Scalable Distributed Computing - Appunti

Francesco Lorenzoni

September 2024

Contents

Part I

Introduction to SDC

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.

This implies to provide definitions not only for the highest-level interfaces, i.e., those seen by users and applications, but also for interfaces to internal parts of the system and describe how those parts interact.

This approach is an alternative to the classical monolithic approach in which components are implemented as one, huge program makes 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.

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

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

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

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
3. Handle **inconsistencies** between nodes
4. 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 *phyisical* 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.

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.

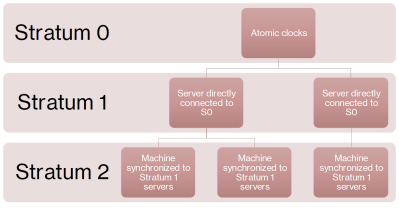

Figure 3.2: NTP Stratum architecture

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, than we can't achieve the high precision.

In future lessons theses two protocols will be covered in more detail

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.

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.

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

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.

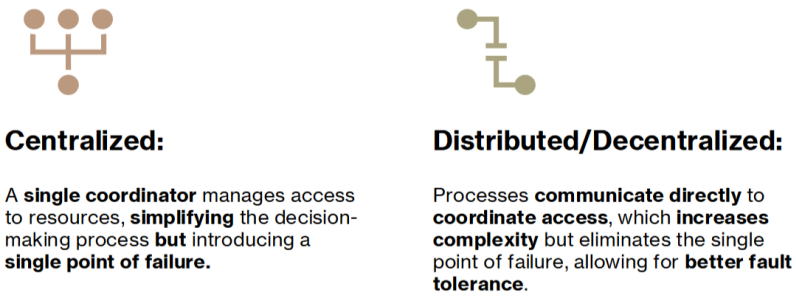

Figure 4.1: Mutual exclusion - Centralized vs Distributed

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++) {
      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 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

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.

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.

Figure 4.2: LDMEA Key Steps

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 $request_queue_i$

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**

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]`
- ◇ 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]` of the token equal to `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 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

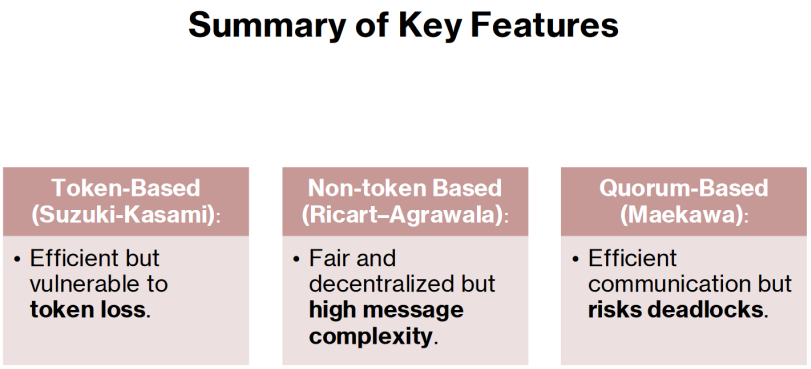

Figure 4.3: Summary of the Key features

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

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

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.

5.3 Resolving Deadlocks

- ◇ **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.

5.3.1 System Model for Deadlock Detection

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).

Thew 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

5.4 Deadlock Models

5.4.1 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

5.4.2 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 —gues 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.

5.4.3 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. In a WFG,a vertex v is in a knot if for all $u :: u$ is reachable from $v : v$ is reachable from$u$. No paths originating from a knot shall have dead ends.

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.

5.5 Deadlock Detection Algorithms

5.5.1 Path-Pushing Algorithm

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.

5.5.2 Edge-Chasing

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. 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.

5.5.3 Diffusion Computation Algorithm

- ◇ 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.

5.5.4 Global state detection-based algorithms

exploit the following facts:

- ◇ a consistent snapshot of a distributed system can be obtained without freezing the underlying computation, and
- ◇ 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

5.5.5 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

5.6 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.

