be an upper bound on n (the number of processors).
  N is important because we may consider dynamic systems
- ◇ Let γ denote the **diameter** of the network, i.e., the maximum number of links in any path between any pair of processors.
  Knowing the diameter may provide a bound on the number of hops required for a message to reach any processor from any other processor, ultimately giving a rough upper bound for latency.

- ◇ A network is **static** if the communication topology remains fixed. It is dynamic if links and network nodes can go down and recover later.
- ◇ In the context of dynamic systems, self-stabilization refers to the time after the "final" link or node failure. The term "final failure" is typical in the literature on self-stabilization
- ◇ Since stabilization is only guaranteed eventually, the assumption that faults eventually stop to occur implies that there are no faults in the system for "sufficiently long period" for the system to stabilize.
- ◇ In any case, it is assumed that the topology remains connected, i.e., there exists a path between any two nodes.

6.3 Self-stabilizing formally said

We define self-stabilization for a system S with respect to a predicate P over its set of global states, where P is intended to identify its correct execution.

States satisfying P are called **legitimate** (safe) states, and those that do not are called **illegitimate** (unsafe) states.

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

1. **Closure** - P is closed under the execution of S: that is, once P is established in S, it cannot be falsified, it can't go in an illegitimate state due to its normal execution, only transient faults can bring it to an illegitimate state.
   A **transient**[1] **fault** is a temporary fault that may arbitrarily change the state of the system, it may be external (e.g., a cosmic ray flipping a bit in memory) or internal (e.g., a software bug causing incorrect state updates), but independent from the normal operation of the system.
   Persistent faults such as a broken link or a crashed (unreplaced) node/hardware are not considered transient faults.
2. **Convergence** - starting from an arbitrary global state S is guaranteed to reach a global state satisfying P within a finite number of state transitions.

Safe states are those that are reachable under normal program execution from the set of legitimate start states, referred as the **reachable set**. So, when we say that a program is self-stabilizing without explicitly mentioning a predicate, we mean with respect to the reachable set. By definition, the **reachable set** is closed under program execution, and it corresponds to a predicate over the set of states.

6.3.1 Classification

Self-stabilization may be classified as either **randomized** or **probabilistic**, depending on the stabilization time.

- ◇ A system is said to be randomized self-stabilizing system, if and only if it is self-stabilizing and the expected number of rounds needed to reach a correct state (legal state) is bounded by some constant k.
- ◇ A system S is said to be **probabilistically self-stabilizing** with respect to a predicate P if it satisfies the following two properties:
  - **Closure**: P is closed under the execution of S. That is, once P is established in S, it cannot be falsified.
  - **Convergence**: There exists a function f from natural numbers to $[0, 1]$ satisfying $\lim_{k\to\infty} f(k) = 0$, such that the **probability of reaching a state satisfying** P, starting from an arbitrary global state within k state transitions, is $1 - f(k)$.

Pseudo or non-pseudo

A **pseudo-stabilizing** system is one that, if started in an arbitrary state, is guaranteed to reach a state after which it ***does not*** deviate from its intended specification.
A stabilizing system, instead, is one that, if started at an arbitrary state, is guaranteed to reach a state after which it ***cannot*** deviate from its intended specification.

Thus, the difference between the two notions comes down to the difference between "*cannot*" and "*does not*" – a difference that hardly matters in many practical situations.

Transient failures

A **transient failure** is *temporary* (short lived) and it does not persist. A transient failure may be caused by corruption of local state of processes or by corruption of channels. A transient failure may change the **state** of the system, but *not* its behaviour. Persistent faults such as a broken link or a crashed (unreplaced) node/hardware are not considered transient faults.

[1] "Transitorio"/"Temporaneo" in italian

6.3.2 Issues in the design of self-stabilizing systems

Some of the main issues are:

- ◇ **Number of states** in each of the individual units in a distributed systems
- ◇ **Uniform** and **non-uniform** algorithms in distributed systems
- ◇ Central and distributed demon
- ◇ Reducing the number of states in a token ring
- ◇ Shared memory models
- ◇ Mutual exclusion
- ◇ Costs of self-stabilization

6.3.3 Dijkstra's Token Ring Algorithm

His system consisted of a set of n finite-state machines connected in the form of a **ring**.

He defines a *privilege* of a machine as the ability to change its current state. This ability is based on a boolean preicate that consists of its current state and the states of its neighbors.

When a machine has a privilege it is able to change its current state, which is referred to as a **move**.
Furthermore, when multiple machines enjoy a privilege at the same time, the choice of the machine that is entitled to make a move is made by a central daemon, which arbitrarily decides the order which privileged machine is allowed to move.

A legitimate state must satisfy the following constraints:

1. There must be at least one privilege in the system (liveness, no deadlock).
2. Every from move from a legal state must again put the system into a legal state (closure).
3. During an infinite execution, each machine should enjoy a privilege an infinite number of times (no starvation).
4. Given any two legal states, there is a series of moves that changes one legal state to the other (reachability).

Dijkstra considered a legitimate (or legal) state as one in which *exactly one machine enjoys the privilege*.

- ◇ This corresponds to a form of mutual exclusion, because the privileged process is the only process that is allowed in its critical section.
- ◇ Once the process leaves the critical section, it passes the privilege to one of its neighbours.

With this background, let's see how the above issues affect the design of a self-stabilization algorithm.

6.3.4 First Solution

Machine 0 is exceptional, and the other machines are identical, and follow the same following algorithm:

- ◇ each machine compares its state with the state of the anti-clockwise neighbour;
- ◇ if they are not the same, it updates its state to be the same as its neighbour.

if there are n machines and each of them is initially at a random state $r \in K$ (K is the set of possible states, with $|K| \geq n$), then all the machines (except the exceptional machine, machine 0) whose states are different from the one of their anti-clockwise neighbour are said to be **privileged**.
There is a *central demon* that decides which of these privileged machines will make the move

Below is the code for each machine, where L and R are the states of the left and right neighbors, respectively, and S is the state of the machine itself.

- ◇ **Machine 0** (exceptional)

```
if L == S then
   S := (S+1) mod K
```

- ◇ **Other machines** ($i, 1 < i < n-1$)

```
if L != S then
   S := L
```

- ◇ Suppose machine 6 (assume $n \gg 6$) makes the first move
- ◇ Its state is not the same as that of machine 5 and hence it had the privilege to make the move and finally sets its state to be the same as that of machine 5
- ◇ Now machine 6 loses its privilege as its state is same as that of its anti- clockwise neighbour (machine 5).
- ◇ Next, suppose machine 7, whose state is different from the state of machine 6, is given the privilege.
- ◇ It results in making the state of machine 7 the same as that of machine 6.
- ◇ Now machines 5, 6, and 7 are in the same state.

Eventually, all the machines will be in the same state in the similar manner.
At this point, only the exceptional machine (machine 0) will be privileged as its condition $L = S$ is satisfied.
Now there exists only one privilege or token in the system (machine 0), which increments its state by 1 (mod K), making the next machine (machine 1) privileged, as its state is now different from that of machine 0. In turn, machine 1 updates its state to be the same as that of machine 0, making machine 2 privileged, and so on. Thus, the privilege circulates around the ring.
This solution, rather simple, requires $K \geq n$ states for each machine to guarantee self-stabilization, besides, also an ordering on the machines is required.

6.3.5 Second Solution

We can improve the first solution.

The second solution uses only three state machines $0, 1, 2$. In the first solution there is only one exceptional machine, the one with state 0. Here, the machines with state 0 and n-1 are exceptional, the former referred to as *bottom machine*, the second as *top machine*. Below is the code for each machine, where L and R are the states of the left and right neighbors, respectively, and S is the state of the machine itself.

- **Bottom** machine (state 0)

```
if (S+1) mod 3 == R then
   S := (S-1) mod 3
```

  Its state depends upon its current state and the state of its right neighbor.
- **Top** machine (state $n-1$)

```
if L == R and (L+1) mod 3 != S then
   S := (L+1) mod 3     // note that L == R
```

  Its state depends upon the states of both its left and right neighbors (the *bottom machine* and machine $n-2$). The condition states that the left and right neighbors must be in the same state, and the state of the top machine must not be equal to the incremented state of its neighbors.
- Other machines $(i, 1 < i < n-1)$

```
if (S+1) mod 3 == L then
   S := L
if (S+1) mod 3 == R then
   S := R
```

  Their state depends upon the states of both their left and right neighbors. If the first condition is not satisfied, the second one is checked.

This schema forces the system to always have at least one privilege, hence to self-stabilize.
Initially, three privileges exist in the system. Actually the number of privileged machines converges linearly to 1.

State of machine 0	State of machine 1	State of machine 2	State of machine 3	Privileged machines	Machine to make move
0	1	0	2	0, 2, 3	0
2	1	0	2	1, 2	1
2	2	0	2	1	1
2	0	0	2	0	0
1	0	0	2	1	1
1	1	0	2	2	2
1	1	1	2	2	2
1	1	2	2	1	1
1	2	2	2	0	0
0	2	2	2	1	1
0	0	2	2	2	2
0	0	0	2	3	3
0	0	0	1	2	2

Table 6.1: Execution trace of Dijkstra's second solution with 4 machines. Here machine 0 is the bottom machine and machine 3 is the top machine.
In the first row, the privileged machines are 0, 2, and 3, because their conditions are satisfied. The central demon chooses machine 0 to make the move. After machine 0 makes the move, the new configuration is shown in the second row, and so on.

6.3.5.1 Observations

The number of states in each of the the individual units that each machine must have for the self-stabilization is an important issue. Dijkstra offered three solutions for a directed ring with n machines, $0, 1, \ldots, n-1$, each having K states, $K \geq n, K = 4, K = 3$.

6.4 To be or not to be...Uniform

In a distributed system, it is desirable and also possible to have each machine use the same algorithm. However, to design self-stabilizing systems, it is often necessary to have different machines use different algorithms.

The individual processes can be anonymous, meaning they are indistinguishable, and all run the same algorithm. Often, anonymous networks are called uniform networks. A network is semi-uniform if there is one process (the root) which executes a different algorithm.

Uniformity may lead to not being able to decide who is entitled to make a move, which may lead to a deadlock.

Generally, the presence of a central demon is assumed in self-stabilizing systems. Even though a demon is considered an undesireable constraint in distributed system, it is acceptable to have a *distributed demon* in such cases. Some stabilization algorithms were designed to work with a centralized demon, but they can be easily adjusted to work with a distributed demon.

6.5 Costs of Self-Stabilization

Self-stabilization per se does not define an upper bound on the time required for convergence. Two measures are used to evaluate the performance of a self-stabilizing system:

- ◇ **Convergence span**: the maximum time required for the system to reach a legitimate state from an arbitrary initial state.
- ◇ **Response span**: the maximum time required for the system to return to a legitimate state after a perturbation (transient fault) that puts the system in an illegitimate state.

Clearly, the aim of the designer of a self-stabilizing system is to reduce the convergence span and the response span.

The Time-complexity measure for self-stabilizing systems is the number of **rounds**.

Generally, communicatino between any two processors in a given system takes at least σ rounds, where σ is the **synchronization delay** of the system (?).

6.6 Designing Self-Stabilizing Systems

Self-stabilization is characterized in terms of a "malicious adversary" that may disrupt the system.

In case the adversary succeeds, a self-stabilizing system is able to recover from the disruption and return to a legitimate state.

A common technique is **layering**, as it happens for internet protocols. This is thanks to the *transitivity* property of self-stabilization: if a system is self-stabilizing and a subsystem is self-stabilizing, then the composed system is self-stabilizing.
If $P \to Q$ (P stabilizes Q) and $Q \to R$, then $P \to R$.

- ◇ Thus, different layers of self-stabilizing programs (each by itself self-stabilizing) can be composed.
- ◇ First step is to build a self-stabilizing "platform" and any program written on that platform automatically becomes self-stabilizing.
- ◇ The basic idea behind a self-stabilizing platform is to provide primitives that can be used to write other programs.
- ◇ To develop self-stabilizing systems using the technique of layering, we require primitives to provide structures on which algorithms may be built.

6.6.1 Main structuring mechanism primitives

We have common clock primitives and topology-based primitives.

Note that a "primitive" is a basic building block that can be used to construct more complex systems or algorithms. They are the fundamental operations or functionalities that serve as the foundation for higher-level abstractions.
In the discussion below, "unison" and "leader election" are examples of such primitives, so mechanisms that provide basic functionalities that can be used to build more complex algorithms.

These primitives serve as foundational building blocks for constructing more complex distributed algorithms and systems, and are associated to key problems in self-stabilization.

First of all we must address clocks. **Unison** is the process of maintaining time through the use of local clocks in shared memory systems. There are two properties required:

- ◇ **Safety**: All clocks have the same value (or differ by at most 1).
- ◇ **Progress** (*Liveness*): each clock is incremented infinitely often by the same amount

Leader election is the most basic primitive with respect to arbitrary dynamic topologies. Electing a leader allows to build a spanning tree, which is a fundamental structure for many distributed algorithms. A spanning tree is a subgraph that includes all the vertices of the original graph, is connected, and has no cycles. A spanning tree with a designated root is called a **rooted spanning tree**.

6.6.2 Communication Protocols

A communication protocol is a collection of processes that exchange messages over communication links in a network.

A protocol may be adversely affected for several reasons:

- ◇ Initialization to an illegal state.
- ◇ A change in the mode of operation. Not all processes get the request for the change at the same time, so an illegal global state may occur.
- ◇ Transmission errors because of message loss or corruption.
- ◇ Process failure and recovery.
- ◇ A local memory crash which changes the local state of a process.

If a protocol is self-stabilizing, they will all be corrected in a finite number of steps, regardless of the reason for the loss of coordination. A communication protocol is stabilizing if and only if starting from any unsafe state (i.e., one that violates the intended invariant of the protocol), the protocol is guaranteed to converge to a safe state within a finite number of state transitions. Stabilization allows the processes in a protocol to reestablish coordination between one another whenever coordination is lost due to some failure.

Gouda and Multari showed that a communication protocol must satisfy the following three properties to be self-stabilizing:

- ◇ It must be *non-terminating* (i.e., it must not reach a state from which no further state transitions are possible).
- ◇ There is an *infinite* number of safe states (i.e. states that satisfy the intended invariant of the protocol) in every computation.
- ◇ There are *timeout actions* in a non-empty subset of processes. These actions are enabled in every state of the process and can be executed at any time.

The reason for the first property is that if a protocol terminates, it may terminate in an unsafe state.
The second property is required because if there are only a finite number of safe states, it is possible that the protocol may leave the safe states and never return to one.
The third property is required because if there are no timeout actions, it is possible that the protocol may reach a state in which no further state transitions are possible.

The spanning-tree construction in distributed systems is a fundamental operation that forms the basis for many other network algorithms, and in fact we've seen how many self-stabilizing algorithms rely on the knowledge of the neighbors.

Chapter 7

Consesus

7.1 Introduction

Safety and **liveness** are two fundamental properties of distributed systems. Safety is about avoiding incorrect decisions, while liveness is about —eventually— making progress.

Consesus is about finding an agreement over a value.

7.1.1 Importance

With Distributed databases & replication it is necessary to ensure consistency across replicas, or at least to ensure that the replicas eventually converge to a consistent state, implementing a synchronization mechanism.

Transparency of a consesus algorithm to the DBMS depends on the level of abstraction, or the viewpoint considered in a given case. An SQL programmer may not care of a consesus algorithm on top, but a DBA (Admin) should.

In general it may be also important for synchronizing behaviour in a distributed system, for example to agree on a leader, or allocate resources.

Consensus algorithms find wide application in **Blockchains** and **Cryptocurrencies**, where the agreement is about the validity of a transaction. An example are both Proof of Work and Proof of Stake, exploited by Bitcoin and Ethereum respectively.

Also in **Microservices** architectures, consesus is important for coordinating services and ensuring data consistency across distributed components.

7.2 Mechanisms for Consensus

There are various ways for implementing consensus. They typically exploit some of the following mechanisms:

- ◇ Majority agreement: Paxos works by ensuring a majority of nodes agree on a value.
- ◇ Leader-based coordination: Sometimes protocols use a leader to drive agreement, such as in Raft.
- ◇ Voting and Quorums: Nodes vote on proposed values, and a *Quorum* (majority) must agree. The difference with majority agreement is that a quorum is a *subset* of nodes, not necessarily all the nodes in the system.
- ◇ Handling failures: Consesus algorithms are designed to tolerate node and network failures: timeouts, retries…

7.3 Safety and Liveness

Safety ensures that the systems never reaches an incorrect state. In the context of consensus, safety means that if a value is decided, it must be the same across all nodes.
Safety can be thought of as the “nothing bad happens”. It prioritizes correctness: Even under failure conditions, the system should not violate its invariants.

Liveness ensures that the system eventually makes progress and reaches a decision. It can be thought as the “something good eventually happens”. The system must continue to work, eventually deciding on a value despite failure or delays. In other words, we must not have a deadlock ☺. In the context of consensus, liveness means that all working nodes eventually decide on a value.

Ensuring safety delays progress.

There is a trade-off between safety and liveness. Liveness might occasionally compromise safety by making premature decisions, especially asynchronous systems. For example, a system might decide on a value before all nodes have agreed on it.

7.3.1 FLP Impossiblity Theorem

Definition 7.1 (FLP Impossibility Theorem) *In an asynchronous system, with the possibility of node failures, it is impossible to have a consesus algorithm that is both safe and live.*

This is a bit pessimistic, but it is a fundamental result in distributed systems.

The theorem is named after Fischer, Lynch, and Paterson, who proved it in 1985.

This is an important result, since it highlights the need of a trade-off between safety and liveness in distributed systems.

7.3.1.1 Proof

1. Starting Configuration - Nodes must agree on a decision (value)
2. Indecision Phase - As messages are exchanged there exists a state where no node has yet enough information to decide on a value yet. This state is prolonged by message delays and process failures.
3. Failure example - If one node crashes or its messages are indefinitely delayed, other nodes are forced to wait, leading to a situation where the system the systems cannot reach consesus
4. Conclusion - Since there is no way to distinguish between slow and crashed nodes, and since nodes rely on receiving messages to make decisions, it's impossibile to guarantee both safety and liveness in asynchronous systems with faults.

At step 3 we understand that in order to ensure Liveness, we must sacrifice safety, because we don't know what the unavailable node (either crashed or slow) would have decided. On the other hand, ensuring safety means that we must wait for the node to recover, which compromises liveness.

7.3.1.2 Implications

There is an impact on consesus algortihms: FLP shows that we must relax some assumptions on properties in real-world systems. Raft and Paxos manage this trade-off by making practical compromises.

7.4 Paxos

Paxos is a family of algorithms for solving consesus in a network of unreliable processors. It was first described by Leslie Lamport in 1998, but initially proposed in 1989.

7.4.1 Components

Paxos is completely asynchronous, so it does **not** assume that all nodes are in the same status when the algorithm starts.

Roles
- ◇ Proposers - Propose a value to be agreed upon.
- ◇ Acceptors - Accept a value proposed by a Proposer.
- ◇ Learners - Learn the agreed value.

