Distributed Systems Cheatsheet

Edoardo Riggio

January 12, 2023

Distributed Systems - S.A. 2022
Software and Data Engineering
Università della Svizzera Italiana, Lugano

Contents

1 Introduction

With the advent in the mid 1980's of 16-, 32-, and 64-bit CPUs – as well as the invention of high-speed computer networks, made it possible for the creation of large numbers of geographically-dispersed networks of computers. These are known as **distributed systems**.

1.1 Definition

A distributed system is a collection of independent computers that appear to the user as a single coherent system.

Each computing element of these massive systems are able to behave independently from one another. A computing element is often referred to as a **node**. This node can either be a hardware device or a software process.

1.2 Consequences

Some of the main negative consequences that arise when dealing with distributed systems are the following:

- **Concurrency**
  This happens when several processes try to read and write on a shared storage service.
- **Absence of a global clock**
  Each node will have its own notion of time. This means that there is no common reference of time between the nodes.
- **Failure independency** The failure of a node can make another node unusable.

1.3 Challenges

Some of the challenges that arise when dealing with distributed systems are outlined in the following sections.

1.3.1 Openness

The services are offered according to standard rules. These rules describe both the syntax and semantics of such services. The standard rules of distributed systems are called **COBRA**(Common Object Request Broker Architecture).

1.3.2 Scalability

Scalability can be with respect to the **system size** – which would mean adding more users to the system; to the **geography** – which would deal with users lying far apart from one another; and to **administration** – which would deal

with the complexity to manage an increasing system.

Some scalability techniques are the following:

- **Hiding communication latencies**
  This is important for **geographical scalability**. Asynchronous communications can be used to reduce the waiting time of the users. This can be achieved with the use of batch processing and parallel applications.
- **Distribution**
  Components are split into parts and spread across the system. An example of this would be the DNS, where we have a tree of domains divided into non-overlapping zones. Furthermore, the name in a zone is handled by a single name service.
- **Replication**
  This can be use to increase both **availability** and **performance**, as well as reduce **latency**. Moreover, caching can be used as a form of replication, but is typically done on-demand.

1.3.3 Transparency

Transparency is the ability of a system to hide some of its characteristics or errors to the user. There exist several different forms of transparency:

- **Access transparency**
  Hide the differences in data representation and machine architecture.
- **Location transparency**
  Users cannot tell where a resource is physically located.
- **Relocation transparency**
  Even if the entire service was moved from one data center to the other, the user wouldn't be able to tell.
- **Migration transparency**
  Moving processes and resources initiated by users, without affecting any ongoing communication and operation.
- **Replication transparency**
  Hide the existence of multiple replicas of one resource.
- **Concurrency transparency**
  Each user is not going to notice if another user is making use of the same resource.
- **Failure transparency**
  The user or application does not notice that some piece of the system fails to work properly. The system is then able to automatically recover from the failure.

1.4 Types of Distributed Systems

There are three main types of distributed systems: **distributed computing systems**, **distributed information systems**, and **distributed pervasive systems**.

1.4.1 Distributed Computing Systems

These systems aim at high-performance computing tasks. These systems can be part of two subtypes:

- **Cluster Computing**
  All the resourced are located in a local-area network, and are using the same OS. In addition, they also have a common administrative domain.
- **Grid Computing**
  This is a "federation" of computer systems. Such systems may have different administrative domains, different hardware, software... Such systems are used to make collaboration between organisations feasible.

1.4.2 Distributed Information Systems

These systems are based on transactions. These transactions are used to make systems communicate between themselves. Transactions follow the **ACID** properties:

- **Availability**
  To the user, the transaction happens indivisibly.
- **Consistency**
  The transaction does not violate system invariants.
- **Isolation**
  Concurrent transactions do not interfere with each other.
- **Durability**
  Once a transaction commits, the changes are permanent.

The application components of each node communicate directly with each other.

1.4.3 Distributed Pervasive Systems

These systems are composed by mobile and embedded computing devices. This means that pervasive systems need to have the following characteristics:

- Embrace contextual changes
- Encourage ad-hoc composition
- Recognise sharing as the default

Some examples of such architectures are home systems, electronic health care systems, and sensor networks.

2 Architectures

In a distributed system there are two types of architectures:

- **Software Architectures**
  How software components are organised and interact between each other.
- **System Architectures**
  How software components are instantiated on real machines.

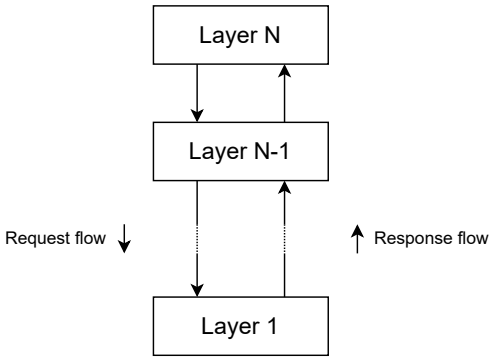
2.1 Software Architectures

Software architectures are based on **components**, which are modular units with well-defined interfaces. Software architectures can be of several different types:

- **Layered architectures**
- **Object-based architectures**
- **Data-centred architectures**
- **Event-based architectures**

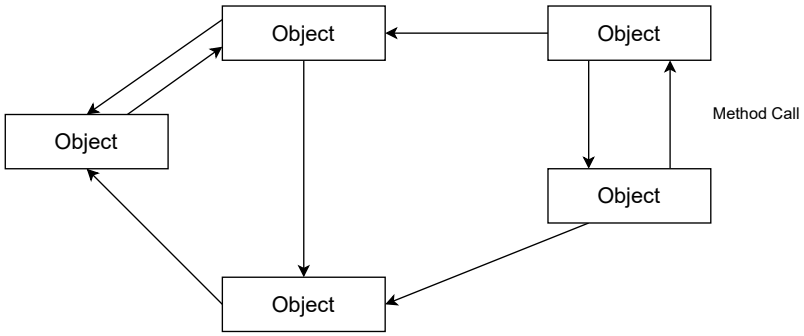
2.1.1 Layered Architectures

In this kind of architecture, the components at layer L_i can call components in layer L_{i-1}, but not components in layer L_{i+1}.