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.

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 P is self-stabilizing with respect to 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 starting from an arbitrary global state, S is guaranteed to reach a global state satisfying P within a finite number of state transitions.

6.3.1 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 distribution uniform
- ◇ Reducing the number of states in a token ring
- ◇ Shared memory models Mutual exclusion
- ◇ Costs of self-stabilization

6.3.2 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 demon, which arbitrarily decides the order which privileged machine is allowed to move.

A legitimate state must satisfy the following constraints:

- ◇ There must be at least one privilege in the system (liveness, no deadlock).
- ◇ Every from move from a legal state must again put the system into a legal state (closure).
- ◇ During an infinite execution, each machine should enjoy a privilege an infinite number of times (no starvation).
- ◇ Given any two legal states, there is a series of moves that change 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 us see how the above issues affect the design of a self- stabilization algorithm.

6.3.3 First Solution

Machine 0 is exceptional, and the other machines are identical.

6.3.4 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.

- **Bottom** machine

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

- **Top** machine

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

- Other machines

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

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

6.3.4.1 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, ..., 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.

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.

6.5 Costs of Self-Stabilization

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**.

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 $P \to R$.

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 consesu 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.

Consesus alorithm 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.

7.2 Mechanisms for Consesus

There are various ways for implementing consesus:

- ◇ Majority agreement: Paxos and Raft work by ensuring a majority of nodes agree on a value.
- ◇ Leader-based coordination: Sometimes protocols use a leader to drive agreement.
- ◇ 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

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. The system must continue to work, eventually deciding on a value despite failure or delays. In other words, we must not have a deadlock ☺.

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 Requirements

...

7.5 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.

7.5.1 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
   - ◇ **Prepare** - Proposer sends a Prepare request with a proposal number. Then the proposer chooses a Quorum of Acceptors and sends the Prepare message containing n to them.
   - ◇ **Promise** - Acceptors respond with a Promise if the proposal number n is higher than any they've previously seen. Otherwise it may not respond or send a negative message back.

2. Phase 2
   - ◇ **Accept** - Upon receiving a Promise from a majority of Acceptors, the Proposer sends an Accept request.
   - ◇ **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

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.

7.5.2 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

- ◇ Google chubby //TODO

7.6 Key Takeaways

Consesus is essential for coordination in distributed systems, and it is a fundamental problem in distributed computing.

Chapter 8

Raft

8.1 Logs 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

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!

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 implementer, and it was hard to get it right.

8.3 Raft

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.

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

8.3.1 Key points

- ◇ **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.

8.3.2 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.

8.3.3 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.

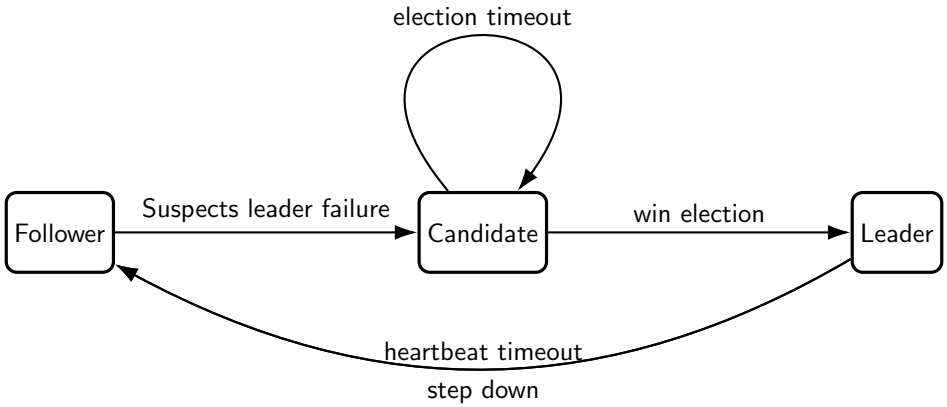

Figure 8.1: State diagram

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
- ◇ **Weak** Consistency - Reads may return stale data
- ◇ **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
- *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

9.3 Example Scenarios

9.3.1 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.