1. Phase 1
   a. **Prepare** - Proposer sends a Prepare request with a proposal number. Then the proposer chooses a Quorum of Acceptors and sends the Prepare message containing a proposal number n to them.
   b. **Promise** - Acceptors respond with a Promise if the proposal number n is higher than any they've previously seen. Otherwise they may not respond or send a negative message back.
2. Phase 2
   a. **Accept** - Upon receiving a Promise from a majority of Acceptors, the Proposer sends an Accept request.

b. **Accepted** - If an ACCEPTOR receives an ACCEPT message from a PROPOSER, it must accept it *if and only if* it has not already *promised* (in Phase 1b) to only consider proposals having an identifier greater than other received.

7.4.1.1 Why the Promise Check is Essential

The promise mechanism in Phase 2b guarantees **safety** by preventing older proposals from overriding newer ones.

When an ACCEPTOR sends a PROMISE(N) in Phase 1b, it commits to ignoring any future proposals with number $< n$.

Example scenario without this check:

1. Proposer A sends PREPARE(10) → Acceptor responds PROMISE(10)
2. Proposer B sends PREPARE(20) → Acceptor responds PROMISE(20) (more recent!)
3. Proposer A sends ACCEPT(10, VALUE_A)
   - ◇ *Without the check*: Acceptor would accept *value_A*
   - ◇ *With the check*: Acceptor rejects because it already promised to only consider $n \geq 20$

This prevents "stale" proposals from being accepted after newer proposals have been initiated, ensuring the system converges to a single consistent value and avoiding race conditions that could lead to conflicting decisions.

If faulty nodes are f and total nodes are $n \geq 2f + 1$, then Paxos can tolerate f faulty nodes, since it exploits strict majority.

7.4.2 Challenges and Applications

Challenges

- ◇ Complexity in understanding and implementing the algorithm
- ◇ Overhead due to message exchanges
- ◇ Network partitions and node failures can affect Liveness

Applications

- ◇ Distributed databases (e.g., Google Chubby).
- ◇ Replicated state machines.
- ◇ Systems requiring strong consistency guarantees.

7.5 Key Takeaways

Consensus is essential for coordination in distributed systems, and it is a fundamental problem in distributed computing. Ensures multiple processes agree on a single value despite failures.

Chapter 8

Raft

8.1 Logs use case

Before delving into Raft, let's consider an interesting use case.

Bear in mind that a **log** is a *sequence* of records, it is meaningless if not ordered. Operations have an effect when executed in an order s, and may have a different one in another order s'.

In distributed systems, it is not trivial to keep logs consistent. On local machines, timestamp ordering is enough, but in distributed systems, clocks are not synchronized, hence we need a more sophisticated approach, typically involving the sequencing of operations, not necessarily based on timestamps.

8.2 Consesus - (Again)

Consesus is the backbone of consistency, fault tolerance, and coordination in distributed systems; it enables systems to operate reliably and predictably, even in the presence of failures and network partitions.

8.2.1 Why is it relevant

- ◇ **Coordination**: Synchronizes independent nodes, such as in *Microservices*, to maintain a consistent state.
- ◇ **Fault Tolerance**: Ensures progress and recovery in the presence of failures.
- ◇ **Data Integrity**: Prevents data divergence in replicated systems.
- ◇ **Avoids Split-Brain**: Manages network partitions to prevent conflicting operations.
- ◇ **Strong Consistency**: Guarantees up-to-date data in distributed databases.
- ◇ **Leader Election**: Allows seamless transitions of leadership in distributed environments.
- ◇ **Blockchain and Transaction atomicity**: Ensures reliable distributed transactions and maintains consensus in decentralized networks.
- ◇ **State Machine Replication**: Maintains consistent replicated states across systems.

> **Converging**
>
> Amongst other things, consesus is used to ensure ***data integrity***, *i.e. prevents* ***data divergence*** *in replicated systems.* That is that different data (replicas) will eventually converge to the same value.
>
> *But to which of the replicas should the others converge?*
> Consesus!

- ◇ **Safety** - "nothing bad happens"
  Safety ensures that the system never reaches an incorrect state, this is achieved by making sure no two nodes decide on different values; there may not happen any divergence, once a value is chosen, it is final and all nodes eventually agree on it.
- ◇ **Liveness** - "something good eventually happens"
  Liveness ensures that the system eventually makes progress and reaches a decision, guaranteeing that the system does not get stuck in an indecisive state, all non-faulty nodes eventually decide on a value.

Liveness might occasionally compromise safety by making premature decisions, especially in asynchronous systems. For example, network delays can cause a node to make a decision based on outdated information, violating safety. Processes can fail temporarily (crash) or permanently, while messages can be lost or delayed indefinitely. These failures make it impossible to guarantee both safety and liveness simultaneously in an asynchronous system, as stated by the FLP impossibility result.

In the real world, compromises are often made between safety and liveness based on the specific requirements of the system. Paxos and Raft carefully manage this trade-off, using timeouts or allowing temporary failures.

8.3 Paxos recalls

Paxos and paper

Even though it was a standard until a few years ago, people realized that Paxos was too complex to implement, and could fit best raw paper than actual software. Too many aspects were left to the implemented, and it was hard to get it right.

- **Roles**:
  - **Proposers** - Propose values to be agreed upon.
  - **Acceptors** - Vote on proposed values.
  - **Learners** - Learn the chosen value.
- **Phases**:
  - 1a - **Prepare Phase** - A proposer selects a proposal number and sends a prepare request to a quorum of acceptors.
  - 1b - **Promise Phase** - Acceptors respond with a promise not to accept any proposals with a lower number and may include the highest-numbered proposal they have accepted.
  - 2a - **Accept Phase** - If the proposer receives promises from a quorum of acceptors, it sends an accept request with the proposal number and value.
  - 2b - **Accepted Phase** - Acceptors respond to the accept request, and if a majority accept, the value is chosen.
- **Key Properties**:
  - **Safety** - Paxos guarantees that once a value is chosen, it is final and all nodes eventually agree on it.
  - **Liveness** - Paxos may not always make progress, especially in the presence of failures or network partitions

8.4 Raft

Paxos prioritizes safety over liveness. It guarantees that once a value is chosen, it's always safe, but there may be cases where it fails to make progress quickly.
Raft also ensures safety but uses leader election to optimize for liveness, making faster decisions when the network conditions allow it. Raft is **not** a byzantine fault tolerant (BFT) algorithm, nodes trust the leader.

Raft achieves consesus through an *elected leader* that coordinates the other nodes. An entity participating to a Raft cluster can either take the role of the **leader** or the role of **follower**; in the latter case, it may become leader through a *"candidacy"* process.
The leader regularly informs the followers of its existence by sending a heartbeat message.

- There exist a timeout for the heartbeats from the leader
- In case no heartbeat is received the follower changes its status to candidate and starts a leader election.
- Heartbeat messages may be piggybacked with log (see below) entries to replicate them to the followers.

Note that the **underlying assumption** is that *everybody knows everybody*.

8.4.1 Key points

- **Log** - Log contains a sequence of client commands and updates to be applied in the same order by all nodes.
- **Leader election** - Raft uses a randomized timeout to elect a leader.
- **Log replication** - The leader replicates its log to the followers.
- **Safety and Liveness** - Raft ensures safety and liveness under failure conditions.

8.4.2 Log Replication

Log replication is a fundamental mechanism in Raft to ensure that all nodes in the cluster have the same data.

The leader is responsible for replicating its log to the followers. Such log contains a series of commands that must be applied in the same order by all nodes in the cluster.

Only the leader can append entries to the log, and each entry is committed once it is replicated to a majority of the nodes.

In case the leader crashes it is important to ensure that a new one is elected, and that it has the same log as the previous one; in case it has incomplete log entries compared to other nodes, it fetches the missing entries from the followers.
The new leader ensures that the system can continue operating without losing any of the commands already agreed upon by the majority of nodes.

Log replication

Log replication is key for ensuring fault tolerance. It guarantees that a log entry is committed when the leader has replicated it to a majority of followers.
Once an entry is committed, it is applied to the state machine on the leader and followers.
This commitment ensures durability: the system guarantees that once a command is applied, it will not be lost, even in the face of failures.

8.4.3 Leader election

There are three states in Raft:

- ◇ **Leader** - The leader is responsible for managing the replication of the log.
- ◇ **Follower** - The follower replicates the leader's log.
- ◇ **Candidate** - The candidate is a node that is trying to become the leader.

Becoming the leader

1. When the current leader fails (e.g. crashes or becomes unreachable), a new leader is elected.
2. Follower nodes can become candidates and initiate electionss if they don't receive a heartbeat from the leader within a specified time (election timeout).
3. Leader election is the foundation of Raft's fault tolerance.

Once a leader is elected, the leader appends new log entries and replicates them to all followers. A log entry is considered committed when it is replicated to a majority of nodes. Followers always accept entries from the current leader to maintain consistency.

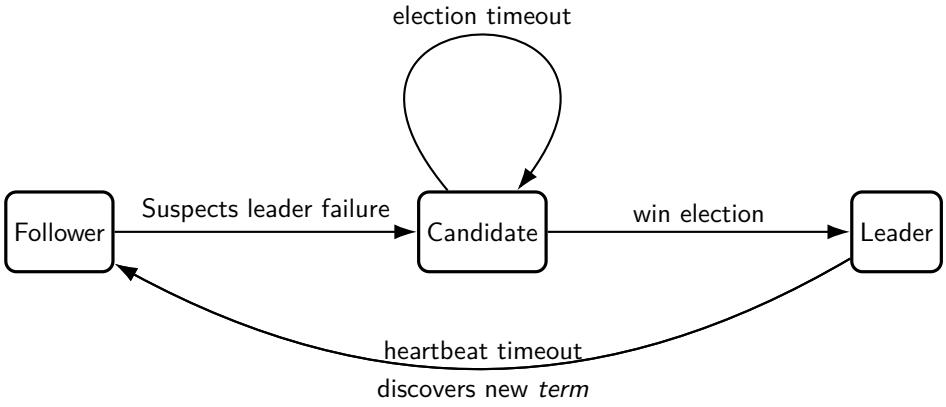


Figure 8.1: State diagram

8.4.3.1 Ensuring Safety and Liveness

- ◇ Safety
  Raft ensures that the logs of all nodes are consistent. No two nodes will decide on different values, maintaining the consistency property.

- ◇ Liveness
  Raft guarantees that as long as a majority of nodes are functioning, the system will continue to make progress (Liveness). Raft avoids the Liveness issues that Paxos sometimes faces, especially with leader election and split-brain situations.

Raft separates the consensus process into distinct stages: leader election, log replication, and commitment. The leader-driven structure is intuitive and provides clear responsibilities.
Compared to Paxos, its safety is simpler to reason about since all decisions are centralized around the leader.

8.4.4 Raft Election and Voting process

8.4.4.1 Election

- ◇ Election Term and Leader State:
  - *term* number represents a logical clock that increases monotonically whenever a new election is initiated.
  - prevent conflicts during leader election and ensures a unique identifier for each term.
- ◇ Follower State:
  - When a node starts or after a leader election, it begins in the follower state.
- ◇ Candidate state:
  - If a follower does not receive any communication from the leader within a given timeout, it transitions to the candidate state and starts a new election term.
  - The candidate increments its current term number and requests votes from other nodes.

8.4.4.2 Voting process

- ◇ RequestVote:
  - When a candidate transitions to the candidate state, it sends RequestVote to all other nodes in the cluster.
  - RequestVote includes the candidate's term number, its own last log index and term, and its eligibility for becoming the leader.
- ◇ Voting process:
  - When a follower receives a RequestVote, it checks if the candidate's term number is higher than its own.
  - If so, it updates its current term and resets its election timeout, acknowledging the leader about the election
- ◇ Candidate state:
  - if a candidate receives votes from a majority of the nodes in the cluster, it becomes the new leader for the current term.
  - If a candidate does not receive enough votes, it returns to the follower state and waits for the next election timeout to start a new election term
- ◇ Leader state:
  - Upon becoming the leader, the node starts sending AppendEntries to replicate log entries to the followers.
  - The leader's election timeout is reset periodically to prevent unnecessary re-elections while it continues to serve as the leader.
- ◇ Heartbeats:
  - The leader regularly sends AppendEntries RPCs with empty log entries (heartbeats) to maintain its authority and prevent other nodes from starting new elections

8.5 Consensus Takeaways

- ◇ Consensus Problem:
  - Distributed consensus is the challenge of getting a set of distributed nodes (processes) to agree on a single value, despite failures and message delays.
- ◇ Importance of Consensus:
  - Critical for ensuring data consistency and fault tolerance in distributed systems.
  - Powers key systems like databases, replicated state machines, and blockchain technologies.
- ◇ Key Properties of Consensus:
  - Safety: The algorithm ensures that nodes never decide on conflicting values. Once a value is decided, it remains fixed.
  - Liveness: The system eventually reaches a decision, despite failures or delays, ensuring progress.
- ◇ Challenges in Asynchronous Systems:
  - In asynchronous systems, where nodes can experience delays or crashes, achieving both safety and liveness simultaneously is challenging.

  - The FLP Impossibility Theorem proves that no deterministic consensus algorithm can guarantee both safety and liveness in fully asynchronous systems with even a single faulty node.
- ◇ Paxos:
  - A widely used consensus algorithm focused on safety. It guarantees that nodes will agree on a value, but it may fail to make progress in some situations, hence not guaranteeing liveness in every case.
- ◇ Raft:
  - Raft is a consensus algorithm designed for understandability and practicality. It divides the consensus process into two dist inct phases: leader election and log replication.
  - Raft guarantees safety by ensuring that only one leader is elected, and achieves liveness under normal conditions, though like Paxos, it may struggle under certain network partitioning scenarios.
  - Its straightforward structure and clear separation of roles make it widely adopted in distributed systems that prioritize clarity and maintainability.
- ◇ Consensus Mechanisms in Practice:
  - Consensus algorithms like Paxos and Raft are the backbone of many distributed systems to ensure reliability, consistency, and fault tolerance, from databases to large-scale cloud systems.

Part III

Spreading data around

Chapter 9

CAP Theorem

Definition 9.1 (CAP Theorem) *It is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees:*

- ◇ ***Consistency***
  Every read receives the most recent write or an error
- ◇ ***Availability***
  Every request receives a (non-error) response, without the guarantee that contains the most recent write
- ◇ ***Partition tolerance***
  The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

Also known as Brewer's theorem, the CAP theorem was formulated by Eric Brewer in 2000.

When a network partition failure happens should we decide to either
- ◇ Cancel the operation and thus decrease the availability but ensure consistency
- ◇ Proceed with the operation and thus provide availability but risk inconsistency

" The CAP theorem implies that in the presence of a network partition, one has to choose between consistency and availability. " — Prof. Dazzi

9.1 Trade off

CAP theorem describes the trade-offs involved in distributed systems: a proper understanding of CAP theorem is essential to make decisions about the future of distributed system design. Misunderstanding can lead to erroneous or inappropriate design choices.

The CAP theorem suggests there are three kinds of distributed systems:
- ◇ **CP** ← Consistent when Partitioned
- ◇ **AP** ← Available when Partitioned
- ◇ ~~**CA** ← Consistent and Available Not Partitioned~~
  This cannot exist in a distributed system; network partitions are inevitable.

Both AP and CP systems can offer a degree of consistency, and availability, as well as partition tolerance.

9.2 Consistency

AP systems provide best effort consistency, meaning that they will eventually become consistent, but not necessarily at the time of the request.
Examples are web caching and DNS.

- ◇ **Strong** Consistency - all users see the same data at the same time. Any read operation returns the result of the most recent write operation (linearizability). This is used for financial transactions.
- ◇ **Weak** Consistency - Reads may return stale data. Common in systems that prioritize high availability and low latency, with minimal constraints on ordering and visibility.

- ◇ **Eventual** Consistency - If no new updates are made to object, eventually all accesses will return the last updated value
  - *Causal* consistency - Processes that have causal relationship will see consistent data. Preserves the causality of operations by ensuring that causally related updates are seen in the same order by all nodes.
  - *Read-your-write* consistency - A process always accesses the data item after it's update operation and never sees an older value.
  - *Session* consistency - As long as session exists, system guarantees read-your- write consistency. Guarantees do not overlap sessions.
  - *Monotonic read* consistency - If a process has seen a particular value of data item, any subsequent processes will never return any previous values.
  - *Monotonic write* consistency - The system guarantees to serialize the writes by the same process. These two properties may be combined

9.2.1 Example Scenario - Dynamic Tradeoff - Dynamic Reservation

In an airline reservation system when most of seats are available it is ok to rely on somewhat out-of-date data, availability is more critical.

When the plane is close to be filled instead, it needs more accurate data to ensure the plane is not overbooked, consistency becomes more critical.

Neither strong consistency nor guaranteed availability, but it may significantly increase the tolerance of network disruption.

9.3 Partitioning

- ◇ **Data** Partitioning - Different data may require different consistency and availability.
  Example: A *Shopping cart* requires high availability, responsive, can sometimes suffer anomalies, while *Product information* need to be available, slight variations in inventory are sufferable. Checkout, billing, shipping records must be consistent.
- ◇ **Operational** Partitioning - Different operations may require different consistency and availability.
  Example: A user login requires high availability, responsive, can sometimes suffer anomalies, while a transaction requires consistency.
  In general,
  - *Reads* - high availability; e.g. "query"
  - *Writes* - high consistency, lock when writing; e.g. "purchase"
- ◇ **Functional** Partitioning - System consists of sub-services. Different sub-services provide different balances.
- ◇ **User** Partitioning - Try to keep related data close together to assure better performance. Besides, having geographical distribution of users may help to decide the trade-off, since users querying servers closer to them may have lower latency.
- ◇ **Hierarchical** Partitioning - Different levels of consistency and availability may be provided at different levels of the system architecture.
  Example: A multi-tier architecture may provide strong consistency at the database level, while allowing for eventual consistency at the application level to improve responsiveness. Local servers (better connected) guarantee more consistency and availability. Global servers has more partition and relax one of the requirement.

9.4 PACELC Theorem

Definition 9.2 (PACELC Theorem) *The PACELC theorem is an extension of the CAP theorem that considers latency and consistency in distributed systems.*

- ◇ ***P*** *- Partition tolerance*
- ◇ ***A*** *- Availability*
- ◇ ***C*** *- Consistency*
- ◇ ***E*** *- Else*
- ◇ ***L*** *- Latency*
- ◇ ***C*** *- Consistency*

PACELC in other terms

- ◇ If there is a partition (P), how does the system trade off availability and consistency (A and C);
- ◇ else (E), when the system is running normally in the absence of partitions, how does the system trade off latency (L) and consistency (C)?

In other terms, PACELC states that in presence of a network partition, a distributed system must choose between consistency and availability (as per CAP theorem). However, even when there is no partition, the system must still make a trade-off between latency and consistency.