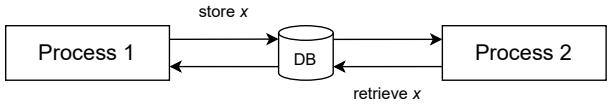
2.1.2 Object-Based Architectures

Each object is a component connected through a remote procedure call mechanism.



2.1.3 Data-Centred Architectures

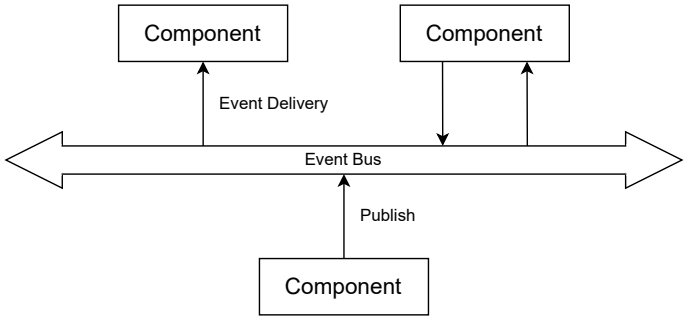
The processes communicate through a common repository. This repository can be, for example, a shared distributed file system or a database.



2.1.4 Event-Based Architectures

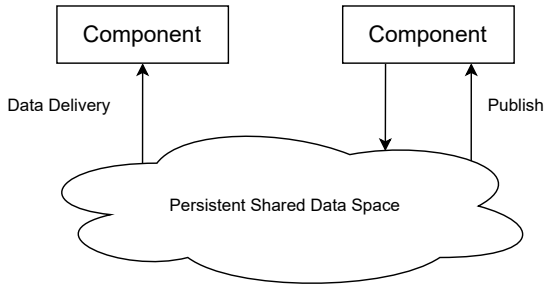
The processes communicate through the propagation of events (**publish-subscribe system**). The middleware ensures that the process which subscribes to that events will receive them.

Processes are loosely coupled, this means that they do not to refer to each other.



2.1.5 Shared Data-Space Architecture

These architectures are similar to data-centred and event-based architectures.



2.2 System Architectures

System Architectures describe what software components are used, where to place each component, and how the components interact with each other.

There are two types of system architectures:

- **Centralised architectures**
- **Decentralised architectures**

2.3 Centralised Architecture

Centralised architectures follow the Client-Server model. The **server** implements some services, while the **client** requests services and waits for replies.

The communication between server and client is **connectionless**. This means that it is highly efficient, but it is more complex to handle transmission failures (UDP).

Server and client may also use another way to communicate. This protocol is connection-oriented, relatively low performance, but more reliable. This is because the node makes a request and receives a reply back in the same connection.

Centralised architectures can be either **single-** or **multi-layered**. Multi-layered architectures enable to divide concerns, meaning that clients can contain only the program implementing the user-interface level (or only part of it). Moreover, architectures can also be **multi-tiered**.

2.4 Decentralised Architectures

Decentralised architectures can be either vertically or horizontally distributed.

- **Vertical Distribution**
  It can be achieved by placing logically different components on different machines. An example of this is a three-tiered application.

- **Horizontal Distribution**
  Both clients and servers are physically split up into logically equivalent parts. Each part operates on its own share of the dataset. An example of this is a peer-to-peer system.

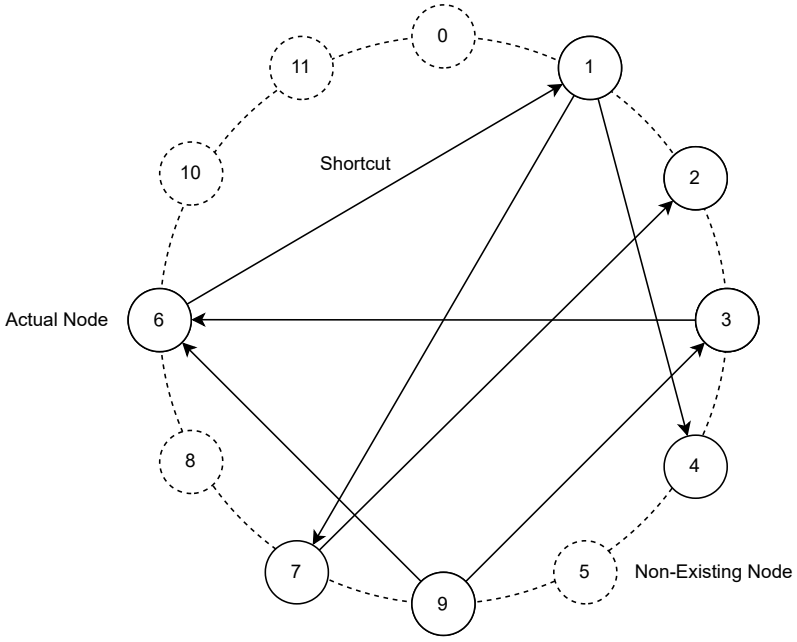
2.4.1 Structured Peer-to-Peer Architectures

Structured peer-to-peer architectures are deterministic procedures used to build an **overlay network**. Here nodes are processes, and links are possible communication channels.

In the case of a structured peer-to-peer architecture, we use **Distributed Hash Tables** (DHT). Each data item that is maintained by the system, is uniquely associated with a key, and this key is subsequently used as an index. Moreover, to each node of the system is assigned an identifier. Each node is made responsible for storing data associated with a specific subset of keys.

Any node can be asked to look up a given key, which boils down to efficiently routing that lookup request to the node responsible for storing the data associated with the given key.

An example of a structural peer-to-peer architecture is a **Chord**, which is depicted below.

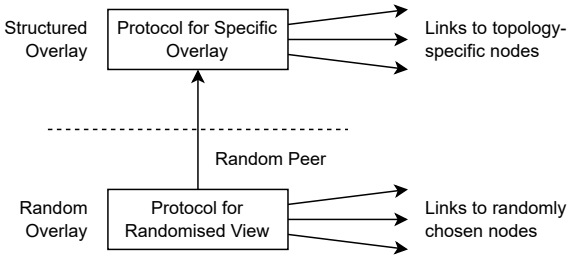


2.4.2 Unstructured Peer-to-Peer Architectures

In this type of architecture, the overlay network is built using a randomised procedure. Each node in the network has a list of neighbours, constructed in a random way. This results in a random graph.

