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.

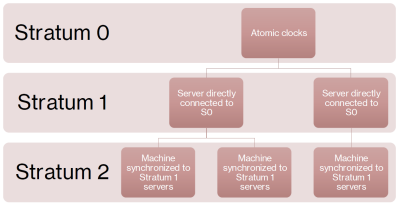

Figure 3.2: NTP Stratum architecture

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.

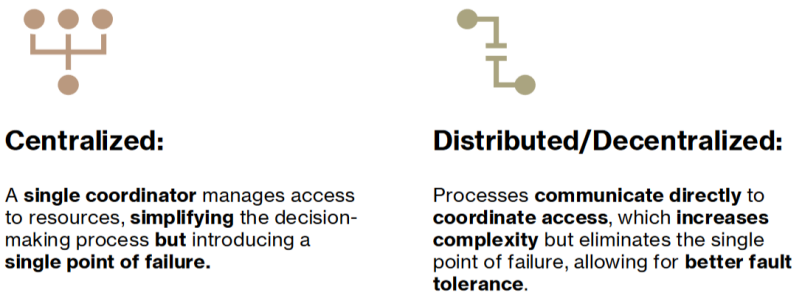

Figure 4.1: Mutual exclusion - Centralized vs Distributed

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

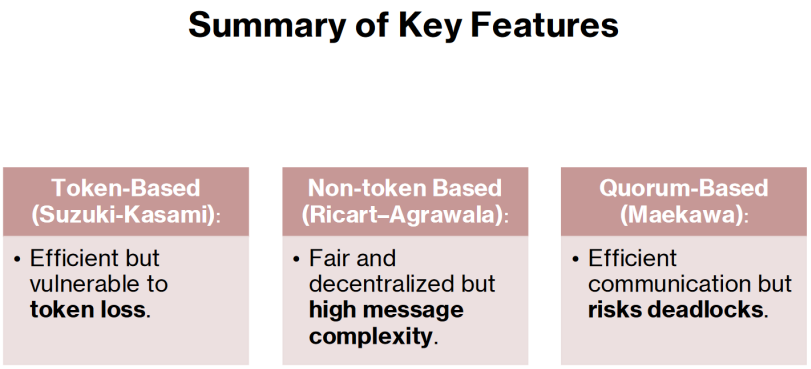

Figure 4.3: Summary of the Key features

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