There are some systems which adopt various PACELC flavors:

- ◇ *Give up*/**PA/EL** - Dynamo, Cassandra: Give up consistency for availability and lower latency.
- ◇ *Refuse*/**PC/EC** - BigTable, HBase, VoltDB, H-Store: Refuse to give up (compromise) consistency for availability and lower latency.
- ◇ *Give up(?)*/**PA/EC** - MongoDB: Give up consistency when a partition happens, and keep consistency in normal operations, when there is no partition.
- ◇ *Keep*/**PC/EL** - Yahoo! PNUTS: Keep consistency when a partition happens, and give up consistency for lower latency in normal operations, when there is no partition.

9.5 Takeaways

- ◇ Consistency Models Define Expectations: Each model sets specific rules on how data should behave across distributed nodes, defining when data changes become visible across the system.
- ◇ Trade-offs Are Inevitable: No single model can satisfy all consistency, availability, and partition tolerance (CAP theorem), so systems often need to choose between strong consistency and high availability, especially during network partitions.
- ◇ Choosing the Right Model Depends on Use Case: Strong consistency is essential for scenarios demanding strict accuracy (e.g., banking), while eventual or weak consistency is more practical for applications prioritizing availability and low latency (e.g., social media).
- ◇ Causal and Sequential Consistency Balance Practicality and Order: These models allow for some flexibility while still preserving the order of events, making them useful in collaborative applications or messaging systems.
- ◇ Eventual Consistency is the Backbone of High Scalability: Systems prioritizing high availability and partition tolerance, like DNS or e-commerce platforms, often rely on eventual consistency, accepting slight delays in consistency for better performance.
- ◇ CAP and PACELC Guide Model Selection: CAP helps in understanding consistency trade-offs during partitions, while PACELC addresses trade-offs between latency and consistency even in the absence of partitions, providing a more nuanced framework for design decisions.
- ◇ Practical Knowledge Matters: Understanding how each model impacts system performance, reliability, and usability is crucial for system architects and developers to make informed decisions, especially in large-scale distributed environments.

Chapter 10

Replication

Definition 10.1 (Replication) *Replication is the process of sharing information so as to ensure consistency between redundant resources, such as software or hardware components, to improve reliability, fault-tolerance, or accessibility.*

The aims of replication are reducing latency, increasing availability, and scaling read throughput. However replication clearly poses some potential problems when handling changes to replicated data. The possible approaches to replication are:

- ◇ single-leader
- ◇ multi-leader
- ◇ leaderless replication

10.1 Leaders and followers

Each node may store a copy of the database, in this case it is called a **replica**, so, ensuring consistency among replicas is crucial.

With leader based replication, one replica is designated as the *leader* who receives write requests by clients, and is responsible for writing new data to its local storage. Later on, followers update their local copy based on the leader's replication log.
When a new follower has to be set up, a standard file copy may result in inconsistencies due to ongoing writes, so a **snapshot** mechanism must be used to ensure data integrity.

10.1.1 Synchronous vs Asynchronous Replication

- ◇ **Synchronous** - The leader waits for the followers to acknowledge the write before acknowledging the write itself.
  - Ensures that followers have an up-to-date copy of the data.
  - The disadvantage is that the system may become unavailable if a follower is slow to respond.
- ◇ **Asynchronous** - The leader does **not** wait for the followers to acknowledge the write before acknowledging the write itself.
  - Follower may fall behind the leader.
  - The advantage is that the system continues processing even if a follower is slow to respond.
  - The disadvantage is that the system may lose data if the leader fails before the followers have caught up, so there's the need for methods to ensure data integrity.
- ◇ **Semi-synchronous** - The leader waits for a **quorum** of followers (at least one follower, typically always the same and placed in another location) to acknowledge the write before acknowledging the write itself.
  - Ensures that at least one follower has an up-to-date copy of the data.
  - Typically there is only one node fully synchronous, while others are asynchronous, and it is placed far away from the leader.

10.2 Failure

- ◇ **Node** failures - Nodes can fail due to faults or planned maintenance. The goal is to minimize impact and keep the system running.

- ◇ **Follower** failure: Catch up recovery - Followers keep a log of data changes, but can catch up with the leader if they fall behind.
- ◇ **Leader** failure: Failover - Failover process involves promoting —hence reconfiguring— a follower to leader. Failover can be manual or automatic
- ◇ Challenges in **Failover** - Asynchronous replication may lead to data loss, as stated before.

10.2.1 Implementing Replication Logs

The replication log is the mechanism by which changes are propagated from the leader to the followers. Different approaches exist, each with specific trade-offs in terms of flexibility, consistency guarantees, and implementation complexity.

The choice of replication strategy depends on the database architecture, the need for cross-version compatibility, and the desired level of abstraction between the storage engine and the replication mechanism.

- ◇ **Statement-based** replication - The leader logs and sends the actual SQL statements (e.g., `INSERT`, `UPDATE`, `DELETE`) to followers, which then execute them.
  Example: Leader executes `UPDATE users SET balance = balance + 100 WHERE id = 42` and sends this exact statement to followers.
  Problems:
  - Non-deterministic functions like `NOW()`, `RAND()` produce different results on each replica
  - Auto-incrementing columns may generate different values
  - Statements with side effects (triggers, stored procedures) may behave differently

  This was used in VoltDB and MySQL before version 5.1.
- ◇ **Write-ahead log (WAL)** shipping - The leader ships its low-level write-ahead log (containing byte-level changes to disk blocks) to followers, which replay the exact same disk modifications.
  Example: PostgreSQL replicates the WAL that describes which bytes changed in which disk pages.
  Problems:
  - Tightly coupled to the storage engine internals
  - Leader and followers must run the **same database version** and architecture
  - Zero-downtime upgrades become difficult (cannot upgrade followers first)
- ◇ **Logical (row-based) log** replication - Replicates a logical representation of changes at the row level, decoupled from the storage engine internals.
  Example: For an inserted row, the log contains: `{table: "users", operation: "INSERT", values: {id: 42, name: "Alice"}}`. For updates, it includes old and new values.
  Advantages:
  - Storage-engine independent: allows different database versions or even different database systems
  - Easier to parse by external applications (e.g., data warehouses, caches)
  - Supports backward compatibility

  Disadvantages:
  - Requires additional implementation effort and expertise
  - Need to maintain the logical log format separately from storage internals

  Used by MySQL's binlog (in row-based mode), PostgreSQL's logical decoding.
- ◇ **Trigger-based** replication - Uses database triggers to capture changes and write them to a separate replication log table, which is then read by an external process that applies changes to followers.
  Example: A trigger on `INSERT/UPDATE/DELETE` writes change records to a `replication_log` table; a replication daemon reads this table and applies changes to other databases.
  Use cases:
  - Flexible replication (e.g., replicate subset of data, transform data during replication)
  - Replication to different database systems
  - Custom conflict resolution logic

  Disadvantages:
  - Higher overhead and performance impact
  - More prone to bugs due to application-level code
  - Trigger logic can become complex to maintain

  Used by Oracle GoldenGate, Databus for Oracle, Bucardo for PostgreSQL.

10.3 Eventual Consistency

The eventual consistency model implies a temporary state where the followers lag behind the leader, but eventually they will catch up. Hence there is some "**replication-lag**" …

However, many modern applications require heavy reading workload, but light writing workload, so eventual consis-

tency is a good trade-off. Besides it is possible to increase capacity for read- only requests by adding more followers. Removes load from the leader. Allows read requests to be served by nearby replicas

10.3.1 Read-after-write consistency

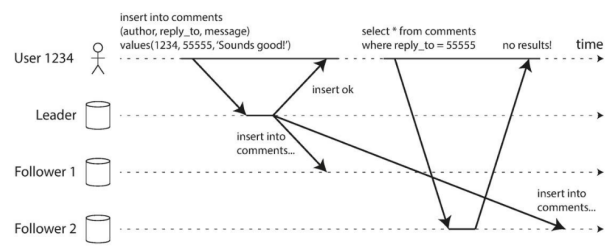


Figure 10.1: Read-after-write consistency time schema

In asynchronous replication it is not trivial to ensure *read-your-write* consistency, i.e. if a client writes a value to the leader, it should be able to read it. In other words, a user may think their data is lost, because they cannot see it, but it is actually there, just not yet replicated to the follower.

Read-after-write consistency guarantees that users see their own updates, but not necessarily the updates of others. It may be implemented by reading user-modified data from the leader.

10.3.2 Monotonic Reads

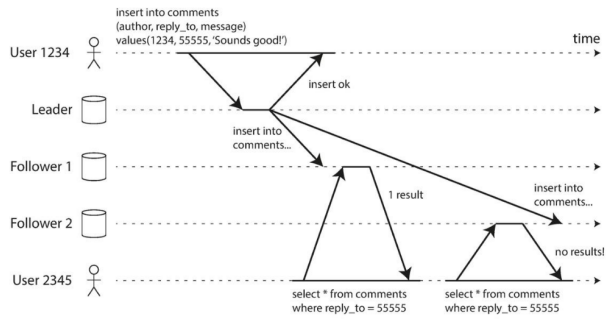


Figure 10.2: Monotonic reads time schema

Monotonic reads consistency guarantees that if a user has read the value of a key, it will not read an older value of the same key in the future. It is stronger than eventual consistency, but weaker than strong consistency.

In asynchronous replication may see data moving backward in time when reading from different replicas with varying lag.
Monotonic reads can be achieved by ensuring that reads are always performed on the same replica.

10.3.3 Consistent Prefix Reads w/different DBs

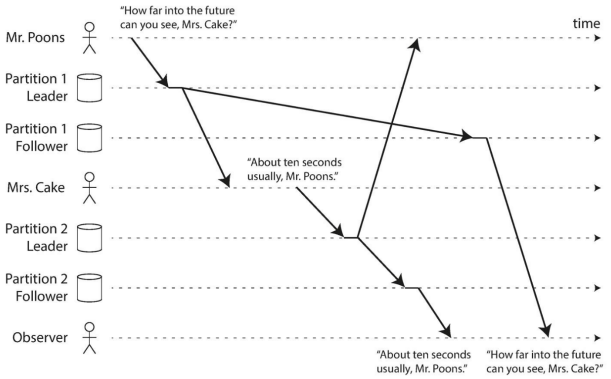


Figure 10.3: Consistent prefix reads with different DBs

Consistent prefix reads consistency guarantees that if a sequence of writes happens in a certain order, then a read will see those writes in the same order.

Suppose that A asks B something, and C is listening, but hears first the answers and later the question: weird, ain't it?
This may happen due to replication lag. We want to prevent casuality violations.

10.4 Multi-Leader Replication

With a single-leader configuration, the leader is a bottleneck, and if it fails, the system elects a new leader (perhaps there may be a small downtime). Especially in geographically distributed systems, having a single leader may introduce high latency for clients far away from the leader.

With multi-leader replication, there are multiple leaders, and each leader can accept writes. This is useful for geographically distributed systems, where each region has its own leader. The same applies for related datacenters, each with its own leader. On a lower level, we could also have multiple leaders for multiple partitions of the data, one per partition.

10.4.1 Collaborative Editing

Real-time collaborative editing is a use case for multi-leader replication.
Changes are instantly appied to the user's local replica and then asynchronously replicated to the server and other users.

Application must obtain a lock on the document before editing, as with single leader replication with transactions (?).
The unit of change may be small, as a single keystroke.

When two users edit the same part of the document, they get local updates, but when the system asynchronously synchronizes the replicas, a conflict occurs and needs resolution.

- ◇ User 1 changes title from A to B
- ◇ User 2 changes title from A to C
- ◇ Both changes are applied locally
- ◇ Conflict detected during asynchronous replication

With single-leader replication, the second writer blocks or aborts if first write is incomplete, having the user to retry the input, ultimately avoiding the issue.
With multi-leader replication, the system must resolve the conflict, and the user must be notified of the resolution.

10.4.2 Conflict Avoidance

Strategies

- ◇ Routing writes towards the same leader prevents conflicts.
- ◇ Routing user requests towards the same datacenter allows to use only the leader of that datacenter; furthermore, geographical locality may be applied, to serve a user with the closest datacenter.
  Conflict resolution may happen *on write*, *on read*, or at *row/document* level.

10.4.3 Topologies

- ◇ **Circular** - each node receives writes and forwards writes
- ◇ **Star topology** - root node forwards writes to all other nodes
- ◇ **All-to-all** - every leader sends writes to every other leader

Note that this is not hardware (neither virtual) network, but a logical network of replication, made up by how we manage the information flow.

In whatever topology it is important to **monitor staleness**, which is monitoring obsolete replicas, i.e. how far behind a follower is from the leader. Some metrics must be kept.

There is also **leaderless replication**, where there is no designated leader, and any node can accept writes. This is useful for systems that require high availability and fault tolerance, as there is no single point of failure. Handling Node Outages is easier with leaderless replication, as any node can accept writes, and there is no need to elect a new leader. Write operations are considered successful once a quorum of nodes acknowledges the write.
Stale values may occur when a node comes back online. These are detected by reads with a quorum, which ensures that the most recent value is returned.

However, it introduces challenges in ensuring consistency among replicas, as concurrent writes may lead to conflicts that need to be resolved.

10.4.4 Concurrent Writes

Several clients can write to the same key simultaneously, and the order of writes may differ on different leaders.
So…"who wins?"

- ◇ **Last write wins** - The last write is the one that is kept.
- ◇ **Most recent timestamp wins** - The write with the most recent timestamp is the one that is kept.
- ◇ **Merge values** - The values are merged together, based on a *happens-before* relationship.

However, especially for geographically spread databases, it is not trivial to determine the most recent timestamp, because clocks may be unsynchronized, and also the network latency may vary. Remember also that the light speed is finite, so it takes time for a signal to travel from one point to another.
For this reason the *happens-before* relationship is used, which is a partial order on events, which reflects the order in which events have occurred.

10.5 Key Takeaways

Replication is a fundamental technique in distributed systems that ensures data availability even when nodes fail, allowing systems to continue operating and serving requests. By maintaining multiple copies of data across different locations, replication also enables systems to function offline, which is particularly useful for applications that need to work without continuous network connectivity.

One of the primary benefits of replication is **latency reduction**: by placing replicas geographically closer to users, we can significantly enhance user experience through faster response times. Additionally, replication naturally **scales read operations** by distributing the workload across multiple replicas, making it essential for handling large volumes of read-heavy requests efficiently.

However, replication introduces significant challenges, particularly in managing concurrent writes and ensuring data consistency across replicas. The fundamental tension lies in balancing consistency, availability, and performance—a trade-off that manifests in different replication strategies (single-leader, multi-leader, leaderless) and consistency models (strong consistency, eventual consistency, causal consistency).

Understanding these consistency models is crucial for designing robust distributed systems. Each model offers different guarantees: *eventual consistency* prioritizes availability and performance but allows temporary inconsistencies, while *strong consistency* ensures all replicas are always synchronized but may sacrifice availability. Effective fault tolerance strategies, including proper failover mechanisms and conflict resolution policies, are necessary to maintain system integrity and reliability in the face of network partitions and node failures.

Chapter 11

Partitioning

Partitioning becomes necessary when dealing with **large data sets** and high **query throughput**. It is a technique that divides a large data set into smaller, more manageable parts. This is done to improve the performance of the system. Partitioning can be done in various ways, such as horizontal partitioning, vertical partitioning, and functional partitioning. In this chapter, we will discuss the different types of partitioning and their benefits.

"Partition" is the most common term, but it may vary depending on the technology:

- ◇ **Shard** in MongoDB, Elasticsearch, and Cassandra.
- ◇ **Region** in HBase.
- ◇ **VBucket** in Couchbase.
- ◇ **Tablet** in Bigtable.
- ◇ **VNode** in Riak and Cassandra.

11.1 Partitioning concepts

First, partitions must be **defined**, in the sense that each piece of data must be assigned to a partition. This can be done in various ways, such as hashing, range partitioning, or list partitioning.

Then, each partition should have its own **characteristics**, i.e. should support a known set of operations, since it acts as a small per se database.

Clearly, different partitions may reside on different nodes, enabling scalability.

11.1.1 Combining partitioning with replication

Partitioning can be combined with replication to improve the performance and reliability of the system. Replication is the process of creating multiple copies of the data and storing them on different nodes. This ensures that the data is available even if one of the nodes fails. By combining partitioning with replication, you can achieve high availability and fault tolerance.

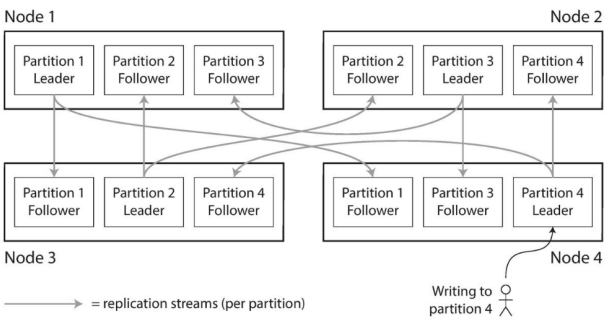


Figure 11.1: Leader-follower model

This can be achieved in various ways:

1. Node Storage of Multiple Partitions
   - ◇ Nodes can store more than one partition, and each partition can be stored on multiple nodes.
2. **Leader** and **Follower** assignment
   - ◇ Each node stores a partition, and one of the nodes is designated as the leader. The leader is responsible

for handling all write operations, while the followers, assigned to different nodes, replicate the data from the leader.

3. Replication and Partitioning
   - ◇ The two techniques are enforced independently.

The Goal of partitioning is to spread data and query load **evenly** among the nodes. Unfair partitioning can lead to **hot spots**, where some nodes are overloaded while others are underutilized.
Randomizing the partitioning function can help to avoid hot spots, but it can also make it difficult to locate data, requiring to query all nodes to get a value.

11.1.2 Key-Range Partitioning

A first improvement might be to provide a range-key assignment allowing to locate data, as it happens in libraries, where books are ordered by author or title.
In this way you know the boundaries of where to search. This is very easy to implement and to understand, however, it is *not* optimal: data may **not** be evenly distributed among the possible keys.

Consider partitioning a student database by student ID. If student IDs are assigned sequentially, then all students enrolled in the same year will have similar IDs. This can lead to uneven distribution of data across partitions, as students from the same year may be assigned to the same partition, while other partitions may be underutilized.

11.2 Avoiding Hot spots

So with key-range partitioning, the problem is that some keys may be more popular than others due to access patterns, leading to hot spots.
In, for instance, a sensor database, all today's writes would end up in the same "today's partition", while the rest of the partitions would be idle.

A solution might be to change-key structure, for instance by adding a prefix to the key, such as the sensor ID, to distribute the data more evenly.

11.2.1 Hash partitioning

A better solution is to use **hash partitioning**, where a hash function is used to map keys to partitions.
However we must be careful because **inconsistent** hashing can lead to hot spots, as the hash function may not distribute keys evenly. In general, however, a good hash function will make skewed data uniformly distributed across partitions.

Note also that hash partitioning is very bad for range queries, as even similar data is spread randomly across the partitions.
In Cassandra, a **compound primary key** is used, where the first part is hashed to determine the partition, while the second part is an index used to order data within the partition, allowing for efficient range queries within a partition.
Range queries on the first column (the hashed one) are not possible, as data is spread randomly across partitions, so all partitions must be queried in such case.