9.4 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
- ◇ User Partitioning
- ◇ Hierarchical Partitioning

9.5 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)?

9.6 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.*

10.1 Replication Fashions

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

10.1.1 Leaders and followers

10.1.2 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.
- **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 Challenges //TODO
- Leader failure: Failover -
- Challenges in Failover -

10.2.1 Implementing Replication Logs

- **Statement-based** replication - Replicate SQL write requests statements from the leader to the followers.
  We cannot ensure a fully consistent system, due to issues with non-deterministic functions, autoincrementing columns, and side effects.
  This was used in VoltDB and MySQL before version 5.1.
- **Write-ahead log (WAL)** shipping - Replicate the log of changes to the database from the leader to the followers.
  Still some problems here, because we assume that the leader and the followers are using the same database engine, same architecture an so on, so that the log is the same.

- ◇ **Logical log** replication - Replicate a logical representation of changes to the database from the leader to the followers.
  This is advantageous and flexible, but it is dischouraged by the fact that some expertise and coding are required. Instead, with the first two approaches there is no need to modify the database in any way.
- ◇ **Trigger-based** replication - Replicate changes to the database by using triggers.

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" ...

10.3.1 Read-after-write consistency

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

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

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).

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.

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.
With single-leader replication, the second writer blocks or aborts if first write is incomplete, having the user to retry the input.
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
- ◇ // TODO

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.

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.

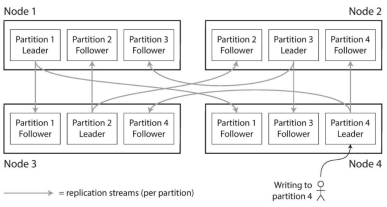

Figure 11.3: Leader-follower model

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 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.

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.

Note also that hash partitioning is very bad for range queries, as even similar data is spread randomly across the partitions.

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
  - *Reading* requires querying all partitions
- ◇ **Term**-based partitioning (global indexes)
  - A Global index covers data in all partitions.
  - Partitioning is based on the term ID, which is not the primary key index.
  -

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

11.3.1 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.

A simple intuition exploited by both Kademlia and Chord is to fix the granularity of the keyspace prior to partitioning.

In other words, having a fixed number of partitions may help to avoid data movement when nodes are added or removed.

11.3.2 Dynamic partitioning

Dynamic partitioning is a technique that allows the system to adapt to changes in the data distribution. This is done by monitoring the data distribution and rebalancing the data when necessary. Dynamic partitioning can be done in various ways, such as automatic rebalancing, manual rebalancing, and dynamic rebalancing.

- ◇ **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.

11.3.3 Routing

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. Routing can be done in various ways, such as static routing, dynamic routing, and consistent hashing.

The opposite approach is to query all partitions, which is very expensive, but may be necessary in some cases.

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

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.

12.1 Error Handling

Transactions hence provide all-or-nothing guarantees, simplyfing error handling, since partial failures are not to be managed.

12.2 Deeper into DBs

Most SQL DBs support transactions. NoSQL databases.

12.3 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.

12.3.1 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.

12.3.2 Single-Object and Multi-Object Transactions

If a transaction involves multiple objects, it is a multi-object transaction, which causes **performance** and **deadlock** issues.

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.

12.4 Avoiding Transactions

12.4.1 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.

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 allowing to avoid "paying" for transactions.