In these kinds of systems, when a node joins the network, it often contacts a well-known node to obtain a starting list of other peers in the system. Moreover, the nodes generally change their local list almost continuously.

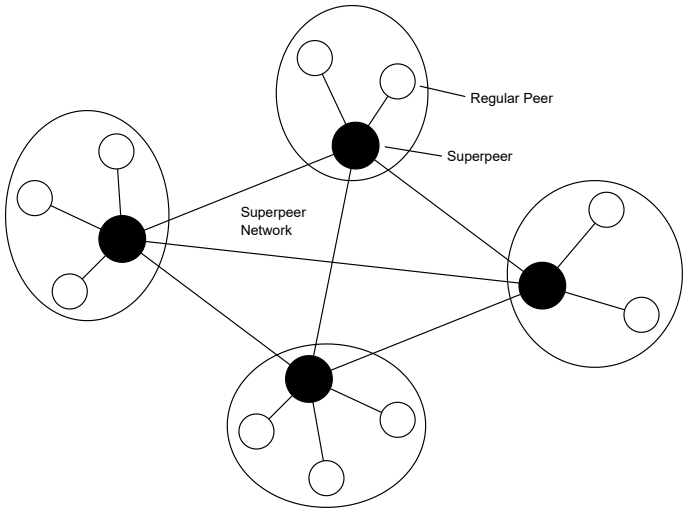
Unlike in structured peer-to-peer architectures, looking up data cannot follow a predetermined route. Instead, in this case we need to resort to searching for data. This can be done in different ways, such as **flooding**, **random walks**, or **policy-based** search methods.



2.5 Super Peer

Brokers are services that collect data on resource usage and availability for a number of nodes that are in each other's proximity. This allows to quickly select the node with sufficient resources.

Nodes that act as brokers are called **super peers**. Super peers organise themselves into a structured peer-to-peer network, leading to a hierarchical organisation of nodes. The nodes that connect to these super peers are known as **weak peers**.



2.6 Middleware

Middleware provides a degree of distribution transparency. They can either follow an object-based architectural style, or an event-based architectural style.

2.6.1 Interceptors

They offer a means to adapt the middleware. Interceptors break the usual flow of control and allow other application-specific code to be executed. Moreover, they are used to improve software management.

2.6.2 Wrappers and Adapters

Wrappers and adapters provide a similar interface to the original abstraction, and implement additional logic before or after invoking the original interface.

2.6.3 Proxies and Stubs

They provide – as in the case of wrappers and adapters – a similar interface as the original abstraction. However, in this case proxies and stubs usually own components rather than extending previously defined ones.

3 Processes

3.1 Introduction

Virtual processors are created by the OS. A process is a program executed on one of these virtual processors. Each processor has a process table in which several different values relative to the process are stored.

Concurrent processes are protected from each other by making the sharing of CPU and hardware resources transparent and using special hardware.

Every time a new process is created, the OS must create an independent address space as well. This is known as **context switching**, and it's really expensive.

3.2 Threads

A thread is a lightweight process that can run inside of a process. Threads share the same memory space. For this reason, context switching between threads is less expensive than between processes.

Threads are used in distributed systems mainly for making communication calls non blocking.

In the case of servers with **single-threaded models**, one thread receives the request and executes it rapidly. This is known as sequential processing.

On the other hand, in the case of servers with **finite state machine models**, one thread executes the requests and the replies. Blocking system calls are replaced by non-blocking ones, and multiple requests can be handled in parallel.

3.3 Virtualisation

The role of virtualisation – in distributed systems – is to extend or replace existing interfaces and mimic the behaviour of other systems.

The main reasons of virtualisation are the following:

- Allow legacy software to run on new mainframe hardware
- Porting legacy interfaces to new platforms
- Isolating systems running on the same server

3.4 Superservers

A superserver is a kind of server that listens on many ports and fork a process to execute a service.

3.5 Stateless vs Stateful Servers

A server is said to be **stateless** if it does not keep any information about the state of the client.

On the other hand, a server is said to be **stateful** if it maintains persistent

information about the clients. This solution is better in terms of performance (it caches data of the users), but can be problematic in the case that the server crashes or experience other memory-related issues.

3.6 Server Clusters

A server cluster is a collection of machines connected through a network. Such clusters have three tiers:

1. **Logical Switch**
   It receives client requests and route them to the servers.
2. **Application Servers**
   Low-end servers are used when the storage is the bottleneck of the system. Otherwise, high-end servers are used when the service deals with computationally expensive computations.
3. **Data-Processing Servers**
   The databases.

4 Communication

In layered protocols, process A con communicate with process B by building a message in its address space. After doing so, a system call is used to send the message from A to B.

4.1 OSI Model

The **Open System Interconnection** (OSI) **Reference Model** is an ISO standard. This type of model is never widely used or implemented.

The OSI model makes use of connection-oriented protocols. Both the sender and the receiver explicitly establish a connection, they communicate, and finally explicitly terminate the connection.

The OSI model is divided into the following parts:

- Application layer
- Presentation layer
- Session layer
- Transport layer
- Network layer
- Data link layer

- Physical layer

In the following sections, we will have a bottom-up look at the different layers.

4.1.1 Physical Layer

This layer deals with how to send bits from one end to the other. It standardises electrical, mechanical, and signalling interfaces. finally, it does not handle errors.

4.1.2 Data Link Layer

The data link layer groups the bits into frames and ensures their correct transmission. It uses checksums to verify the integrity of the data frames.

4.1.3 Network Layer

It routes the messages from source to destination. To do so, it needs a routing protocol. The most common ones are:

- **TCP**
  Transmission Control Protocol (TCP) is both reliable and connection-oriented.
- **UDP**
  Universal Datagram Protocol (UDP) is both unreliable and connectionless.

4.1.4 Session Layer

The session layer is an enhanced version of the transport layer – but rarely supported.

4.1.5 Presentation Layer

This layer simplifies the communication between machines with different internal data representation.