11.2.1.1 Secondary Indexes

Secondary indexes are a way to avoid hot spots in hash partitioning.
The idea is to create a separate index that maps the secondary key to the primary key, and then use the primary key to locate the data.
This way, the secondary key is hashed and distributed evenly across the partitions, while the primary key is used to locate the data.

- ◇ **Document**-based partitioning (local indexes)
  - Each listing has a unique document ID
  - Database is partitioned based on the document ID
  - Secondary indexes are on fields like color and make
  - Each partition maintains its own secondary indexes
  - *Reading* requires querying all partitions
- ◇ **Term**-based partitioning (global indexes)
  The "term" is essentially the value of the indexed field.
  - A Global index covers data in all partitions.
  - Partitioning is based on the term ID, which is not the primary key index.

- Example: Colors a-r in partition 1, s-z in partition 2.
- A single entry in secondary index may include records from all primary key partitions.
- Term determines the partition of the index
- The Term concept comes from full-text indexes

Global index allows clients to request specific partitions, and avoids scatter/gather over all partitions, so read operations are more efficient. However, write operations become more complex, as the write to a document may require updating multiple partitions, both the document partition and the index partition. Besides, each term in a document could be on a different partition, leading to multiple writes for a single document update.

11.2.1.1.1 Detailed Comparison: Document-Based vs Term-Based

Consider a database of used car listings with:

- ◇ **Primary key**: `document_id` (unique listing ID)
- ◇ **Secondary indexes**: `color`, `make` (brand), `model`
- ◇ Database partitioned by `document_id`

Document-Based Partitioning (Local Indexes) Each partition maintains its **own local secondary indexes** only for the documents it contains.

```
Partition 0 (document_id: 0-499):
|- Documents: {id:0, color:red}, {id:1, color:blue}, ...
`- Local Index:
   |- color:red -> [0, 23, 145, ...]
   |- color:blue -> [1, 67, 234, ...]
   `- make:toyota -> [12, 89, ...]

Partition 1 (document_id: 500-999):
|- Documents: {id:500, color:red}, {id:501, color:green}, ...
`- Local Index:
   |- color:red -> [500, 678, 891, ...]
   |- color:green -> [501, 602, ...]
   `- make:toyota -> [534, 712, ...]
```

Query example (scatter/gather):

```
Query: "SELECT * WHERE color='red'"
  |-> Partition 0: [0, 23, 145]
  |-> Partition 1: [500, 678, 891]
  |-> Partition 2: [1001, 1234]
  `-> Partition 3: [1500, 1789]
       |
       v
  Merge results: [0, 23, 145, 500, 678, 891, 1001, 1234, 1500, 1789]
```

✓**Write**: Simple - updates only the partition containing the document
✗**Read**: Complex - must query **all** partitions and merge results

Term-Based Partitioning (Global Indexes) A single **global index** for each secondary field, partitioned by term (field value).

```
Global Index for "color" (partitioned by term):
Partition A (colors: a-m):
  |- blue -> [doc:1, doc:67, doc:234, doc:501, ...]
  |- green -> [doc:501, doc:602, ...]
  `- black -> [doc:34, doc:456, ...]

Partition B (colors: n-z):
  |- red -> [doc:0, doc:23, doc:145, doc:500, doc:678, ...]
  |- white -> [doc:89, doc:345, ...]
  `- yellow -> [doc:123, doc:789, ...]
```

Note: Document IDs in indexes can come from **any partition** of the main database!

Query and write examples:

```
Query: "SELECT * WHERE color='red'"
  `-> Global Index Partition B (red falls in n-z)
       `-> Returns: [doc:0, doc:23, doc:145, doc:500, doc:678, ...]

INSERT {id:1500, color:'red', make:'toyota'}
```

```
|-> Database Partition 3 (for doc_id=1500)
|-> Global Index "color" Partition B (for term='red')
`-> Global Index "make" Partition X (for term='toyota')
```

✓**Read**: Efficient - query goes to a single index partition
✗**Write**: Complex - may update multiple partitions (document + multiple index partitions)

Aspect	Document-Based	Term-Based
Read	Slow (scatter/gather)	Fast (single partition)
Write	Simple (single partition)	Complex (multiple partitions)
Consistency	Easier (local)	Harder (distributed)
Complexity	Low	High
Best for	Write-heavy workload	Read-heavy workload

Table 11.1: Document-Based vs Term-Based Secondary Index Partitioning

Trade-offs Summary **Use Document-Based when**: Writes are more frequent than reads, secondary index queries are rare. Example: logging system.

Use Term-Based when: Reads are more frequent than writes, secondary index queries are common. Example: e-commerce with searches by color/brand.

Analogy

- ◇ **Document-Based**: Like having a library in each city with its own catalog. To find all "Hemingway" books, you must check every city's library catalog.
- ◇ **Term-Based**: Like having a national catalog partitioned by author (A-M, N-Z). To find "Hemingway" go directly to the H partition, but when adding a new book you must update both the local library and the national catalog.

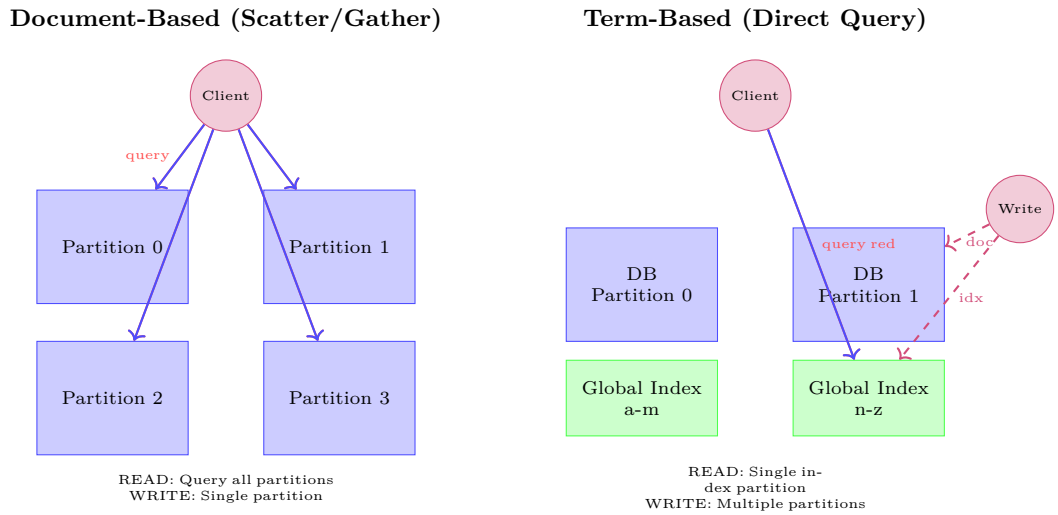


Figure 11.1: Visual comparison of Document-Based vs Term-Based partitioning query patterns

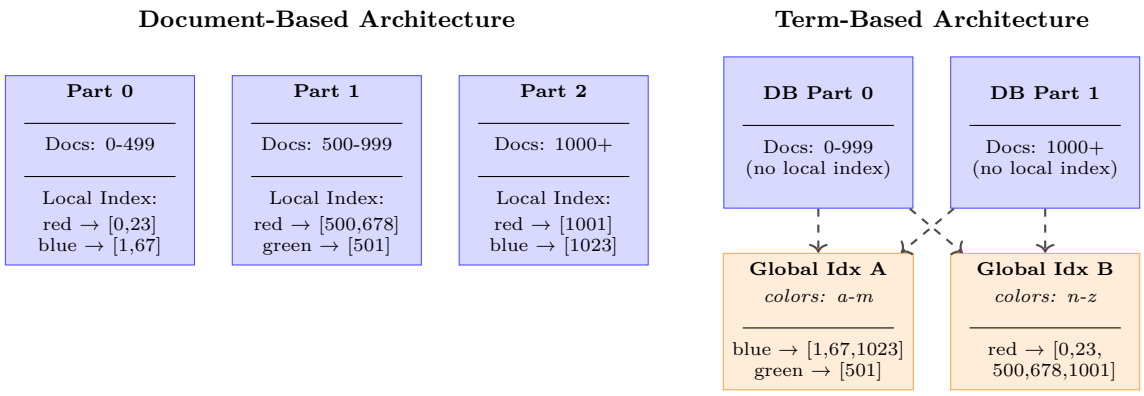


Figure 11.2: Data distribution in Document-Based vs Term-Based secondary index partitioning

Visual Comparison

11.3 Rebalancing

Rebalancing is the process of moving data between partitions to ensure that the data is evenly distributed among the nodes. This is necessary when the data distribution changes, for example, when new nodes are added to the system or when the data distribution becomes uneven due to hot spots. Rebalancing can be done in various ways, such as automatic rebalancing, manual rebalancing, and dynamic rebalancing.

Challenges in rebalancing

- ◇ Fair load distribution
- ◇ Continuous operation during rebalancing
- ◇ Minimizing data movement

- ◇ **Automatic** rebalancing - System automatically rebalances the data when necessary without any human intervention. May be unpredictable and expensive, but requires less maintanance.
- ◇ **Manual** rebalancing - Administrator manually rebalances the data when necessary. May be better since and admin may have a more *comprehensive* view of the distributed system, while a machine may be limited by network partitioning, discovery, etc.
- ◇ **Hybrid** approach - Combines automatic and manual rebalancing. The system automatically rebalances the data when necessary, but the administrator can override the system's decisions.

11.3.1 Techniques

A common approach to address rebalancing is the "**Hash mod N**" strategy, where each key is hashed and then the modulo operation with the number of nodes (N) determines the partition. However, this approach can lead to significant data movement when nodes are added or removed, as all keys need to be rehashed.

Optimizing rebalancing - Kademlia and P2P recalls

Assigning keys to nodes computing $h(key)$ mod #{nodes} $\rightarrow$ node is intuitive but leads to a tremendous amount of data movement when nodes are added or removed, since all modulo values must be recomputed.
A better solution is to use a **distributed hash table** (DHT) such as **Kademlia** or **Chord**, which allows to find the node responsible for a key in $O(\log n)$ steps.

Kademlia's approach: Both nodes and keys are hashed into the same fixed identifier space (e.g., 160-bit). Each key is stored on the node whose ID is *closest* to the key's hash, measured by XOR distance: $d(a, b) = a \oplus b$. When nodes join/leave, only keys in their immediate XOR neighborhood need to be transferred, minimizing data movement.

Example: If key hash is `101101` and node IDs are `101100`, `110001`, `101111`, the key goes to `101100` (XOR distance = 1, the smallest).

The key intuition exploited by both Kademlia and Chord is to **fix the granularity of the keyspace prior to partitioning**. By pre-defining a large identifier space (e.g., 2^{160} possible IDs), adding or removing nodes only affects a small fraction of keys, as the keyspace boundaries remain constant.

Another idea is to have many **fixed partitions**, more than nodes, and assigning several (like 1000) partitions to each node. When a node is added it steals partitions from other nodes, and when a node is removed its partitions are reassigned to other nodes. This way, only a small portion of the data needs to be moved during rebalancing. Partitions are never split or merged, only reassigned. Clearly, we can exploit more powerful nodes to hold more partitions.
The issue with this is that incorrect boundaries can lead to hot spots, so careful monitoring is required.

Dynamic Partitioning is another approach, where partitions can be split or merged based on the data distribution. This allows for more flexibility in rebalancing, as partitions can be adjusted to better fit the data distribution. However, this approach can be more complex to implement and manage.
Partitions are split when they exceed a configured size, and merged when some other gets deleted.
Each partition assigned to one node.

Proportional Partitioning may be enforced in different fashions:

- ◇ Dynamic Partitioning - Number of partitions varies is proportional to dataset size. Splitting and merging processes maintain partition size.
- ◇ Fixed Number of Partitions - the number of partitions is fixed and proportional to dataset size, it is independent of the number of nodes.
- ◇ Proportional Assignment to Nodes - Number of partitions proportional to number of nodes

This allows for better utilization of resources, as more powerful nodes can handle more partitions. However, this approach requires careful monitoring and management to ensure that the data is evenly distributed among the nodes.

11.3.2 Routing and Querying

Routing is the process of determining which partition to send a query to. This is done by either forwarding requests to a routing tier, or by having partition-aware clients which know how the data is distributed. Alternatively, client and also query any node, if there is no routing tier nor they are partition-aware. Routing can be done in various ways, such as static routing, dynamic routing, and consistent hashing.
ZooKeeper is often used as a routing tier, as it provides a distributed configuration service that can be used to store the partitioning information.

Queries may be executed in parallel across multiple partitions, or they may be executed sequentially. Parallel execution is more efficient, but it requires more resources. Sequential execution is less efficient, but it is easier to implement.

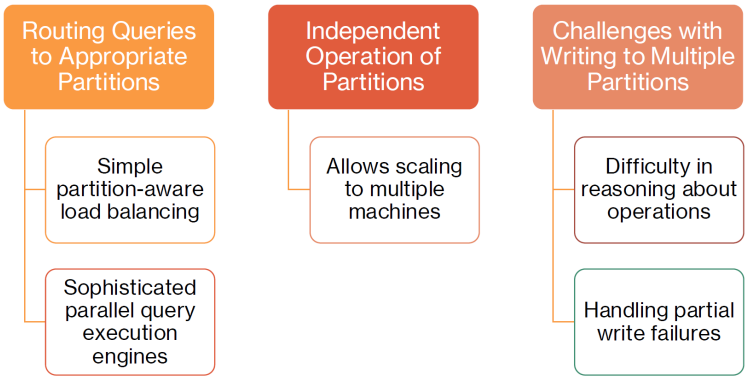


Figure 11.3: Routing

11.4 Takeaways

Data partitioning enhances scalability by distributing data across multiple nodes, and when combined with replication, it can improve the performance and reliability of the system. However, partitioning can introduce challenges such as hot spots and rebalancing. To address these challenges, you can use techniques such as hash partitioning, secondary indexes, and dynamic partitioning. By carefully designing the partitioning strategy, you can achieve high availability, fault tolerance, and scalability in your system.

Chapter 12

Transactions

Everything may fail at any time in a distributed system. Nodes may crash, networks may partition, messages may be lost or duplicated, clients may disconnect unexpectedly, race conditions may occur…In this scenario, it is crucial to maintain data consistency across multiple nodes. Transactions provide a framework for maintaining data consistency in the presence of failures.

Definition 12.1 (Transaction) *A **transaction** is a sequence of operations that are executed as a single unit of work. A transaction can consist of multiple operations, such as reads, writes, and updates. A transaction has to be **atomic**: all the operations in the transaction are executed successfully or none of them are.*
Besides, when a transaction is executed on some data, none of the other transactions can alter that data until the transaction is completed.

Transactions are clearly crucial in a distributed system, as they build a framework for allowing to maintain data consistency across multiple nodes.

Transactions come at a cost, since in a distributed system even a simple mechanism such as a mutex may become costly to implement.
Transactions hence provide all-or-nothing guarantees, simplifying **error handling**, since partial failures are not to be managed.

12.1 Deeper into DBs

Most SQL DBs support transactions. NoSQL databases typically do not support transactions, but some of them provide limited support for transactions on a single object.

12.1.1 Why NoSQL DBs do not support transactions?

NoSQL databases typically do not support transactions because they are designed to be highly scalable and distributed, and transactions can be difficult to implement in a distributed system. Transactions require coordination between multiple nodes, which can be costly and can lead to performance issues.

NoSQL databases typically prioritize scalability and performance over strong consistency guarantees, which are provided by transactions. However, some NoSQL databases provide limited support for transactions on a single object, which can be useful for certain applications.

12.2 ACID Properties

- **Atomicity**: all operations in a transaction are executed successfully or none of them are.
- **Consistency**: the database is in a consistent state before and after the transaction.
- **Isolation**: the transaction is executed in isolation from other transactions.
  In other words, transactions appear to run serially
- **Durability**: once a transaction is committed, the changes are permanent.
  Perfect durability is unattainable

These were coined by *Jim Gray* in 1983.

12.2.1 Atomicity

Ensures no other thread can see a half-finished operation. System is either in the state before or after the operation. It is not about concurrency, but about handling faults during transactions. Ensures all writes in a transaction are discarded if a fault occurs. Prevents partial changes and ensures data integrity.

12.2.2 Consistency

Involves statements about data (*invariants*) that must always be true.

Example: In accounting, credits and debits must always balance

Application must define transactions correctly to preserve consistency: database cannot guarantee consistency if application writes bad data.

Database can check specific invariants using constraints, but these are application specific. Atomicity, isolation, and durability are *database properties*, but **Consistency**, instead, is an *application property* relying on atomicity and isolation.

12.2.3 Isolation

Multiple clients accessing the same records can cause race conditions. Isolation ensures transactions are isolated from each other, thus they appear to run serially. In fact it is formalized as serializability.

12.2.4 Durability

Durability is the property that ensures that once a transaction is committed, the changes are permanent.

This is a mess to ensure and in general is not possible to guarantee, but it is possible to make it very unlikely that a transaction is lost.

The issue is related to hardware faults, power outages, broken firmware,…
Given the absence of a one-size-fits-all solution, typically the approach involves a combination of writing to disk, replicating to remote machines, and backups.

12.2.5 Single-Object and Multi-Object Transactions

If a transaction involves multiple objects, it is a multi-object transaction, which causes **performance** and **deadlock** issues. Multi-object transactions require some way of determining which read and write operations belong to which transaction.

Single-object operations prevent lost updates when multiple clients write concurrently. These operations are not traditional transactions, since they do not provide atomicity across multiple objects. Terms like “lightweight transactions” or “ACID for single objects” can be misleading.

Transactions require aborting and rolling back changes if something goes wrong. Note however that retrying after permanent errors (e.g., constraint violations), or overload-related errors is pointless. It is also important to manage side effects, such as sending emails or updating caches, which may not be rolled back and must be performed only once the transaction is committed.

12.3 Avoiding Transactions

Transactions can safely run in parallel if they do not modify the same data. Concurrency issues arise when two transactions modify the same data simultaneously.
Race conditions can occur when one transaction reads data that is concurrently modified by another transaction.

Databases hide concurrency issues by providing transaction isolation. Isolation allows the application to pretend that no concurrency is happening. Serializable isolation guarantees that the transactions are executed in a serial order, which is the most strict isolation level.

Serializable isolation comes with a performance cost which makes it less common in practice. Weaker isolation levels are common, but are harder to understand and may lead to subtle bugs.

12.3.1 Read Committed

In *Read Committed* isolation level, a transaction can only read, and overwrite, committed data. This is the default isolation level in most databases. Databases prevent dirty reads using an approach that stores both the old committed value and the new value set by the current transaction.

Definition 12.2 (Dirty Read) *A **dirty read** occurs when a transaction reads data that has been written by another transaction that has not yet been committed.*

Definition 12.3 (Dirty Write) *A **dirty write** occurs when a transaction overwrites data that has been written by another transaction that has not yet been committed.*

In this model, **no** dirty reads nor dirty writes are allowed.

This model is more reliable than Non-transactional models, but still allows to avoid "paying" for transactions.