Definition 12.4 (Non-repeatable Read) *A **non-repeatable read** occurs when a transaction reads the same data twice and gets different results.*

Non-repeatable reads are possible in Read Committed isolation level,

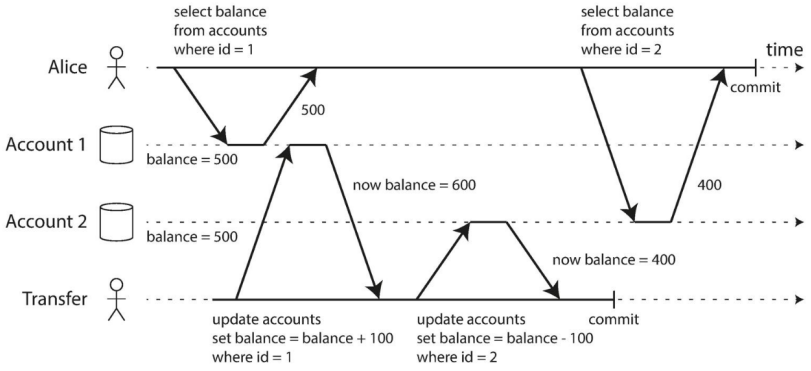

Figure 12.1: Read committed issue: Alice might think that her whole balance is 900, whilst it is 1000.

12.4.2 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 at databases some techniques for locking the database, using locks or atomic operations. However it is costful

12.5 Write Skew and Phantoms

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.*

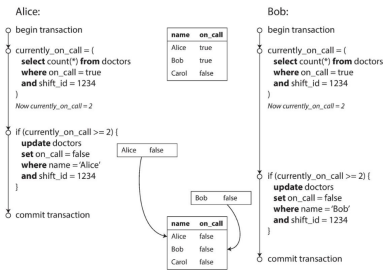

Figure 12.2: Both doctors read that `currently_on_call >= 2` and "leave the hospital", causing no doctor to be on call ☹.

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

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.

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

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`

$$Map(key_a), value_1) \rightarrow list(key_b, value_2)$$

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`

$$Reduce(key_b, list(value_2)) \rightarrow list((key_c, value_3))$$

`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.

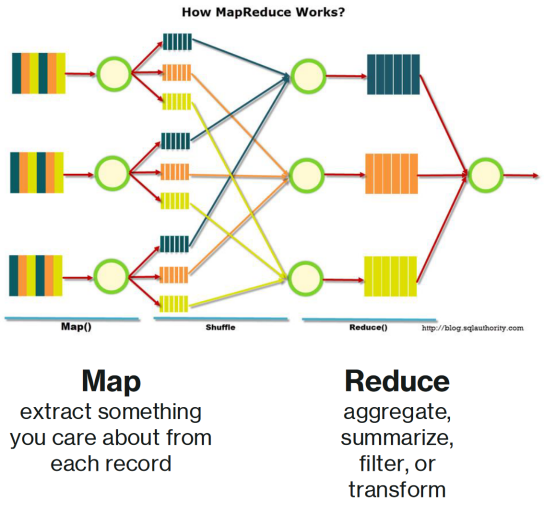

Figure 14.1: MapReduce overview

14.2 Instatiating MapReduce

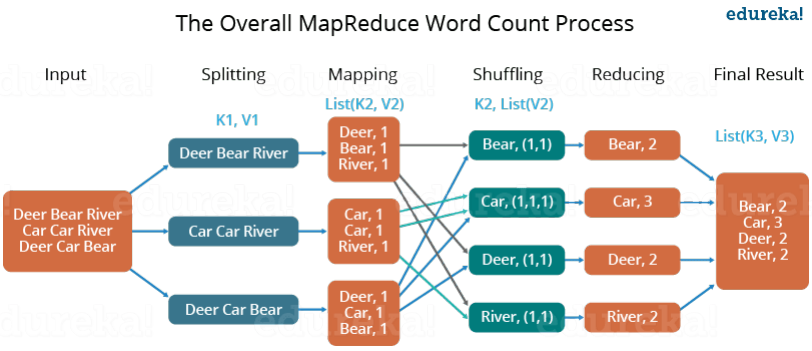

Figure 14.2: MapReduce WordCount example

- One `JobTracker` to 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

If a `TaskTracker` fails or times out, that part of the job is rescheduled.

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

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.