4.2 Middleware Protocols

This type of protocols usually resides at the application layer. The middleware protocols can have several different roles, such as:

- Authentication
- Commit protocols
- Distributed locking protocols

4.3 Types of Communication

There are several different ways in which a client and a server can communicate:

- **Persistent Communication**
  The message is stored by the communication middleware until it can be delivered to the receiver.
- **Transient Communication**
  The message is delivered only if both sender and receiver are executing.
- **Asynchronous Communication**
  The sender continues immediately after submitting the message. This message is temporarily stored inside of the middleware.
- **Synchronous Communication**
  the server is blocked until the message has been received by the receiver.

4.4 Remote Procedure Call

A **Remote Procedure Call** (RPC) is a simple communication mechanism, and is more natural than the send/receive primitives.

4.4.1 Basic RPC Operation

The RPC first makes itself look like a local procedure call. When a read operation is performed, a client stub is called. The client stub pack all the parameters into a message and sends it to the server.

Both the calling and the receiving procedures are not aware of the distribution of the system.

4.5 Message-Oriented Communication

Message-oriented transient communication uses **Berkeley sockets**. These sockets are communication endpoints used by applications to read are write. The process of the socket is the following:

Server:

1. Create the socket
2. Bind the socket to a port
3. Listen on the opened port
4. Accept upon communication request
5. Communicate with the client

6. Close the socket

Client:

1. Create the socket
2. Request connection to the server
3. Communicate with the server
4. Close the socket

4.5.1 Message-Queuing Models

The applications communicate between one another by inserting messages into queues. Each application has its own private queue, and both the sender and receiver do not have to necessarily be active at the same time.

There is guarantee of delivery at destination, but the time at which it reaches its destination cannot be known.

Queue managers interact with both relays and applications. The relays forward the messages to other queue managers. The message is delivered as follows:

1. Queue managers interact with both application and relays
2. The relays forward the messages to other queue managers
3. The overlay network composes the sender, the destination, and the relays

4.5.2 Message Brokers

A message broker acts as an application-level gateway. They do so by integrating existing and new applications into a single coherent system. Moreover, they convert the incoming messages ti formats that can be understood by the destination application.

4.6 Multicast Communication

Application-level multicasting has the following characteristics:

- Support for sending data to multiple receivers
- Nodes reorganise themselves into an overlay network at the application level

There are two main types of overlays:

- **Tree-Based Overlay**
  There is a single path between every pair of nodes.

- **Mesh-Based Overlay**
  There are multiple paths between every two pair of nodes. Although it is harder to find the best path between a pair of nodes, these overlay networks are generally more robust.

5 Naming

It is impossible to memorise IPv4 and IPv6 addresses to reach websites. For this reason. we use **naming**. Naming can be of three types:

- Flat
- Structured
- Attribute-Based

5.1 Flat Naming

Flat names are composed of names, identifiers, and addresses. They can be described as follows:

- **Name**
  It is "human-friendly" and it is not necessarily unique or used for identifying.
- **Identifier**
  It uniquely identifies an entity.
- **Address**
  It is not "human-friendly" and is used for low-level naming of access points.

5.1.1 Forwarding Pointers

When an entity is moved from point A to point B, it leaves a reference in point A pointing to B. The requests to A must be forwarded to B in a transparent manner.

There are some drawback to forwarding:

- Fast moving entities leave a long chain of entities
- Each intermediate location needs to allocate resources for the forwarding process
- Any missing step in the chain makes the whole chain unusable

5.1.2 Home-Based Approaches

In this case, a home location is used to keep track of the current location. By doing so, IPv6 addresses become identifiers. The following is the procedure of home-based approaches:

1. Mobile devices requests a local temporary address
2. The address is registered with the home agent
3. The home agent forwards the packets to the mobile device
4. The home agent tells the sender the new temporary location

There are some drawbacks that come with home-based approaches, such as:

- Increased latency
- Home agent must always be available
- Permanent relocations must be handled

A possible solution to all of these problems comes with registering home locations using a **Domain Name Server** (DNS).

5.1.3 Distributed Hash Tables

We assign an m-bit identifier space to each node of the system. To resolve the successor node k, we use a **finger table**.

$$FT_p[i] = succ(p + 2^{i-1})$$

Where p is the node, and i is the number of nodes succeeding p. The complexity of the successor computation is:

$$O(\log(nodes))$$

A drawback of this method is that nodes joining and leaving force the finger table to continuously update. This issue can be solved by making the nodes run the function `succ(k)` in the background.

5.1.4 Hierarchical Approaches

Each **root node** keeps track only of the location of the directory nodes of the next lower-level subdomains. On the other hand, each **leaf node** contains the location of the entities.

A lookup request is performed bottom-up. This means that the request is forwarded amongst the children of the directory node. Afterwards, the request goes up the hierarchy.

5.2 Structured Naming

5.2.1 Name Spaces

Name spaces can be represented as labeled, directed graphs with directory and leaf nodes.

A **pathname** is a sequence of labels corresponding to the edges in that path.

5.2.2 Linking and Mounting

A **symbolic link** is a link between a node and an absolute path. This is particular since normally nodes are linked to files – in the case of a OS's file system.

Mounting a foreign name space in a distributed system is a type of symbolic link. For this symbolic link to work it requires:

- Name of an access protocol
- Name of a server
- Name of the mounting point in the foreign name space

5.2.3 Name Space Distribution

Generally, large-scale name services are organised into **organisational layers**. The layers are the following:

- **Global Layer**
  It is composed by the root name and its children (very stable)
- **Administrational Layer**
  The nodes are managed by a single organisation (somewhat stable)
- **Managerial Layer**
  Nodes are managed locally (may change frequently)

5.2.4 Name Resolution

All clients have access to a name resolver. This name resolution can happen in two ways:

- **Iteratively**
  The name resolver performs each step of the name resolution.
- **Recursively**
  The name resolver delegates the name resolution to the root server.

Each one of the previously described methods has its own drawbacks. In the case of the **iterative** method, it has high communication costs and poor caching capabilities. In the case of the **recursive** method, it has an increased load on each server of the chain, as well as increased latency for non-cached requests.