Definition 12.4 (Read Skew) *A **read skew** occurs when a transaction reads different temporal versions of the same data, leading to a temporary inconsistency, violating an invariant.*

Read skews are possible in *Read Committed* isolation level,

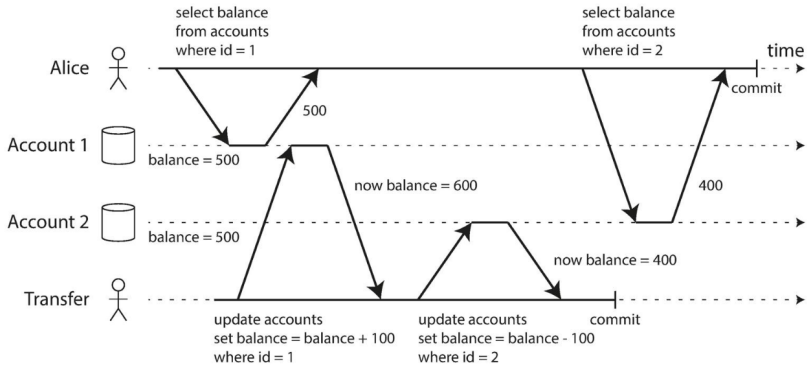
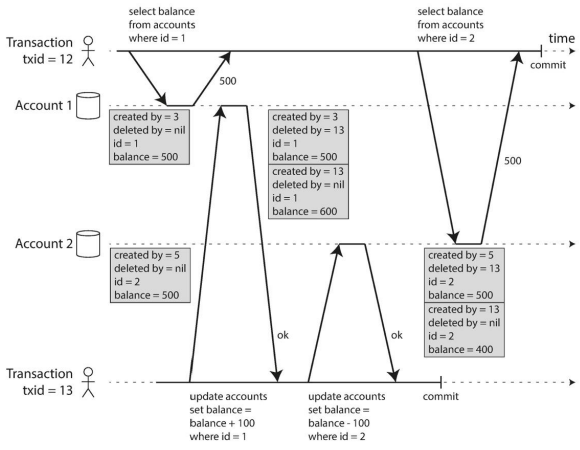


Figure 12.1: Read committed issue: Alice might think that her whole balance is 900, whilst it is 1000.

12.3.2 Snapshot Isolation



Snapshot Isolation is a more relaxed isolation level than serializable isolation, but it is still stricter than *Read Committed* isolation.

In *Snapshot Isolation*, a transaction reads a snapshot of the database at the beginning of the transaction and writes to the database at the end of the transaction.

The snapshot allows transactions to see all the data that was commited at the start of the transaction.

It is possible to use some techniques for locking the database, using locks or atomic operations. However it is costful

Figure 12.2: Snapshot isolation and MVCC

This model implements isolation by means of a lock to prevent dirty writes, but allows dirty reads (?), since reads do not acquire any locks. Readers never block writers, and writers never block readers.
Along with locks, MVCC (Multi-Version Concurrency Control) is used to provide each transaction with a snapshot of the database at the start of the transaction. In practice, each write creates a new version of the object, and the old version is kept around for readers that started before the write.

Compared to *Read Committed*, we could say that while *Read Committed* uses a separate snapshot for each **query**, *Snapshot Isolation* uses a single snapshot for the whole **transaction**.

12.4 Write Skew and Phantoms

There may happen write-write conflicts, causing **lost updates**. They occur during read-modify-write cycles, when two transactions read the same data, then update —possibly separate— objects based on the read data, and then commit, causing a conflict.

A key solution is to eliminate the read-modify-write cycle by using atomic operations. When these are insufficient, it is possible to use explicit locks to prevent concurrent access to the data.
Alternatively, the DB could also detect lost updates and in case abort one of the transactions.

Definition 12.5 (Write Skew) *A **write skew** occurs when two transactions read the same data, then update —possibly separate— objects based on the read data, and then commit, causing a conflict.*

There are some techniques to avoid write skew, such as using row locks, atomic operations, or specific constraints, but we did not discuss them pretty much.

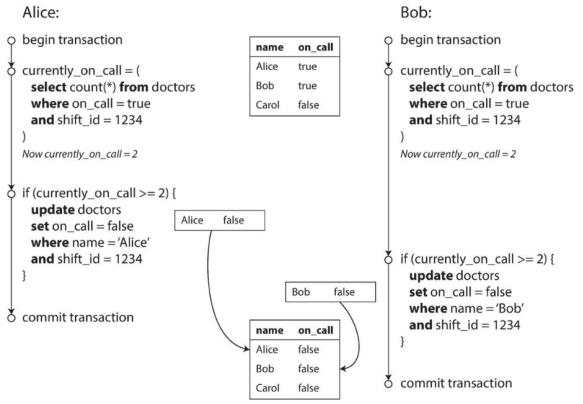


Figure 12.2: Both doctors read that `currently_on_call >= 2` and "leave the hospital", causing no doctor to be on call ☹.

Part IV

Frameworks and models for Distributed systems

Chapter 13

Big data

Distributed computing is the key to Big Data processoing, as it allows to *split* huge tasks into smaller ones, *parallely* execute them, *aggregate* the results and provide a final output.

Horizontally scaling is the key to Big Data processing, both from a Strong Scaling and Weak Scaling perspective (see section 1.2).

13.1 MapReduce Framework

Google was a pioneer in Big Data processing, and introduced the MapReduce framework in 2004. It is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster.

Hadoop

Initially BigData was built on top of Apache Hadoop, an open-source framework for distributed storage and processing of large datasets.

It allowed to used frameworks such as MapReduce to process data across large clusters.

Now it is already old-fashioned, and frameworks allowing some real-time analysis have taken over, such as the Lambda Architecture and the Kappa Architecture. Also Spark, basically a more efficient evolution of MapReduce, is widely used. Hadoop is still used as a distributed file system (HDFS), but not as a processing framework.

13.2 Infrastucture requirements

Not only powerful CPUs and GPUs are required, but also a **high-speed network** to allow for fast data transfer between nodes.
This may come in many flavours and architectures, such as **Ethernet**, **Infiniband**, or **Myrinet**. Often RDMA (Remote Direct Memory Access) is used to reduce latency and CPU overhead.

However, there is also the need for a **distributed File System**, to allow for a seamless and efficient way to manage and access data that is distributed across multiple machines or nodes in a network.

We want also to **abstract over distributed data**, hiding the complexity of data distribution, allowing users and applications to access files with a **unified view** as if they were stored on a single machine, ultimately providing also **location transparency**, i.e. decoupling the physical location of data from its logical location.

This does not refer to Fibre Channel or iSCSI, we are at a higher level of abstraction, where we want to provide a **file system interface** to the user, hiding the complexity of the underlying distributed storage.

There are different types of File Systems, each with its own characteristics and requirements:

- **HPC** FS - Latency is crucial here
  - GlusterFS (Lustre) - High performance, POSIX compliant. Allows for almost linear (weak) scalability

  - BeeGFS - Offers high data transfer speeds and flexible scalability. In general more flexible than Lustre, and still is Open Source.
- ◇ General Purpose FS
  - NFS - Based on UDP
  - Ceph - Mostly used in cloud and virtualized environments.
- ◇ **Big Data** FS - High throughput is crucial here
  - HDFS (Hadoop Distributed File System) - High throughput, fault-tolerant, designed for large files
    Its architectures mostly consists of *NameNodes* and *DataNodes*, where the former keeps metadata and the latter stores data and executes command.

    Hadoop, Based on HDFS, allows for distributed processing of large data sets across clusters of computers. Quite vintage.
- ◇ **P2P** FS - Decentralized, no central authority
  IPFS (InterPlanetary File System) - Content-addressable, versioned, peer-to-peer. It allows to create a distributed and permanent web, where files are identified by their hash, not by their location.

In the slides there is a brief summary on how Hadoop HDFS works.

Chapter 14

MapReduce

Instances of MapReduce are Hadoop and Spark. We will discuss them later. As stated in chapter 13, Google was a pioneer in Big Data processing, and introduced the MapReduce framework in 2004.

Distributed computing is the key to Big Data processing, as it allows to *split* huge tasks into smaller ones, *parallely* execute, and *aggregate* the results to provide a final output.
This is also known in data-analysis as *split-apply-combine*.

14.1 MapReduce Framework

`Map`

$\texttt{Map}(\langle key_a, value_1 \rangle) \rightarrow \texttt{list}(\langle key_b, value_2 \rangle)$

Various kind of processing on the input data may be done, typically filtering and sorting.

Note how the signature is slightly different from the typical map

`Reduce`

$\texttt{Reduce}(\langle key_b, \texttt{list}(value_2) \rangle) \rightarrow \texttt{list}((\langle key_c, value_3 \rangle))$

`Reduce` refers to the summarization, produced in an associative way, of the results produced by `Map`.

Each Reduce call typically produces either one key value pair or an empty return, though one call is allowed to return more than one key value pair.

1. **Map** - Each worker node applies the `Map` function to the local data, and writes the output to a temporary storage.
2. **Shuffle** - worker nodes redistribute data based on the output keys, such that all data belonging to one key is located on the same worker node.
3. **Reduce** - worker nodes now process each group of output data, per key, in parallel.

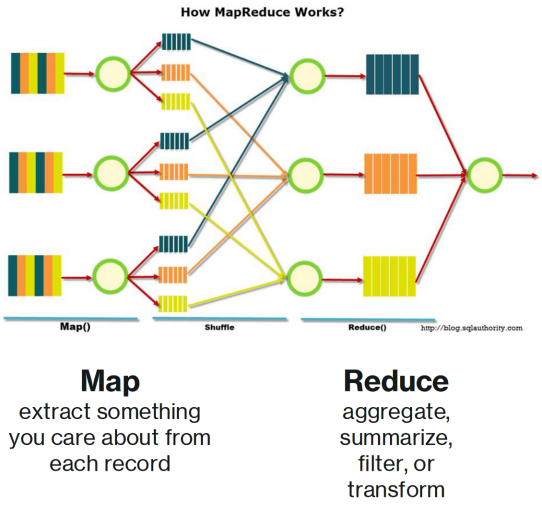


Figure 14.1: MapReduce overview

14.2 Instatiating MapReduce

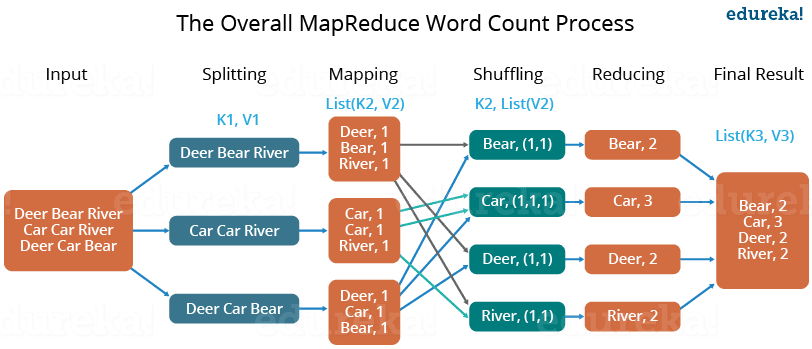


Figure 14.2: MapReduce WordCount example

- ◇ One `JobTracker` to decide to which client applications submit MapReduce jobs
- ◇ `JobTracker` pushes work to available `TaskTracker` nodes in the cluster keeping the work close to the data
- ◇ With a **rack-aware** file system the `JobTracker` knows which node contains the data.
  If the work cannot be hosted on the actual node where the data resides, priority is given to nodes in the same rack

TaskTracker failure: If a `TaskTracker` fails or times out, that part of the job is automatically rescheduled by the `JobTracker` to another available node. Since the input data is stored in HDFS (with replication, typically 3 copies), the failed task can be re-executed on a different node that has a replica of the required data blocks. This provides fault tolerance at the task level. The `JobTracker` detects failures through the heartbeat mechanism: if a `TaskTracker` stops sending heartbeats, it is marked as failed and its tasks are reassigned.

Data locality: Each `TaskTracker` preferably works on data stored locally (on the same node) or within the same rack, following the principle of **data locality**. The `JobTracker` exploits the rack-aware file system to assign tasks to nodes that already have the data, minimizing network transfers. If local execution is not possible, the scheduler tries to assign the task to a node in the same rack; only as a last resort will it schedule the task on a node in a different rack, incurring higher network overhead.

This data locality optimization is enabled by **HDFS** (Hadoop Distributed File System): the *NameNode* maintains metadata about which data blocks are stored on which *DataNode*s, and the `JobTracker` queries this information to make intelligent task placement decisions. Since worker nodes typically run both *DataNode* (storage) and `TaskTracker` (compute) roles, computation can be co-located with data, embodying the principle of *"moving computation to data rather than data to computation"*.

JobTracker failure recovery: If the `JobTracker` fails, the work is not lost because the job information and metadata are typically persisted to a reliable storage (such as HDFS). When the `JobTracker` restarts, it can recover the job state from this persistent storage and resume coordination. However, in classic Hadoop MapReduce (v1), `JobTracker` was a **single point of failure** (SPOF) —without automatic failover—, meaning that job recovery required manual intervention. In Hadoop 2.x (YARN), this limitation was addressed by introducing high availability for the `ResourceManager` (YARN's equivalent of `JobTracker`), with automatic failover to a standby instance, significantly improving fault tolerance.

A heartbeat is sent from the `TaskTracker` to the `JobTracker` every few minutes to check its status. The `JobTracker` and `TaskTracker` status and information can be viewed from a web browser.

14.2.1 Issues

The allocation of work to `TaskTrackers` is based on **slots**: every `TaskTracker` has a fixed number of slots for map tasks and reduce tasks, and the `JobTracker` allocates tasks to the `TaskTracker` based on the number of free slots. Hence there is no consideration concerning the actual workload of tasks, which may be heterogeneous, leading to an unbalanced workload amond the nodes.

If even one node is slow, the whole job is slow, as the `JobTracker` waits for the slowest task to finish.

With speculative execution enabled, however, a single task can be executed on multiple slave nodes.

14.3 Hadoop in depth

Hadoop is an open-source framework for distributed storage and processing of large datasets. It allows to use frameworks such as MapReduce to process data across large clusters.

A small Hadoop cluster includes a single master and multiple worker nodes. The master node consists of a *JobTracker*, *TaskTracker*, *NameNode*, and *DataNode*. A slave or worker node acts as both a *DataNode* and *TaskTracker*; it is possible to have **data-only** and **compute-only** worker nodes.

The *NameNode* manages the file system namespace and regulates access to files by clients. The *DataNode* manages storage attached to the nodes that it runs on.
The *JobTracker* receives jobs from clients and distributes them to *TaskTracker*s. The *TaskTracker* is responsible for executing the individual tasks as directed by the *JobTracker*.

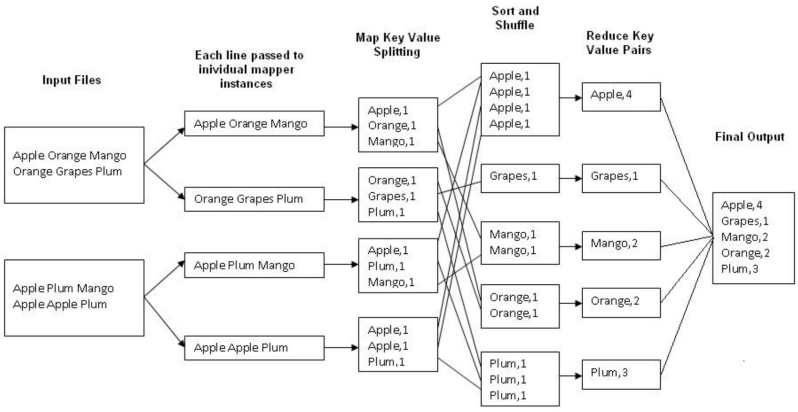


Figure 14.3: Hadoop WordCount example, equivalent to *Hello world!*

In Figure 14.3 *"individual mapper"* refers to a node implementing map.

In the slides there are some —not complete— instructions in Python to implement a simple WordCount using Hadoop.

Listing 14.1: Launching hadoop

```
hadoop jar
$HADOOP_HOME/share/hadoop/tools/lib/hadoop-
streaming-*.jar \
-input /user/hadoop/input \
-output /user/hadoop/output \
-mapper mapper.py \
-reducer reducer.py \
-file mapper.py \
-file reducer.py
```

Hadoop has been the first widely used solution, but nowdays also other solutions exists, such as Spark, while Hadoop is considered somehow outdated.

14.4 Spark

Figure 14.4: Spark logo

Spark is a fast and general-purpose cluster computing system. It provides high-level APIs in Java, Scala, Python, and R, and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming (quite outdated, the updated version is *"Spark structured streaming"*).

Lazy vs Eager evalutation

Definition 14.1 ***Lazy evaluation*** *is an evaluation strategy that delays the evaluation of an expression until its value is actually required (a dependent expression is evaluated).*

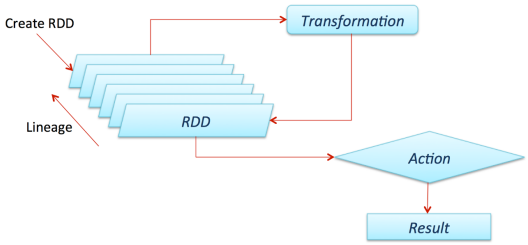
Definition 14.2 ***Eager evaluation*** *is an evaluation strategy that evaluates an expression as soon as it is bound to a variable.*

As we will see Spark is based on lazy evaluation, which allows to optimize the execution plan.

14.4.1 Architecture

Every Spark application consists of a *driver* program running the *main* function and executing various parallel operations on a cluster. The driver program accesses Spark through a *SparkSession* object, which represents a connection to a Spark cluster.

The main abstraction Spark provides is a **resilient distributed dataset** (`RDD`), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel.



Definition 14.3 (RDD) *It is a single* ***immutable*** *distributed collection of objects.*
Each dataset is divided into logical partitions, which may be computed on different nodes of the cluster.
RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.

Note that since an RDD is a single *immutable* distributed collection of objects, it fits nicely in the functional programming paradigm.

Listing 14.2: Creating an RDD

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

RDDs may be created by parallelizing an existing collection in your driver program, or by referencing an external dataset such as a shared file system, HDFS, HBase, or any data source offering a Hadoop InputFormat.

RDD can be persisted in memory, allowing it to be reused efficiently across parallel operations. RDDs automatically recover from node failures.

RDDs can be created from Hadoop InputFormats (such as HDFS files) or by transforming other RDDs.

Another abstraction in Spark are **shared variables**, which are **broadcast** variables and **accumulators**.
When Spark runs a function in parallel as a set of tasks on different nodes, it ships a copy of each variable used in the function to each task. A variable may need to be shared across tasks, or between tasks and the driver program. Spark supports two types of shared variables:

- **Broadcast variables** - allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks.
- **Accumulators** - are variables that are only added to through an associative operation and can therefore be efficiently supported in parallel.

RDDs support two types of operations: **transformations** and **actions**.