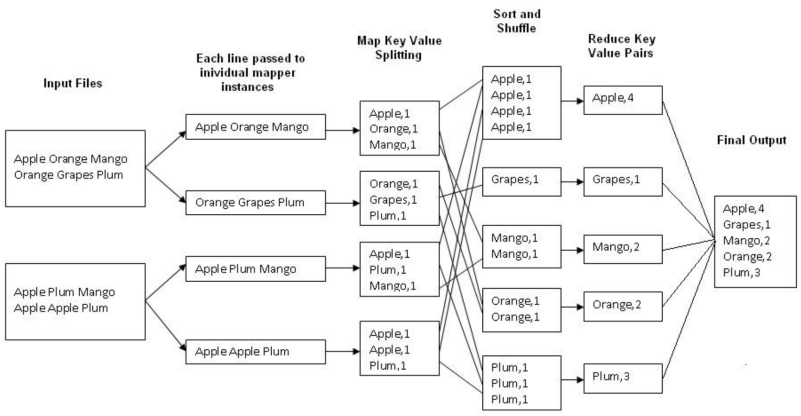

Figure 14.3: Hadoop WordCount example, equivalent to *Hello world!*

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.

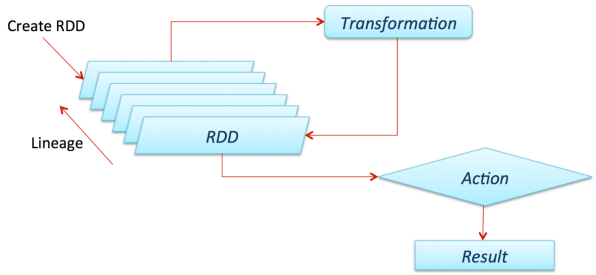

Figure 14.4: RDD schema

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** - 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
# we can instruct Spark to persist the
     RDD
# lineLengths.persist() # place before
     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 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 **??**. 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.

15.1 Towards the Actor Model

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

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).

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.

15.1.1 MapReduce - Why not?

The key issue is that MapReduce does not fit well scenarios where we don't know the number of steps in advance, or where we don't know where the computation will take place. In general MapReduce is not a good fit for heterogeneous environments.

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

15.2 BSP - Bulk Synchronous Parallel model

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.

15.3 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.

15.4 The Actor Model - Key Concepts

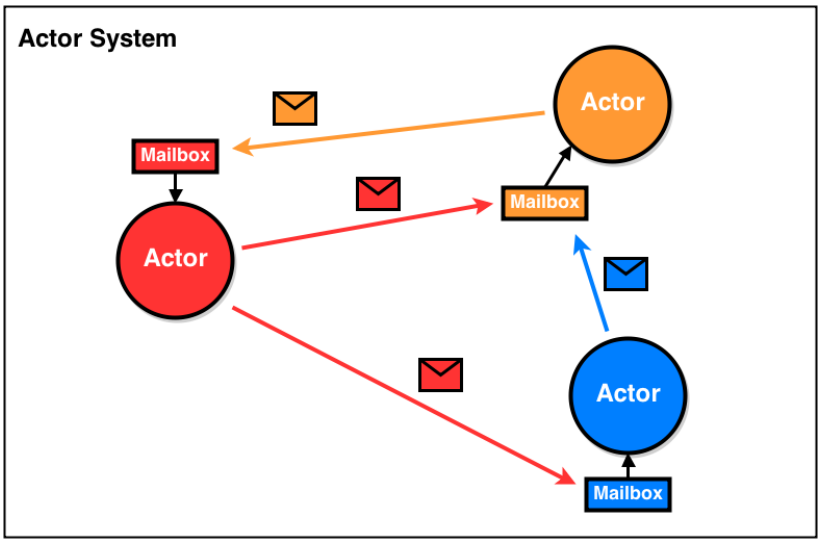

Figure 15.1: Actor model schema

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.

15.4.1 Indeterminacy and quasi-commutativity

Messages are delivered asynchronously, and the order of delivery is not guaranteed, hence, the actor model is **not** completely **deterministic**.

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.

15.4.2 Fault tolerance

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.

15.4.3 Challenges