5.2.5 DNS Name Space

The DNS is a hierarchically organised name space. Some of its characteristics are:

- DNS root has no name
- Each subtree is a domain
- The path to the subtree root node is the domain name
- All nodes contain records
- One server is responsible for one zone

5.3 Attribute-Based Naming

Attribute-based naming systems are also known as directory services. In this case, an entity is described in terms of an `(attribute, value)` pair. A set of these attributes is used for searching.

5.3.1 LDAP

Lightweight Directory Access Protocol (LDAP) has the following characteristics:

- Each service contains a number of directory entries
- Each entry is composed of a number of `(attribute, value)` pairs
- Each attribute has an associated type
- Attributes can be single- or multi-valued
- The collection of all entries for a service is called a **Directory Information Base** (DIB)

In a DIB, each entry has a **Relative Distinguished Name** (RDN). This name is globally unique.

RDNs can be used a globally unique names – which is the same thing that happens in DNS. Moreover, the RDNs establish a hierarchy of entries – which is known as the **Directory Information Tree** (DIT).

6 Synchronisation

6.1 Clock Synchronisation

In a **centralised system**, the is only one clock. This makes time unambiguous. On the other hand, in **distributed systems**, there are multiple different clocks, one for each system. In this case there is no global time. Such a lack of global time makes agreements on time non trivial.

6.2 Physical Clocks

A **computer timer** (called RTC – Real Time Clock) is composed of a quartz crystal. When we apply an electrical current to this crystal, its molecular structure distorts before going back to normal. The oscillation produced by the clock defines its frequency (**clock ticks**). Being quartz crystals physical clocks, no two clocks are the same. Moreover, different clocks will tend to diverge over time (**clock skew**).

In **real-time systems**, on the other hand, we can use universal clocks such as **UTC** (Coordinated Universal Time), or **GPS** (Global Positioning System). These two solutions are used to synchronise clocks with real-time clocks.

6.3 Clock Synchronisation Algorithms

6.3.1 System Model

Each machine has a timer that causes an interrupt H times a second. Every time we have an interrupt, the handler adds 1 second to the **software clock** – which is handled by the OS. For a **clock value** C and **UTC time** t, the clock of **machine** p can be defined as:

$$C_p(t)$$

Ideally, the clock should be equal to the UTC time.

$$C_p(t) = t$$
$$C'_p(t) = \frac{\partial C}{\partial t} = 1$$

Where $C'_p(t)$ is the **frequency of p's clock at time** t, $1 - C'_p(t)$ is the **clock skew**, and $C_p(t) - t$ is the **offset relative to a specific time** t. Following are three types of clocks defined by the value of $C'_p(t)$.

$$\begin{cases} \frac{\partial C}{\partial t} < 1 & \text{slow clock} \\ \frac{\partial C}{\partial t} = 1 & \text{perfect clock} \\ \frac{\partial C}{\partial t} > 1 & \text{fast clock} \end{cases}$$

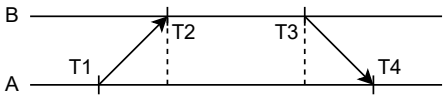
When two clocks are drifting from UTC in opposite directions, at a certain time δ after their last synchronisation, they will differ by $2\rho\delta$ seconds from each other. Where ρ is the **maximum drift time**. In order for the two clocks not to differ by more than δ seconds, they must be synchronised at lest every

$$\frac{\delta}{2\rho}$$

Seconds. As in the previous case, δ is the **time after the clocks' last synchronisation**, and ρ is the **maximum drift rate**.

6.3.2 Network Time Protocol

In this algorithm, the clients contact a time server, which stores the accurate current time. Let's suppose that client A requests the time from server B.



Then this will be the procedure:

1. A sends a request to B with timestamp T_1;
2. B records T_2 from its own clock;
3. B returns a response containing both T_1 and T_2;
4. A records the value T_4 from its own clock.

If we assume similar propagation delays then $T_2 - T_1 \approx T_4 - T_3$. This makes client A able to compute the offset between the two clocks as such:

$$\Theta = T_3 + \frac{(T_2 - T_1) + (T_4 - T_3)}{2} - T_4$$

$$= T_3 + \frac{\Delta T_{req} + \Delta T_{res}}{2} - T_4$$

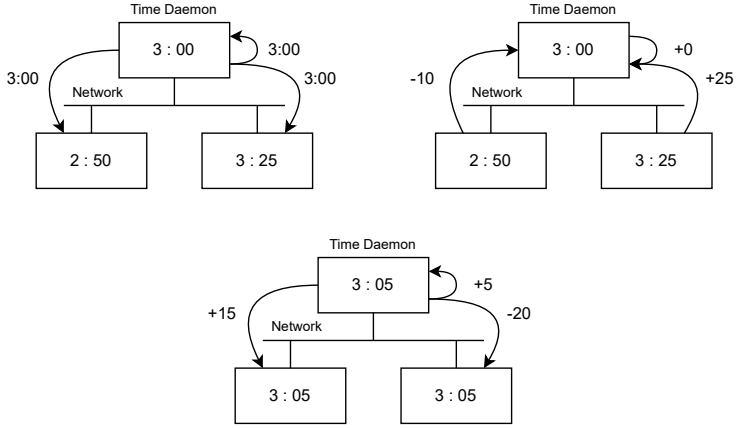
If $\Theta \neq 0$, then:

$$C_A := C_A + \Theta$$

Where C_A is A**'s clock** and Θ is the **offset between the clocks**.

6.3.3 Berkeley Algorithm

While in the NTP (Network Time Protocol) the time server is passive, in **Berkeley UNIX** the time server polls every machine periodically. Based on the client's response, the time server computes the averages and tells the client to update their timer. The time daemon – which is the time of the server itself – is updated manually.

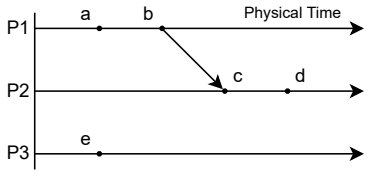


6.4 Logical Clocks

Many applications need to agree on a current time. This time, though, does not need to necessarily match real time. In addition, if two processes don't interact with each other, it's not necessary for their clocks to be synchronised.