- **Transformations** - These are operations —such as `map` and `filter`— that create a new dataset from an existing one. These are *lazy* operations, meaning that Spark doesn't apply the transformation immediately, but remembers the transformation applied to the base dataset.
- **Actions** - return a value to the driver program after running a computation on the dataset. When an action is called, Spark computes the result of the transformation chain.

```
lineLengths = lines.map(lambda s:
    len(s))
totalLength = lineLengths.reduce(
    lambda a, b: a + b)
# lineLengths gets consumed by the
    reduce
```

`lineLengths` holds the result of the `map` **transformation**, which however is not computed instantaneously —due to *lazyness*—, and `totalLength` holds the result of the `reduce` **action**. At this point, Spark breaks the computation into tasks to run on separate machines, and each machine runs

```
    # we can instruct Spark to persist the
        RDD
    # lineLengths.persist() # place before
        reduce
```

both its part of the map and a local reduction, returning only its answer to the driver program.

Note that being RDDs immutable allows for three key advantages:

1. **Consistency** - optimizations and lazy evaluation of transformations wouldn't be possible if the RDDs were mutable. Besides, we can always go back to the original RDD, and there is no need to keep track of the changes and replicas of the data.
2. **Resiliency** - if a node fails, the RDD can be recomputed from the original data.
3. **Concurrency** - multiple tasks can operate on the same RDD without worrying about the data being changed, so zero concurrency issues.

Implementing a simple WordCount in Spark is quite simple, as shown in Listing 14.3. This is much simpler than setting up hadoop.

Listing 14.3: Spark WordCount example

```
import sys
from pyspark import SparkContext
sc = SparkContext(appName="WordCountExample")
lines = sc.textFile(sys.argv[1])
counts = lines.flatMap(lambda x: x.split(' ')) \
   .map(lambda x: (x, 1)) \
   .reduceByKey(lambda x,y:x+y)
output = counts.collect()
for (word, count) in output:
   print "%s: %i" % (word, count)
sc.stop()
```

Chapter 15

The Actor Model for the Compute Continuum

Computing infrastructures in 2024 are characterized by strong **pervasiveness**, we are surrounded by them, and computations happen everywhere anytime.

This is the **Compute Continuum**, a concept that describes the continuous interaction between the physical and digital worlds, and the seamless integration of the digital world into the physical world.

All assumptions which can (sometimes) be made in traditional programming models, cannot ever be made in the Compute Continuum:

- Centralized management and control
- Homogeneous resources
- Seamless access and well-defined security and identity mechanisms
- Performance predictability
- Uniform management and reliability processes
- …we could go on …

15.1 Meet up with today's necessities

Even MapReduce, which was designed for distributed computing, is not a good fit for this scenario. It does not deal well with heterogeneity, and it requires to know in advance where the computation will take place, which is not possible in the Compute Continuum, instead, often the infrastructure is managed by different parties, inducing further heterogeneity, lack of control, and oscillations in latencies and bandwidths.

However, we will see that the MapReduce is an instance of a more general concept/model: the *Bulk Synchronous Parallel* model.

15.1.1 BSP - Bulk Synchronous Parallel model

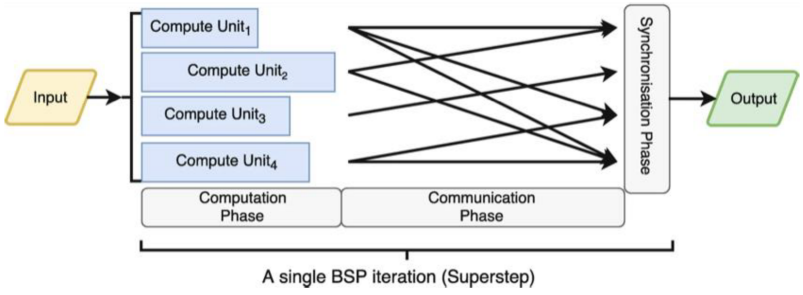


Figure 15.1: BSP Model schema

This model is a *bridging model* for designing parallel algorithms. It defines three global super-steps:

1. Concurrent computation
2. Communication
3. Barrier synchronization

The problem with having a full synchronization at the end, is that reliability and network partitioning can cause the system to hang a lot. This is a more critical issue than latency, throughput, and bandwidth.

15.1.2 Delegation

In order to reach the scalability we are looking for we want model that are by design decentralized.

- ◇ Actor Based models and frameworks
  - Akka - Java/Scala based, no longer OpenSource
  - Orleans - .NET based
  - CAF - C++ based
  - Ray - Python based
  - ZIO Actors - Scala based, oriented for computations happening in a single machine

  This is the model used by the Halo 4 backend
- ◇ Reactive programming - paradigm that emphasizes the use of asynchronous data streams to handle events and data flows.
  Used in the Netflix API

The underlying architecture is typically not simply threads, because a single machine may also manage millions of actors, and threads are not a good fit for this scenario. Actors may be organized, for instance in a list, and a single thread may invoke them in a round-robin fashion. We could also have multiple threads, handling such lists of actors; there are many possible implementations.

15.2 Towards the Actor Model

Given the strong decentralization of the Compute Continuum, the need for a decentralized computational models became strong.

Decentralized programming models provide unique features that are fundamental to support the development and efficient management of resources and applications in the compute continuum

- ◇ Adaptability - Many devices and platforms with different capabilities. Applications must adapt to different hardware and software environments.
- ◇ Openness - By using open standards and protocols, decentralized programming models enable applications to interoperate seamlessly across different devices and platforms.
- ◇ Modular philosophies - Applications as systems of interacting components, allowing for easier maintenance, updates, and scalability.
- ◇ Resilience - Systems must be able to recover from failures and continue operating.

The **Actor Model** is a computational model for decentralized concurrent computation that treats actors as the universal primitives of concurrent computation. In response to a message that it receives, an actor can make local decisions, create more actors, send more messages, and determine how to respond to the next message received. Actors may modify their own private state, but can only affect each other indirectly through messaging (removing the need for locks).

Often compared to OOP, the Actor Model differs in distribution and concurrency handling: while OOP typically relies on shared state, locks and synchronous method calls, the Actor Model emphasizes message passing and asynchronous communication between independent actors to achieve concurrency.
Furthermore, Actors are based on “behaviour”, rather than “class” as in OOP.

This model nicely fits modern computing challenges:
- ◇ Self-organisation programming
- ◇ Collective adaptive systems
- ◇ Cyber-physical systems and IoTs
- ◇ Edge and fog computing

Fun fact, the Actor Model was first proposed by Carl Hewitt in 1973, and it was inspired by the work of Alonzo Church on the *Lambda Calculus*, it was oriented for implemeting AI.

15.3 The Actor Model - Key Concepts

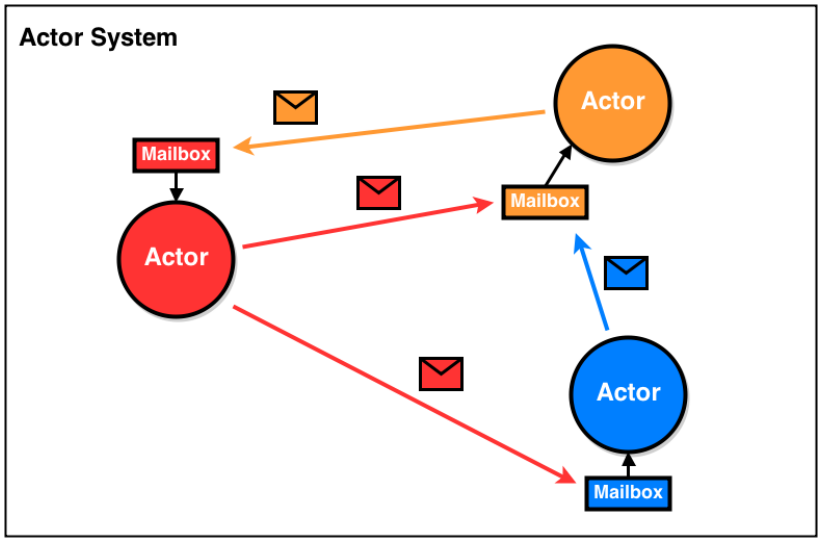


Figure 15.2: Actor model schema

The actor model provides a simple and intuitive programming model for building highly concurrent and distributed systems.

It emphasizes message passing and actor isolation, which helps to improve the modularity and scalability of the system.

Definition 15.1 (Actor) *An actor is a computational entity that, in response to a message it receives, can concurrently:*

- ◇ *send a finite a number of messages to other Actors*
- ◇ *create a finite number of new Actors*
  Enables to model to express any degree of concurrency
- ◇ *designate the behavior to be used for the next message it receives*
  Actors are not stateless, and keep track of the progress of the computation

Modularity in the Actor model comes from *one-way asynchronous communication*. Once a message has been sent, it is the responsibility of the receiver. Messages in the Actor model are **decoupled** from the sender and are delivered by the system on a best-effort basis.

Note also that since Actors can create new Actors, they implicitly create a hierarchical relation between them, allowing to exploit the locality of the computation.

15.3.1 Indeterminacy and quasi-commutativity

Messages are delivered asynchronously, and the order of delivery is not guaranteed, hence, the actor model is **not** completely **deterministic**. This is called **indeterminacy**.
Furthermore, when multiple messages are waiting in an actor's mailbox, the order in which they are processed is determined by the actor's scheduling policy, which may be non-deterministic.

Quasi-commutativity is a property of the actor model that states that the *order* of messages sent by an actor to another actor does *not* affect the outcome of the computation (in some cases).
In other words, there must be some flexibility concerning the order of messages, to avoid making the system obey a strict order of messages dictated by delays, errors, and lost messages.

15.3.2 Fault tolerance and Scalability

The actor model is inherently fault-tolerant, as the failure of one actor does not prevent the rest of the system from functioning, besides, actor may be restarted or moved to another machine in case of failure.
Furthermore, it provides a scalable approach to building systems by allowing actors to be dynamically created and distributed across multiple machines, enabling systems to scale horizontally by adding more machines to handle increasing loads. This is even simplified by the **absence** of primitives such as locks and semaphores, reducing the risk deadlocks, race conditions and bootlenecks due to busy waiting. One-way asynchronous communication allows all this.

Self-organising systems, as well as **adaptive** systems, may be built using the actor model framework, where actors can adapt their behavior based on the state of the system and the messages they receive, aiming at a global goal. Not to mention IoTs where, essentially, each device may be implemented as an actor.

15.3.3 Challenges

- ◇ **Complexity**: Designing and implementing actor-based systems can be complex
- ◇ **Testing**: Testing and debugging actor-based systems can be challenging due to their concurrent and asynchronous nature. It can be difficult to reproduce and isolate bugs that arise from race conditions
- ◇ **Overhead**: The actor model adds some performance overhead due to the need to manage and schedule actors, and the cost of message passing between actors
- ◇ **Consistency**: Ensuring consistency and coordination between actors can be challenging, especially when dealing with distributed systems where actors may be running on different machines or in different data centres.
- ◇ **Integration**: Integrating actor-based systems with existing systems and technologies can be challenging, especially when dealing with legacy systems or systems that use different programming languages or platforms.

Chapter 16

Distributed Messaging

Messaging systems enable loose coupling and asynchronous communication between distributed components, they act as the glue binding components together.

When applied to distributed computing, messaging systems can be used to implement a wide range of communication patterns, such as request/reply, publish/subscribe, and event sourcing. A too simplified message passing mechanism may not be enough, since it must take into account delays, message ordering, lost packets, scalability, and fault tolerance.

16.1 Message Passing mechanisms

- ◇ Request-response - a client sends a request to a server, and the server sends a response back.
- ◇ One-way messaging - a client sends a message to a server, and the server does not send a response.
- ◇ Publish-subscribe - a client subscribes to a topic, and the server sends messages to all clients subscribed to that topic.

Publish-subscribe is useful because allows to receive information from multiple services, explicitly (and implicitly) categorize them, and assign them different semantics.

16.1.1 Adapting Publish-subscribe to Point-to-point

Note however that trying to create point-to-point communication with publish-subscribe may be cumbersome, and lead to un-elegant architectures, like having all clients subscribing to everyone else: a mess.

Introducing a server in the middle simplifies the topology, and allows to have a single point of contact for all clients, which forwards messages to the correct recipient. On the other hand, it is not distributed, not scalable, and may become a bottleneck! Besides, if you need also a different server gathering different clients' messages, you are back to square one.

We could also have single queues for each client, but this is not scalable, and may lead to a single point of failure, again.

16.2 Message Brokers - Kafka

Message brokers are software systems that receive messages from producers and deliver them to consumers. They act as intermediaries between producers and consumers, and can provide additional features such as message storage, routing, and filtering.

Kafka is a distributed message broker that is designed for high-throughput, fault-tolerant, and scalable messaging. It is used in a wide range of use cases, such as log aggregation, stream processing, and event sourcing.

Messages in Kafka are similar to DB rows, and are stored in topics, which are similar to DB tables. Each message has a key, a value, and a timestamp, and is stored in a partition.

Messages are written in *Batches*, which are collections of messages produced to the same topic and partition within

a short time window. Batching improves throughput and reduces latency, but is a tradeoff: larger batches allows for more messages to be stored, but it might take longer for a message to be propagated.
Messages are opaque byte array to Kafka, which simply stores and forwards them. It is the responsibility of producers and consumers to serialize and deserialize messages. Hence, it is recommended that additional structure be imposed on the message content; JSON and XML are common portable choices, but many developers use **Apache Avro**.

Messages are categorized into **topics**, which in turn are broken down into **partitions**, where they are written in append-only fashion and are read in order from beginning to end. Each partition can be hosted on a different server, hence a single topic may be scaled horizontally across multiple servers to **scale-out**.

Multiple partitions imply there's no guarantee of message time-ordering across the entire topic.

Kafka sacrifices a global order of messages for scalability, and allows to have multiple consumers reading from the same topic, and even the same partition[1] at the same time. In general it is not impossible to have a global order —see blockchains— but it may be very costful in a distributed environment.
However, if it is necessary for a topic to have a global order, it is possible to have a single partition for that topic.

Bypassing the Single partition limitation

To avoid having a single partition but still preserve the order, we can implement some trick.

Suppose you have a topic for "user tracking", with each user being identified by a key.
You could create a partition for odd user IDs and one for even user IDs, and then have a single consumer for each partition.

16.2.1 Producers and Consumers

Producers write messages to topics, and consumers read messages from topics. Producers and consumers can be scaled horizontally to improve throughput and fault tolerance. Scaling out consumers allows to consume message-intense topics.
Producers may specify the partition to which the message should be written, use a custom partitioner, or let Kafka decide based on a partitioning strategy (e.g., round-robin, hash-based).

Consumers subscribe to topics and reads messages in order. The **offset** is a unique identifier for each message within a partition, and is used by consumers to keep track of their progress. The offset is always increased, and is never reset.
By storing the offset of the last consumed message for each partition a consumer can stop and restart without losing its place. Consumers work as part of a consumer **group**, which assures that each partition is consumed by only one consumer in the group.

Note that, in order to preserve the ordering of the message within a partition, a partition may be consumed only by one member of a consumer group at a time.
In this way, the sequential semantics of a partition are preserved.
This implies that the maximum degree of scalability is given by the number of partitions, and not by the number of consumers.

16.2.1.1 Rebalancing (Partitions number)

The number of partitions is set when the topic is created, and in general cannot be changed during the provisioning of the topic.
However, Kafka has a feature called *rebalancing*, which allows to dynamically create partitions, allowing for more consumers to consume from the same topic. However, it is a bit of a mess, and requires the provisioning to stop, compute the rebalancing plan, and then restart the provisioning.

Consumer group rebalancing (discussed later in Section on Rebalancing) is different and more common: it reassigns existing partitions among available consumers when members join or leave the group, without changing the number of partitions. This happens automatically and frequently in production systems.

[1] if they belong to different groups. This will be discussed later

16.2.1.2 Brokers and Clusters

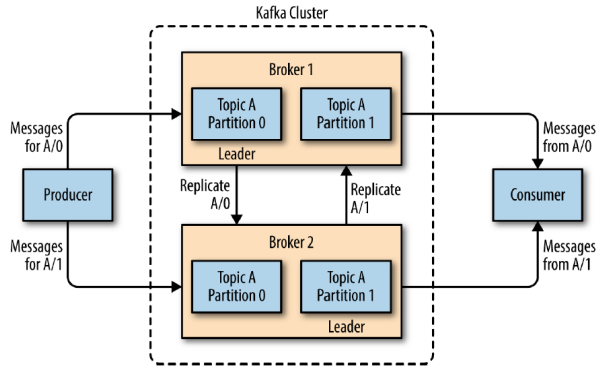


Figure 16.1: Kafka brokers cluster

A single Kafka server is called a *broker*, and a group of brokers is called a ***cluster***. Each broker is responsible for a set of partitions, and is able to handle reads and writes for those partitions.

Brokers communicate with each other to **replicate** data and keep the cluster in sync. Each partition has a **leader** broker, which is responsible for handling reads and writes for that partition in the cluster. The leader broker replicates data to follower brokers, which can take over as leader if the current leader fails.

Consumers always get messages from the leader, never from the replicas, allowing to enforce asynchronous consistency.

16.2.1.3 Retention

For each topic it is possible to define a **retention policy**, which specifies how long messages should be retained in the topic. Messages can be retained for a fixed amount of time, or until a certain size limit is reached.

Kafka brokers are configured with a default retention setting for topics, defining either size or time limits, which once reached, messages are deleted.

16.2.1.4 Multiple Clusters

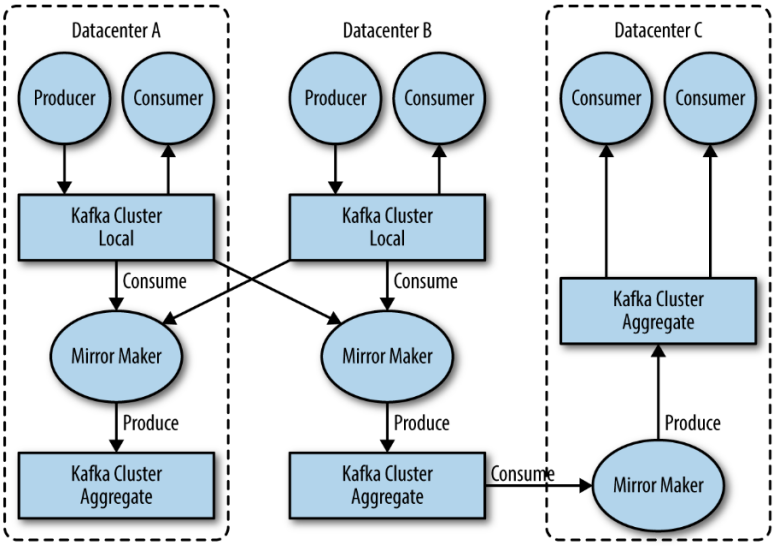


Figure 16.2: Clusters and `MirrorMaker`s schema
Here datacenter A and B locally produce messages. They both have a MirrorMaker consuming messages from the local cluster and from the other remote cluster. MirrorMakers produce messages to their respective **Kafka Cluster Aggregate**, which is a cluster getting a comprehensive view of the messages of all Datacenters. There is also a datacenter C, which does not produce any messages, only consumers are present; it has a MirrorMaker consuming messages from the Cluster Aggregate of datacenter B (which contains also messages from datacenter A) and producing them to its local Kafka Cluster Aggregate from which local consumers may read.