6.4.1 Lamport's Logical Clocks

Given the **happened-before** relation depicted by $\rightarrow$, we can say that $a \rightarrow b$ means that event a comes before event b. The happens-before relation is **transitive**, this means that if we have both $a \rightarrow b$ and $b \rightarrow a$, then we can say that events a and b are concurrent. Given the following diagram:



We can define some other properties:

- If $a \rightarrow b$, then $b \nrightarrow a$;
- If $a \nrightarrow b$, and $e \nrightarrow a$, then $a \parallel e$;
- Even if $b \rightarrow d$, this does not imply causality.

Each process P_i keeps a Lamport timestamp L_i. A timestamp of event e in P_i is defined as:

$$T_{P_i} = L_i(e)$$

Where T is the **timestamp**, P_i is the **i-th process**, L_i is the **timestamp function**, and e is the **event's name**.

When a process updates its local Lamport clock and wants to transmit its value in a message, then it must follow certain rules, namely:

- The Lamport timer L_i must be incremented before each event is issued by the process:
$$P_i : L_i = L_i + 1$$
- When process P_i sends a message m, the value $t = L_i$ needs to piggyback on message m – this means that the timestamp must be sent together with the message;
- Upon receiving (m, t) – where m is the **message** and t is the **timestamp** – process P_j will compute:
$$L_j = \max(L_j, t)$$
And applies rule 1 before timestamping the next event – which will be receive(m).

From the rules above, we can say that

$$e \to e' \Rightarrow L(e) < L(e')$$

Where e and e' are the **two events**, and $L(e)$ and $L(e')$ are the **Lamport timestamps of those two events**.

6.4.2 Totally Ordered Logical Clocks

Since we sometimes need to have a total order of events, Lamport timestamps are not sufficient. The following is an extension of Lamport logical clocks.

Let e_i be an event in P_i with timestamp T_i, and let e_j be and event in P_j with timestamp T_j. We now define two **global timestamps** as (Ti, i) for P_i and (T_j, j) for P_j. The following

$$(T_i, i) < (T_j, j)$$

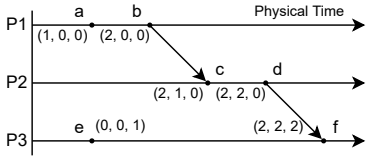
Will hold iff either $T_i < T_j$, or $T_i = T_j$ and $i > j$.

6.4.3 Vector Clocks

A key property of vector clocks is that:

$$e \to e' \Leftrightarrow L(e) < L(e')$$

Given an array of N processes, each process P_i keeps its own vector clock V_i. The vector clock is used to timestamp local events. Every process sends the vector's timestamp together with the message to the other process.



The vector timestamps in this representation are compared as follows:

- $v = v'$ iff $V[j] = V'[j]$ for $j = 1, 2, 3, \ldots, N$
- $v \leq v'$ iff $V[j] \leq V'[j]$ for $j = 1, 2, 3, \ldots, N$
- $v < v'$ iff $V \leq V'$ and $V = V'$

As with Lamport clocks, also vector clocks have rules for updating the vector. These rules are:

1. Initially the vector clock is as follows:

$$V_i[j] = 0 \qquad \text{for } j = 1, 2, 3, \ldots, N$$

2. Before process P_i timestamps an event, it sets

$$V_i[i] = V_i[i] + 1$$

3. Process P_i then needs to include timestamp $t = V_i$ in every message it sends;

4. When P_i receives a timestamp t in a message, it sets

$$V_i[j] = \max(V_i[j], t[j]) \qquad \text{for } j = 1, 2, 3, \ldots, N$$

   This operation is called **merge**.

Where $V_i[j]$ is the **number of events process P_i has timestamped**, and $V_i[j]$ with $i \neq j$ is the **number of events at process P_j that have potentially effected P_i**.

6.5 Mutual Exclusion

Let's assume that we have N processes named P_i, that processes do not share variables, that the system is asynchronous, and that there are no process failures. Given these assumptions, we have an application-level interface composed by the following functions:

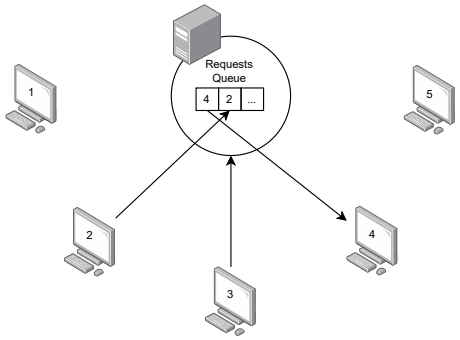
- `enter()`
- `resourceAccess()`
- `exit()`

Some more requirements are needed for the mutual exclusion, namely:

1. **Safety**
   At most one process may execute in the critical section at a time.
2. **Liveness**
   Requests to enter the critical section will eventually succeed. No starvation or deadlocks are allowed.
3. **Ordering**
   If one request to enter the critical section happened before another, then entry to the critical section is granted in that order.

6.5.1 Central Server Algorithm

in this case, a central server handles the requests to enter the critical section by saving each request in a FIFO queue.



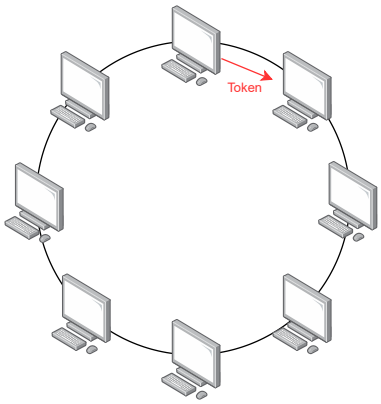
The actions performed in the image above are the following:

1. Process P_2 requests the token to access the critical section;
2. Process P_3 releases its token for access to the critical section;
3. The server grants access to P_4 to enter the critical section.

In absence of failures, ME requisites 1 and 2 are respected. ME requisite 3, however, is not respected. Moreover, he central server may become the bottleneck of the whole system in case of heavy loads. The server is also a single-point-of-failure, this means that is the server fails, the whole system shuts down.

6.5.2 Ring-Based Algorithm

The processes are now arranged in a logical ring. These processes circulate a token around the ring. As before, the token is needed by the process to access the critical section. If one of the processes has the token, but does not need to enter the critical section, it sends the token to the neighbouring process. This token is passed to the neighbouring process even when the process exits from the critical section.



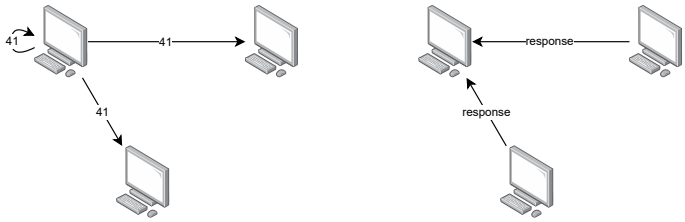
ME requisites 1 and 2 are respected, but requisite 3 is not respected. Moreover, bandwidth is consumed even if no process requires access to the critical section.

6.5.3 Multicast and Logical Clock Algorithm

The idea of this algorithm is the following:

1. The process that requires access to the critical section multicasts a request message to all the other processes;
2. If a response is received by the process from all of the other processes, then access to the critical section is granted.

We can have two possible scenarios.



Here, to the process multicasting the request is granted access to the critical section. The reason is that both of the other processes sent a response message back to the requesting process.

Instead, in this case, the to the process multicasting the request is not granted access to the critical section. The reason is that only one of the other two processes has sent a response message back to the requesting process.

This algorithm guarantees ME requisites 1, 2, and 3. Moreover, each process P_i has its own Lamport clock. The messages request entry at time (T, P_i), where T is the **sender's timestamp**, and P_i is the **sender's id**.

The problem with this algorithm is that every time a process P_i needs access to the critical section, $2(N-1)$ messages need to be exchanged – for both multicast and replies. Moreover, up to $N-1$ messages are sent when a process exits the critical section – since it needs to communicate it to the rest of the processes.

6.5.4 Maekawa's Voting Algorithm

Access to the critical section does not need to permission from all of the processes, but from a subset of them is enough – only if the subsets overlap.

This algorithm is very similar to the previous. In this case, though, we need for a sufficient number of votes – not all. To achieve this, to each process is assigned a voting set V_i.

ME requirement 1 is respected, but the algorithm is deadlock-prone. By queuing outstanding requests in a happens-before order, then both ME requirements 2 and 3 are also respected.

By using this algorithm, we will have $\approx 2\sqrt{N}$ messages when asking to join the critical section, and $\approx \sqrt{N}$ messages when exiting the critical section.

6.6 Election Algorithms

These kind of algorithms are used for choosing a coordinator, which is represented by a single process. The coordinator is like the sever in the server-based mutual exclusion algorithm.

Each process P_i has a variable $elected_i$, which is initially set to $\perp$. Moreover, two requirements must always be met. These requirements are:

1. **Safety**
   At any point in time

   $$\forall P_i : elected_i \in \{\perp, P\}$$

   Where P is a non-crashed process with the largest id;

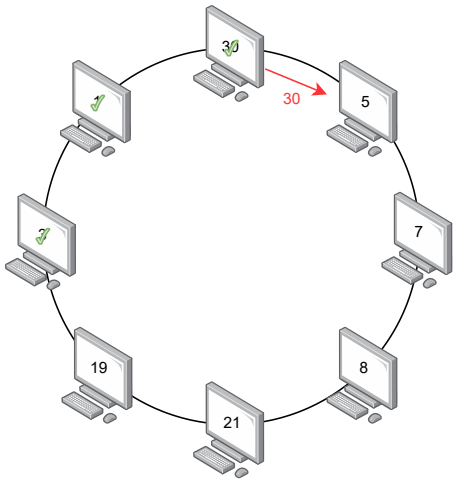
2. **Liveness**
   All non-crashed processes P_i will eventually set

$$elected_i = P$$

   Where P is a non-crashed process with the largest id.

6.6.1 Ring-Based Election Algorithm

Let's suppose that we have an asynchronous system, no failures, processes arranged in a logical ring, and that messages can only circulate in one direction.



The process of election is the following:

1. Every process is initially marked as **non-participant**;
2. To start an election, process P_i makes itself a participant, and places its id onto the message to send to its neighbour;
3. When P_j receives the message:
   - If the id is greater than P_j's id, then the message is forwarded directly to the neighbouring process;
   - If P_j is not marked as a participant, it is marked as a participant and its id is sent to the neighbouring process.
4. If P_i receives back its own id after one complete cycle of election, then it is elected as **leader** and marked as non-participant. Then, it sends the **elected** message with its own it to its neighbouring process;
5. When P_j receives the **elected** message, it saves the id of the leader and passes the message onwards.

Let's consider the worst-case scenario. Let's say that process P_i starts the election and that the process before it (process P_j) is the one with the highest id. In this case we will have the following:

- $N - 1$ messages needed to reach process P_j;
- N messages for P_j to announce to the other processes it is now the leader;
- N messages for all of the other processes to acknowledge the id of the new leader.

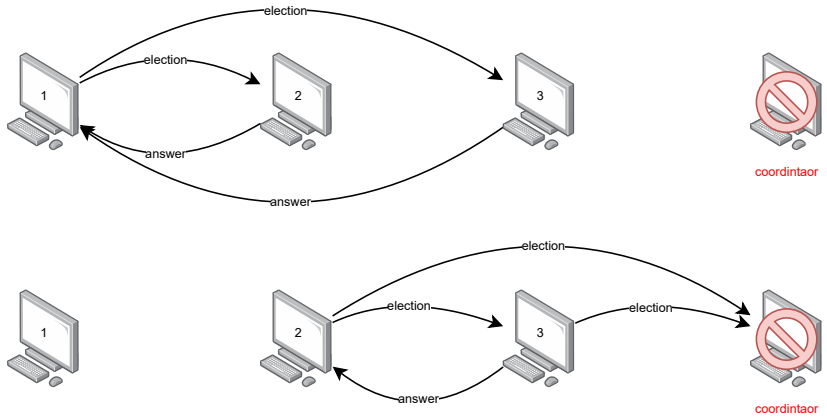
A total of $3N - 1$ messages are sent for the election process to be complete.