As kafka deployments grow, it is often advantageous to have multiple clusters, in order to have **segregation**, **isolation**, and **fault tolerance**.

When working with multiple datacenters it could be required that messages be copied between them. Kafka includes a tool called `MirrorMaker`, used for this purpose, which simply is a Kafka consumer and producer, linked together with a queue: messages are consumed from one Kafka cluster and produced for another

16.3 Using Kafka

16.3.1 Producers

First addressing the producer point of view, it is important to consider the requirements on this side:

- ◇ Is every message critical?
- ◇ Are accidental duplicates acceptable?
- ◇ Are there latency or throughput requirements?

These questions are important to correctly use the API for the producer, which can be either `fire-and-forget`, `synchronous`, or `asynchronous`.

- ◇ `fire-and-forget` - the producer sends the message and does not wait for a response. Kafka automatically retries to send messages if they fail to be delivered, so most of them will be actually delivered, but some may be lost.
- ◇ `synchronous` - the producer sends the message and waits for a response. This is the safest way to send messages, but it is also the slowest. `send()` returns a `Future()` object, which we may invoke `get()` on to wait for the response.
- ◇ `asynchronous` - the producer invokes `send()` along with a *callback* and continues with its execution. The callback is invoked when the message is successfully delivered, or when an error occurs.

16.3.1.1 Message record format

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

Once sent the `ProducerRecord`, the producer will serialize the key and value objects so can be sent over the network. Later on, if a partition is specified in the `ProducerRecord`, the partitioner simply returns the partition we specified, otherwise it computes the partition based on the key (if present) or in round-robin fashion.

A separate thread is responsible for sending those batches of records to the appropriate Kafka brokers, which yield a Metadata record upon successful delivery.

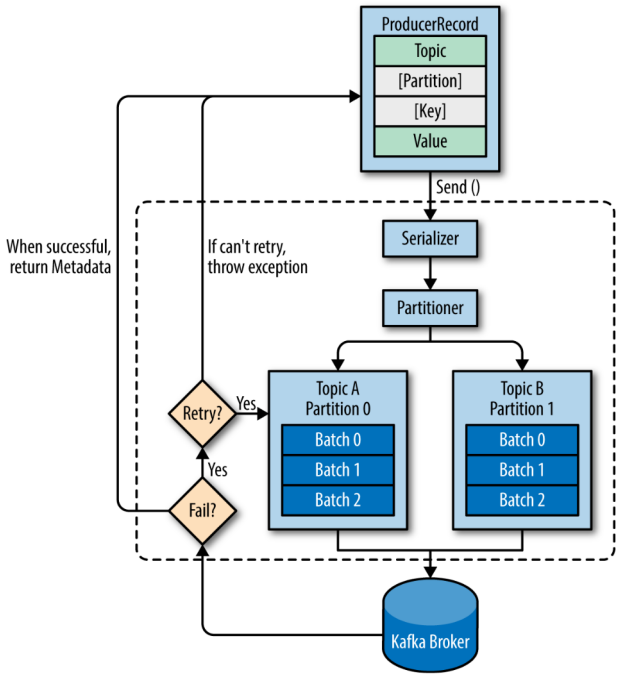


Figure 16.3: Sending `ProducerRecord`

16.3.2 Consumers

When dealing with Kafka messages it is important to consider that the producers may write to topics at a rate higher than the consumers can read from them.
This is undesirable, as it may lead to the consumers falling behind, and eventually running out of memory; hence, we must allow to scale the consumers.

Kafka consumers are typically part of a group. When multiple consumers are subscribed to a topic and belong to the same group, each consumer in the group will receive messages from a different subset of the partitions in the topic.

Having more consumers than partitions is not a problem, as the consumers will simply be idle.
In general it is reasonable to have many partitions, allowing to scale when needed.

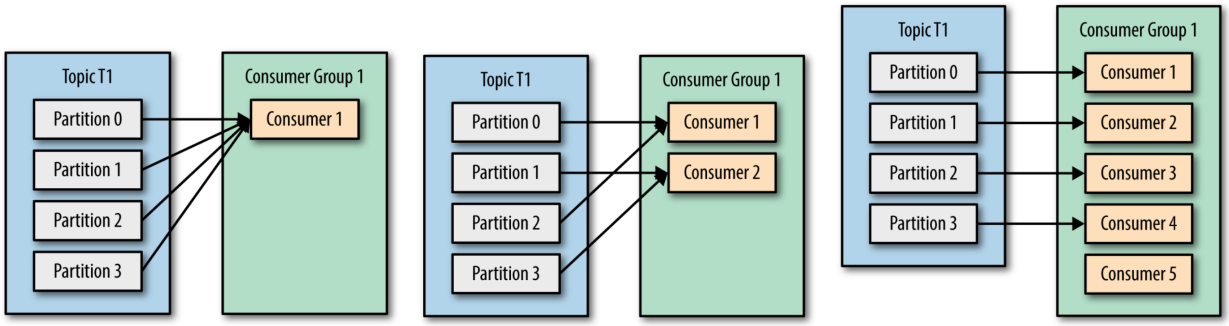


Figure 16.3: Partitions are assigned to Consumers to have a fair distribution.

It is not uncommon to have multiple applications that need to receive the same data. Simply assigning different consumers groups to different applications ensures that messages are delivered to all of them.

The separation among consumers remains.

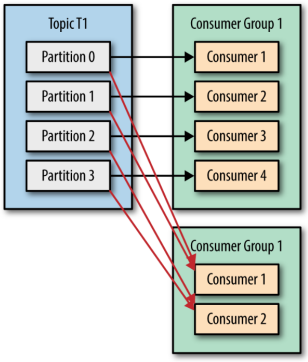


Figure 16.4: Multiple Consumer Groups

16.3.3 Rebalancing

Moving partition ownership from one consumer to another is called a **rebalance**. Rebalances are important because they provide the consumer group with high availability and scalability, but in the normal course of events they are *fairly undesirable*, because during a rebalance, consumers can't consume messages, so a rebalance is basically a short **window of unavailability** of the entire consumer group.

When partitions are moved from one consumer to another, the consumer loses its current state; if it was caching any data, it will need to refresh its caches—slowing down the application until the consumer sets up its state again.

16.3.3.1 When is it needed?

- ◇ A new consumer joins the group
- ◇ A consumer leaves the group
- ◇ A consumer is considered dead
- ◇ The set of partitions for a topic changes

The most interesting case is when a consumer "dies". If the consumer stops sending heartbeats for long enough, its session will time out and, after a few seconds, the group coordinator will consider it "dead" and trigger a rebalance. During those seconds, no messages will be processed from the partitions owned by the dead consumer ☹.

Instead, when closing a consumer cleanly, the consumer will notify the group coordinator that it is leaving, and the group coordinator will trigger a rebalance immediately.

16.4 Kafka Strengths

Kafka is able to seamlessly handle multiple producers, whether those clients are using many topics or the same topic, and multiple consumers, whether they are in the same group or different groups, and also allows to seamlessly read from the same stream without interfering with each other, opposed to many queuing systems where once a message is consumed by one client, it is not available to any other.

Disk-based retention allows consumers to avoid working necessarily in real-time, and to catch up with the stream at

their own pace. Durable retention means that if a consumer falls behind, either due to slow processing or a burst in traffic, there is no danger of losing data. This also means that maintenance can be performed on consumers, taking applications offline for a short period of time, with no concern about messages backing up on the producer or getting lost.

All the presented features make Kafka a powerful and flexible messaging system, suitable for a wide range of use cases in distributed computing, especially:

- ◇ Activity tracking
- ◇ Messaging
- ◇ Metrics and logging
- ◇ Commit log
- ◇ Stream processing

Chapter 17

Data Processing

Data may be processed in either **batch** or **real-time** (*stream*) fashion, or a combination of both. Batch processing is used when data can be collected and processed in a single operation, while real-time processing is used when data must be processed as soon as it is generated.

17.1 Batch Processing

This approaches relaxes the need for producing results as soon as data arrives, storing it in a buffer to later process it in chunks.

Batch processing is typically cheaper and easier to implement than real-time processing, but it may not be suitable for all use cases, but is very effective for data transformations, generate reports, and other tasks that do not require immediate results.

We have high latency, but also high throughput, and the system is easier to reason about, however, it lacks the ability for real-time decision making.

The key paradigm here is clearly MapReduce, and the key frameworks are Hadoop and Spark.

Batch processing is characterized by a trade-off between throughput and latency; it typically has high throughput but can have high latency depending on the size and complexity of the workload.

17.2 Stream Processing

Here the focus is on processing data as soon as it is generated, providing low-latency results, and thus enabling near-real-time processing.
Stream processing is designed to handle unbounded datasets, such as those generated —typically in a continuous flow— by IoT devices, allowing to compute real-time analytics, monitoring, and decision-making.

17.2.1 Challenges

Stream processing requires efficient **state management** in a system with high data velocity.
Fault tolerance is a key challenge in stream processing due to the potential for data loss if a node fails.
Consistency for data spread across multiple nodes is not trivial to achieve.
Furthermore, stream processing system feature the **windowing** concept, which is powerful, but bears with it the need to define proper **window boundaries**.

Data consistency refers to the accuracy and correctness of data over time. In stream processing, data **consistency** is crucial to ensure that real-time analytics are accurate and reliable. There are three main types of data consistency models used in stream processing systems:

- **Exactly-once** processing ensures that each data item is processed exactly once, preventing duplicates or omissions. This is the most desirable level of consistency, but it can be challenging to achieve in distributed systems.
- **At-least-once** processing guarantees that each data item is processed at least once, but it may be processed multiple times, leading to potential duplicates.

- ◇ **At-most-once** processing ensures that each data item is processed at most once, but it may be lost if there are failures in the system.

17.2.2 Time semantics

In stream processing systems, understanding different time semantics is crucial for correctly handling events, especially when dealing with out-of-order data or latency issues. The three main time concepts are defined from the **perspective of the stream processing system** (e.g., Flink, Kafka Streams, Spark Streaming):

- ◇ **Event time**: The timestamp when the event *actually occurred* in the real world, typically embedded in the event data itself. For example, when a sensor detects a temperature reading, the event time is the moment the sensor took that measurement. This is the most accurate representation of when something happened.
- ◇ **Ingestion time**: The timestamp when the event *enters the stream processing system* (e.g., when it arrives at the Kafka broker or is read by the streaming application). There may be delays between event time and ingestion time due to network latency, batching, or other factors.
- ◇ **Processing time**: The timestamp when the stream processing system *actually processes* the event (e.g., when a Flink operator applies a transformation or aggregation to it). This is the system's wall-clock time at the moment of processing. Processing time can differ significantly from event time due to system load, backpressure, or processing delays.

Example scenario: A temperature sensor records 25°C at `10:00:00` (event time). Due to network delays, this data reaches Kafka at `10:00:05` (ingestion time). The Flink application processes this event at `10:00:10` (processing time) because it was busy handling previous events.

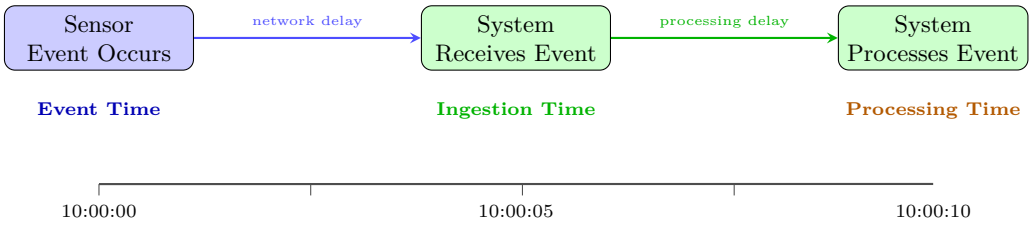


Figure 17.1: Time semantics in stream processing: the lifecycle of an event from occurrence to processing

The choice of time semantics affects windowing, aggregations, and correctness guarantees. Event time is preferred for accurate analytics, but requires handling out-of-order events and late arrivals. Processing time is simpler but can produce incorrect results if events are delayed.

Watermarks are used to track the progress of event time, and to determine when to emit results. They act as markers indicating that no more events with timestamps earlier than the watermark should be expected (or that such events will be considered late). In general, watermarks aim to manage late-arriving data by providing a mechanism to balance between result accuracy and latency.

In **Stream** processing there are **Transactions**, which are atomic operations between streams. They are implemented with two-phase commit protocols, where a coordinator manages the transaction and ensures that all participants either commit or abort the transaction together, or idempotent writes, i.e. writing the same data multiple times has the same effect as writing it once.
To address Fault Tolerance in stream processing, systems often implement checkpointing and state backends, or log replay, to periodically save the state of the stream processing application. This allows the system to recover from failures by restoring the state from the last checkpoint, minimizing data loss and ensuring continuity in processing.
Clearly it may happen also that events arrive out of order, and we need to handle this, either by buffering to wait for more data, or by allowing lateness in *windowing*.

- ◇ **Time-based windowing** is used to group events into windows based on time, and to process them in parallel. Windows can be fixed or sliding, and can be based on event time or processing time.
- ◇ **Count-based windowing** instead is used to group events into windows based on the number of events, but the same principles apply.
- ◇ **Session-based windowing** groups events based on inactivity periods.

17.3 Lambda architectures

A **lambda architecture** is a data processing architecture designed to handle massive quantities of data by taking advantage of both *batch* and *stream* processing methods. It consists of three layers:

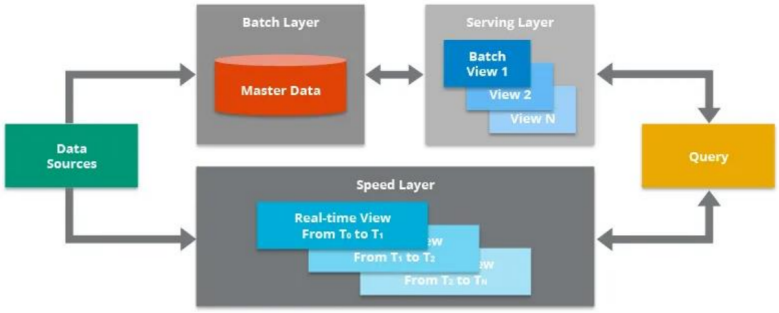


Figure 17.2: Lambda architecture schema
Data sources simultaneously feed both the batch layer and the speed layer. The batch layer processes the data in large chunks, while the speed layer processes the data in real-time. The serving layer then combines the results from both layers to provide a unified view of the data to the end-users. It gets data directly from the batch layer, while it uses queries to the speed layer to get real-time updates.

- ◇ The **batch layer** is responsible for processing large volumes of data in a fault-tolerant and scalable manner. It is used to generate views of the data that can be queried by the serving layer.
- ◇ The **speed layer** is responsible for processing real-time data in a low-latency manner. It is used to generate real-time views of the data that can be queried by the serving layer.
- ◇ The **serving layer** is responsible for serving queries on the views generated by the batch and speed layers. It is used to provide real-time access to the data to the end-users through APIs and various tools.

The architecture provides a unified view of both batch (hence, historical) and real-time data processing, possibly handling large volumes of data and scaling horizontally. However it may be complex to implement and maintain, and may generate redundant data processing and storage costs.

Lambda architecture requires careful coordination between batch and speed layers. The architecture may even need to be redesigned or updated as data processing needs evolve.

17.4 Kappa architectures

Lambda difficulties in implementation led to the development of Kappa, which simplifies the architecture by removing the batch layer, focusing on streams, ultimately offering better scaling and fault tolerance than lambda.
Kappa relies on recomputing state from streams rather than storing it. It exploits checkpointing and state backends to manage state.

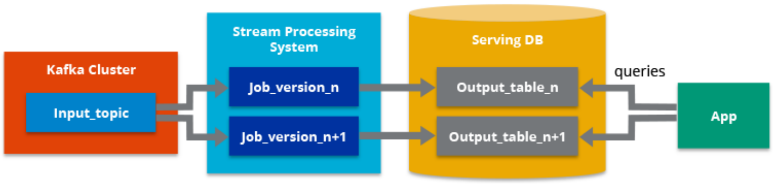


Figure 17.3: Kappa Architecture

A Kappa architecture is a data processing architecture designed to handle massive quantities of data by taking advantage of stream processing methods. It consists of a single layer that processes both batch and real-time data in a fault-tolerant and scalable manner.

The main components, as shown in Figure 17.3, are:

- ◇ **Kafka Cluster (Input Topic)**: Acts as the immutable event log that stores all incoming data. This distributed messaging system provides durability, scalability, and fault tolerance for the data stream. All events are retained for a configurable period, enabling replay and reprocessing.
- ◇ **Stream Processing System**: The core computation layer that processes events from Kafka in real-time. Multiple job versions (e.g., `job_version_n`, `job_version_n+1`) can run simultaneously, allowing for seamless updates and deployments without downtime. This enables A/B testing and gradual migration to new versions.

- ◇ **Serving DB (Output Tables)**: Stores the materialized views and results produced by the stream processing jobs. Different job versions write to separate output tables, allowing clients to query the appropriate version. This database must support high write throughput and low-latency reads.
- ◇ **App (Client)**: End-user applications that query the serving database to access processed results. The app can send queries to specific output table versions based on requirements.

KAPPA is very well suited for processing data from IoT devices, or analytics in general, where continuous stream processing is the primary requirement.

It is still more complex than traditional stream processing techniques, but it is more efficient and scalable.

17.5 Other Architectural flavors

17.5.1 CQRS - Command Query Responsibility Segregation

" What happens if i have many writes and few reads? "

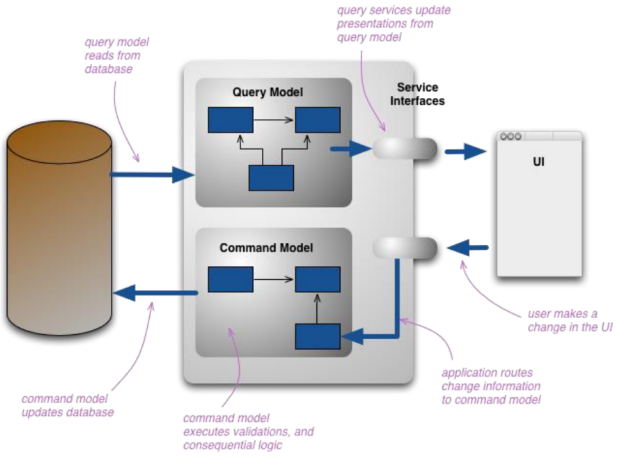


Figure 17.4: CQRS architecture

Command Query Responsibility Segregation (CQRS) is a design pattern that separates the read and write operations of a system into two separate components, along with a UI to interact with them:

- ◇ The **command model** (write) is responsible for handling the write operations of the system, such as creating, updating, and deleting data. Hence it is responsible for altering the system **state**.
- ◇ The **query model** (read) is responsible for handling the read operations of the system, such as retrieving data.

This allows for better scalability, performance, and maintainability of the system.

This looks good on the paper, but it is not always easy to implement, and may lead to inconsistencies between the read and write models, alongside with latency and communication overhead.

17.5.2 Shared-Nothing

A shared-nothing architecture is a distributed computing architecture in which each node is **self-sufficient** and has its own private memory and storage. The nodes communicate with each other by **passing messages**. This seems to nicely fit the Actor Model approach, as each actor can be seen as a node in a shared-nothing architecture. In this paradigm, data is split across multiple nodes, with each node responsible for a subset of the data, on which it has complete control.

This architecture allows for horizontal scaling by adding more nodes to the cluster, enabling also fault-tolerace, as a failure in one node does not affect the rest of the cluster.
In general its design favors high availability and low-latency.

This is cool and has many advantages, but it also poses challenges in terms of complexity, communication overhead, and data partitioning strategies. Despite offering a high degree of flexibility, it may be complex to standardize and maintain.

Shared-nothing architectures are commonly used for big data processing and web applications.

Hadoop and DynamoDB exploit shared-nothing architectures.

Feature	Lambda Architecture	Kappa Architecture	CQRS	Shared-Nothing Architecture
Description	Combines batch and stream processing for unified data views.	Focuses solely on stream processing, eliminating batch layers.	Separates read (queries) and write (commands) operations.	Decentralized nodes with no shared resources.
Primary Use Case	Mixed workloads requiring both batch and real-time data processing.	Real-time data streams, such as IoT or continuous analytics.	Systems with high read/write loads or complex domain logic.	Large-scale distributed systems requiring high scalability.
Key Components	Batch Layer, Speed Layer, Serving Layer.	Stream Processor, State Management, Event Logs.	Command Model, Query Model, Event Store.	Independent nodes, Partitioned Data, Local State.
Data Consistency	Consistency eventually across batch and stream layers.	Eventual consistency in state and data views.	Separate consistency models for commands and queries.	Relies on consistency protocols (e.g., eventual or strong).
Scalability	High scalability but requires coordination across layers.	Naturally scalable due to single data pipeline.	High scalability with independent query and write paths.	Horizontal scalability with no shared bottlenecks.
Latency	Low for speed layer; higher for batch processing.	Low latency, designed for real-time processing.	Low latency for reads, but can have write overhead.	Minimal latency in read/write, depends on partitioning.
Fault Tolerance	High, with redundancy in both batch and stream layers.	High, with focus on replaying events for state recovery.	Fault tolerance depends on event sourcing mechanisms.	High, as nodes operate independently.
Complexity	High, due to the need to manage two processing layers.	Moderate, simpler than Lambda but requires stream expertise.	High, due to synchronization between command and query models.	High, complexity in partitioning and load balancing.
Advantages	Unified view of historical and real-time data.	Simplified architecture with real-time focus.	Optimized read/write paths; flexibility in scaling.	High scalability and resilience; no single point of failure.
Disadvantages	Complexity in maintenance; redundant storage.	Limited to real-time use cases; relies heavily on logs.	Higher implementation cost; requires expertise.	Difficult to ensure consistency and partitioning strategy.
Examples	Log analysis, fraud detection, hybrid analytics.	IoT sensor data processing, real-time analytics.	E-commerce systems, banking transaction systems.	Global-scale distributed databases (Cassandra, DynamoDB).
Best For	Scenarios needing both batch and real-time processing.	Systems requiring low-latency, real-time stream data.	High-volume, complex, or read-heavy applications.	Large-scale distributed systems with decentralized data.

Table 17.1: Comparison of Data Processing Architectures

17.6 Data Processing Frameworks

17.6.1 Data Lakehouse

This is a modern data architecture that combines the best features of data lakes and data warehouses. It allows for the storage of raw data in a data lake, and the processing of that data in a data warehouse.

A data lake is a centralized repository that allows for the storage of *structured and unstructured* data at any scale. Its main use cases include data archiving, big data analytics, and machine learning.
A data warehouse, instead, is a centralized repository that allows for the storage of *structured and processed data from multiple sources*. Its main use cases include business intelligence, reporting, and data analysis. Differently from the data lake, the main focus of a data warehouse is on data quality, consistency, and performance, while a data lake is more about storing large volumes of raw data for future processing.

This architecture provides unified approach for structured, unstructured, and semi-structured data, and allows for the use of both batch and real-time processing methods.

An example is SNOWFLAKE, which is a cloud-based data warehouse that supports both batch and real-time processing methods.

17.6.1.1 Data Mesh

Data Mesh is a modern architectural approach designed to address the challenges of managing large-scale and complex data systems, especially in distributed organizations. It emphasizes decentralization, domain-driven design, and treating data as a product.

Key difference from Data Lakehouse: While **Data Lakehouse** is a *technical architecture* focused on *where and how data is stored* (combining lake and warehouse capabilities in a unified platform), **Data Mesh** is an *organizational and architectural paradigm* focused on *who owns and manages data* (decentralized ownership by domain teams).

Main characteristics:
- ◇ **Decentralization**: Data responsibility is distributed across domain teams, avoiding a centralized data team as a bottleneck. Each domain owns its data as a product.
- ◇ **Domain-oriented ownership**: Data is organized and owned by business domains (e.g., Sales, Marketing, Logistics), not by a central data platform team.
- ◇ **Data as a Product**: Each domain treats its data as a product with clear ownership, quality standards, and APIs for consumption.
- ◇ **Self-serve data infrastructure**: Provides platform capabilities that enable domain teams to manage their own data pipelines and products autonomously.
- ◇ **Federated computational governance**: Establishes global policies and standards while allowing domains autonomy in implementation.
- ◇ **Interoperability**: Data products across domains are designed to be interoperable through standard interfaces and contracts.
- ◇ **Scalability**: Scales naturally with organizational and data growth, as each domain independently manages its data.
- ◇ **Data Discovery**: Metadata and catalogs help discover data assets across the distributed domains.
- ◇ **Automation**: Strong reliance on automation for deploying and managing data pipelines.
- ◇ *Example*: Netflix uses Data Mesh principles to provide teams with the autonomy to own their datasets while maintaining consistency and governance.

Data Lakehouse vs Data Mesh: A Data Lakehouse can be used *within* a Data Mesh architecture as the technical storage layer for individual domain data products. They are complementary: Lakehouse solves the *"what storage technology"* problem, while Mesh solves the *"how to organize teams and ownership"* problem.

17.6.1.2 Choosing the right architecture

- ◇ **Lambda Architecture**: Best for systems needing both real-time and historical data processing.
  - *Example*: Log analysis systems combining real-time alerts and long-term trend reports, or big data analytics platforms.
- ◇ **Kappa Architecture**: Ideal for real-time data applications with high throughput.
  - *Example*: IoT sensor data processing, where insights must be generated continuously.
- ◇ **CQRS**: Suitable for applications with distinct read and write patterns, or complex domain logic.
  - *Example*: E-commerce platforms with high read traffic (product browsing) and write operations (orders).
- ◇ **Shared-Nothing Architecture**: Perfect for distributed systems needing extreme scalability.

- *Example*: Global-scale applications like social networks or e-commerce backends.

17.6.2 Federated Learning

With federated learning, the model is trained on the **edge** devices, and the updates are sent to a central server. This allows for better privacy and security, as the data never leaves the edge devices.

This approach is well suited for IoT devices, as it allows for the training of models on the edge devices, and the updates are sent to a central server for aggregation.

17.6.3 Serverless Data Processing

Abstracts infrastrcture management, and allows for the deployment of data processing pipelines without the need to manage servers. This paradigm is increasingly applied to data processing workflows.

The key difference with traditional architectures is that serverless computing automatically manages the allocation of compute resources, scaling them up or down based on demand. There is no need to provision or manage servers, as the cloud provider takes care of that.

Chapter 18

TLAV - Thinking Like a Vertex

MapReduce + HDFS is a solution for many BigData problems, as it allows for processing large amounts of data in a distributed manner. However, it is not the best solution for all use cases, as it has some limitations, such as high latency and lack of real-time processing capabilities.

Recall that HDFS is a distributed file system that allows for storing large amounts of data across multiple machines.

18.1 Spark SQL

Spark SQL allows to execute SQL queries, which are translated into Spark jobs (more precisely as *actions* and *transformations*). It is built on top of the Spark Core, and provides a more user-friendly interface for working with data. The result is yielded as a DataFrame, which is a distributed collection of data organized into named columns, on which we may execute various operations by means of a high-level SQL `spark.sql("SELECT * FROM people")` API.

18.1.1 Structured Streaming

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

Streaming computation represented in the same way of a batch computation on static data. The Spark SQL engine will take care of running it incrementally and continuously and updating the final result as streaming data continues to arrive.

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

When we say "exactly-once semantics" we mean that each incoming event affects the final results exactly once. Even in case of a machine or software failure, there's no duplicate data and no data that goes unprocessed.

The key idea in structured streaming is to treat a live stream as if it were a table that is being continuously appended. This allows for writing SQL queries that process the data in real-time, and to get the results as soon as new data arrives.

Consider the input data stream as the "Input Table". Every data item that is arriving on the stream is like a new row being appended to the Input Table.

A query on the input will generate the "Result Table". Every trigger interval, new rows get appended to the Input Table, which eventually updates the Result Table. Whenever the result table gets updated, we would want to write the changed result rows to an external sink

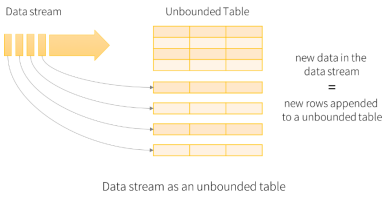


Figure 18.1: Structured Streaming schema

Output mode

- ◇ **Complete** Mode - The entire updated Result Table will be written to the 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 since the last trigger will be written to the external storage.
  - Note that this is different from the Complete Mode in that this mode 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, because without aggregations (e.g., simple `SELECT`, `WHERE`, `MAP` operations), each input row produces a new output row independently, without modifying existing rows in the Result Table. Aggregations (e.g., `COUNT`, `SUM`, `AVG`) are what cause existing result rows to be updated when new data arrives, since they maintain state across multiple input rows.

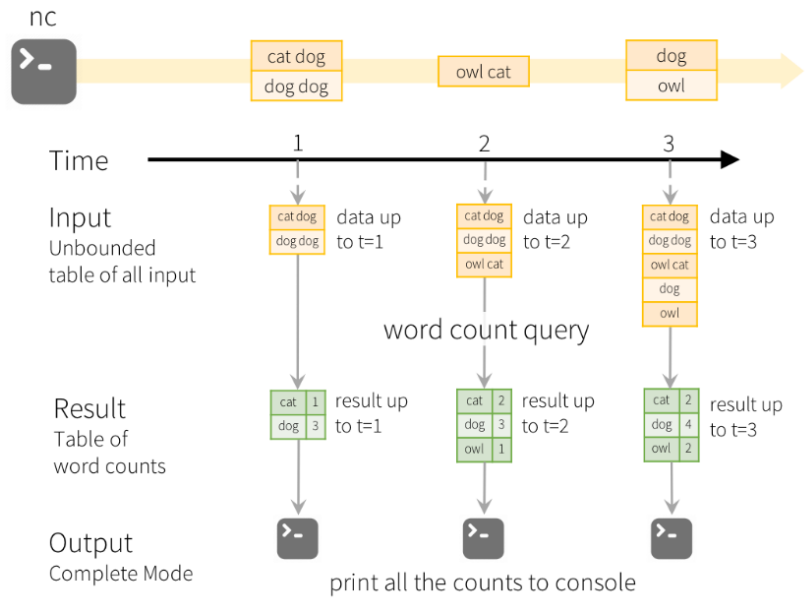


Figure 18.1: Spark query output example

Note that Structured Streaming does not materialize the entire table. It reads the latest available data from the streaming data source, processes it incrementally to update the result, and then discards the source data. It only keeps around the minimal intermediate state data as required to update the result.

18.2 Working on Graphs

Graphs are a powerful data structure that can be used to represent complex relationships between entities. A graph is made up of vertices (also called nodes) and edges (also called links).

We may use MapReduce to process graphs, but it is not the most efficient way to do so, as it requires multiple passes over the data, and it is not well-suited for iterative algorithms. MapReduce forces to think in terms of a linear sequence of operations, which does not fit well with the nature of graph algorithms, which are often iterative and require multiple passes over the data.

18.2.1 GraphX

GraphX is a distributed graph processing framework built on top of Apache Spark. It provides a set of APIs for working with graphs, and allows for the execution of graph algorithms in a distributed manner. GraphX represents graphs as a combination of two RDDs: one for the vertices and one for the edges. This allows for efficient processing of large graphs, as the data can be partitioned and distributed across multiple machines.

GraphX exploits the concept of TLAV - Think Like a Vertex, which is a programming model for graph processing that allows the user to define a function that is executed on each vertex of the graph.

18.2.1.1 TLAV - Think Like a Vertex

In TLAV, the programming logic is organized around vertices, which are the primary units of computation.

Each vertex maintains its state (i.e., its label and attribute) and can send messages to its neighbouring vertices based on its state and the state of its neighbours. The messages are then received by the neighbouring vertices, which can update their state accordingly.

This is similar to the actor model, where each actor maintains its own state and communicates with other actors through message passing.

TLAV is a powerful abstraction that can be used to express a wide range of graph algorithms in a concise and efficient manner. By thinking like a vertex, you can design efficient and scalable graph algorithms that can process large-scale graphs in a distributed environment.

18.2.1.2 Actor Model, BSP and TLAV

In GraphX, the actor model is used to represent vertices in a graph and to perform distributed graph processing.

Each vertex is represented as an actor, which communicates with other actors by exchanging messages. By using the actor model, GraphX provides a programming model for distributed graph processing that is based on vertex-centric computations.

In GraphX, the computation proceeds in a series of supersteps, where each superstep consists of message passing and vertex computations. In each superstep, each vertex receives messages from its neighbours and updates its state based on those messages. After each superstep, the vertices exchange messages and the computation proceeds to the next superstep.

Chapter 19

Vertex Centric Computing

This lecture was held by Emanuele Carlini.

The lecture spans in three sections:

1. Introduction to vertex-centric computing
2. Real stuff from Paper
3. Talk about thesis projects.

19.1 Introduction to vertex-centric computing

Vertex-centric computing takes a bit from Distributed computing, Graph theory, and Big Data.

19.1.1 Graph Theory

First, let'address graphs. Graphs are typically defined as $G = (V, E)$, where V is the set of vertices, and E is the set of edges. Edges can be directed or undirected, and can have weights.
Vertices have a in/out[1] degree.

Edges typically represent the relationships between vertices.

Definition 19.1 (Graph Theory) *Graph theory is a branch of mathematics that studies graphs, which are mathematical structures used to model pairwise relations between objects.*

Hence, GT deals with Connectivity, Flows, Topology, and many other things.

19.1.1.1 Topologies

- ◇ **Random graphs** - Every edge has a probability p of being present.
- ◇ **Small-world graphs** - Most nodes can be reached from every other node by a small number of hops.
- ◇ **Preferential attachment** - New nodes prefer to connect to nodes with high degree.

19.1.2 Towards Vertex-Centric

Definition 19.2 (Vertex-Centric) *Vertex-centric computing is a programming model for graph processing that allows the user to define a function that is executed on each vertex of the graph.*

Listing 19.1: TLAV in a nutshell

```
for each iteration
   for each vertex v
      fetch data from neighbors
      update internal state
      send data to neighbors
```

[1] In case of a directed graph

Note that each node can know something without seeing the whole network.

TLAV's logic is very simple and straightforward, making it very flexible; it may be applied to:
- ◇ Decentralized networks (one host = one node)
- ◇ Distributed networks (one host = multiple nodes)
- ◇ Multi-core machines (one core = multiple nodes)

19.2 Practical Examples

19.2.1 Fast Connected Components Computing in Large Graphs by Vertex Pruning

A paper by Emanuele Carlini, Alessandro Lulli, Patrizio Dazzi, Claudio Lucchese and Laura Ricci.
Definition 19.3 (Connected Component) *A **Connected Component** (CC) in an undirected graph is a maximal subset of vertices in which there is a path between any pair of vertices. In directed graphs, we distinguish between **Strongly Connected Components** (SCC, with directed paths in both directions) and **Weakly Connected Components** (ignoring edge direction).*
Note: A Connected Component is different from a **knot** (as seen in deadlock detection in directed graphs). In undirected graphs, a CC is based on bidirectional connectivity. A knot (in directed graphs) requires that every node has at least one outgoing edge to another node in the subset, forming a cyclic structure. While a knot is always within a (weakly) connected component, a connected component is not necessarily a knot.

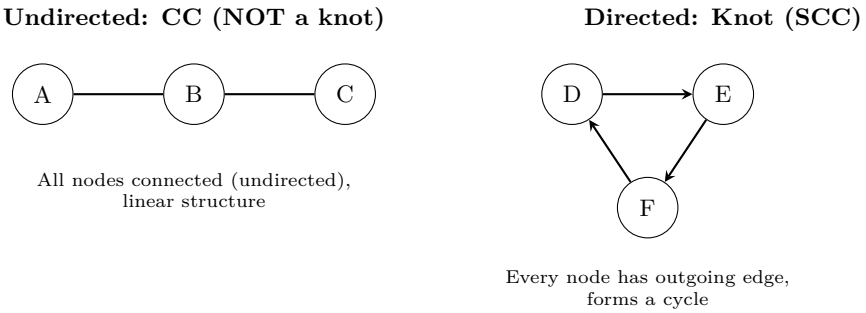


Figure 19.1: Difference between Connected Component and Knot: the left shows an undirected CC (linear structure, paths exist in both directions due to undirected edges), while the right shows a directed knot which is also a Strongly Connected Component (cyclic structure with directed paths)

The paper talks about an algorithm called *"Cracker"*, which is a vertex-centric algorithm to find connected components in undirected graphs, which exploits an **incremental** approach and **message passing** to achieve its goal.

The key idea is to incrementally remove nodes when we are sure of their CC. Each CC is identified by a unique ID. Each iteration takes less time than the previous one.

```
while (not finished)
   // find the seeds
   selection()
   simplification()
// propagate the seeds
propagation()
```

This was used as backend for a web page crawler. Many web pages were represented as nodes, with edges representing similarities computed with locality-sensitive hashing (was this the name?).
Computing the CCs allowed to group similar pages together, and perform more specific heuristics on them, which, being polynomial (worse than linear) would have taken too much time to be performed on the whole graph.

19.2.2 Decentralized minimum vertex cover

Definition 19.4 (Vertex Cover) *A **Vertex Cover** on an undirected graph G is a subset C of the nodes of the graph, such that for each edge, at least one endpoint of the edge is included in C. The minimum vertex cover problem consists in finding the C of minimum size.*
- ◇ Every node starts as deselected.
- ◇ Superstep 1 - Mark as selected the node with the highest degree among the neighbors and the node itself
- ◇ If the node is selected or all of its neighbors are, do nothing
- ◇ Superstep 2 - Otherwise, select among all non selected neighbors, the one with the best gain, until you find a worst neighbor, and select yourself.
- ◇ Superstep 3 - If the node and all its neighbors are selected, deselect the node