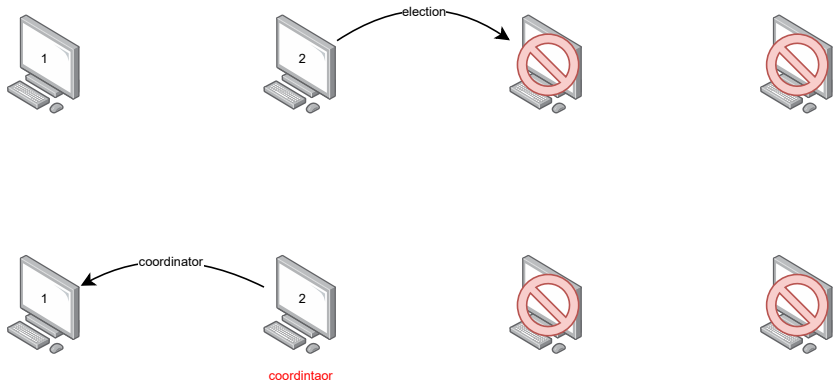
6.6.2 Bully Algorithm

In this algorithm, we take into account the fact that a process may crash. However, messages are sent reliably and all processes know every other process' id. Moreover, this is a synchronous system, where:

- T_{trans} is the maximum message transmission delay;
- $T_{process}$ is the maximum process delay;
- $T = 2_{trans} + T_{process}$ is the total elapsed time upper bound.

Graphically the process can be represented as follows:





The steps performed by the algorithm are the following:

1. Let P_1 suspect the crash of the current leader: process P_4;
2. If P_1 has the highest id, then
   - Send a **coordinator** message to all other processes with lower id to elect itself as the new leader;
   - Otherwise, P_1 sends an **election** message to all the processes with higher id, and awaits for an **answer** message.
3. If no **answer** message arrives within the predefined time T, then
   - P_2 considers itself the leader and sends a **coordinator** message to all processes with lower id;
   - Otherwise, P_2 awaits another period T for a **coordinator** message. If none arrives, then P_2 will start a new election.
4. If P_1 receives a **coordinator** message, it sets
$$selected_1 = P_2$$
5. If P_1 receives an **election** message, it will answer with an **answer** message and starts a new election – if one hasn't already been started.

In the best-case scenario, $N - 1$ messages will be exchanged. In the worst-case scenario, N^2 messages will be exchanged.

7 Replication and Consistency

Replication guarantees several different things:

- **Reliability**
  It protects against both failure and corrupted data;

- **Performance**
  It helps in scaling up numbers of clients and geographical area. By increasing the number of clients, it follows an increase in the number of processes needing access to data.

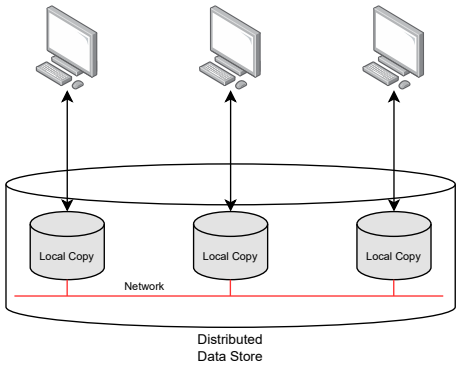
The main problem with replication is that multiple copies may lead to consistency issues. For this reason, we should modify the copies or the original with caution.

The solution to consistency problems could be solved by **loose consistency**. This means that global synchronisation is avoided – because too expensive, and that replica states may diverge.

7.1 Data-Centric Consistency Models

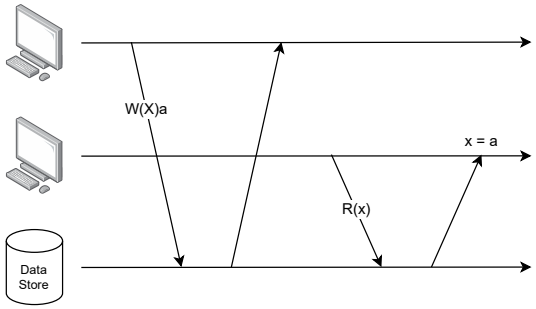
7.1.1 Data Store

In these kinds of models, we have a **distributed data store** on which read and write operations are performed.



7.1.2 Consistency Model

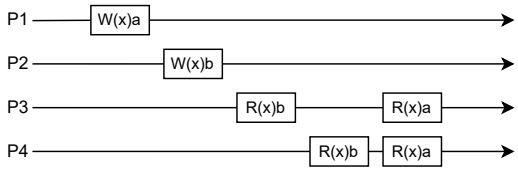
A contract exists between the processes and the data store. If this contract – i.e. set of rules – is respected by the processes, then the store will work correctly.



7.1.3 Sequential Consistency

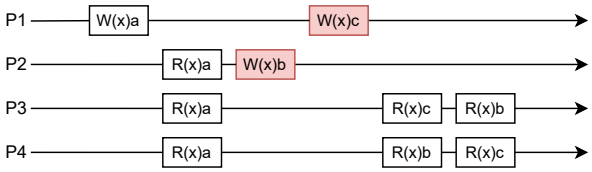
This concept was first introduced by Lamport. Read and write operations are allowed, and the variables are all initially set to NIL.

A data store is said to be **sequentially consistent** when the result of any execution is the same as if the read and write operations were executed in some sequential order. Moreover, the operations of each individual process appear in the program the same way as they appear in the sequence. In this case, time is not taken into consideration.



7.1.4 Causal Consistency

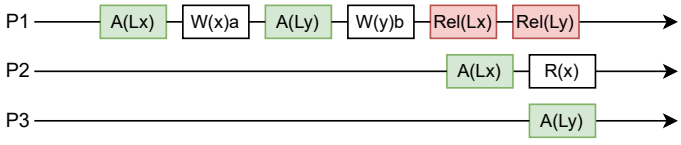
Writes that are potentially causally related, must be seen by all the processes in the same order. Concurrent writes may be seen in a different order on different by different machines.



In the case above, write operations $W_{P_2}(x)b$ and $W_{P_1}(x)c$ are concurrent. Although being concurrent, this execution is causally consistent.

7.1.5 Entry Consistency

Sequential and causal consistency were initially developed for shared-memory multiprocessor systems. These paradigms do not work well in the case of distributed systems.



Processes, before being able to modify shared data, must acquire a lock. This lock will then be released when the data has been successfully modified. A lock is associated with every data item.

7.2 Client-Consistency Models

7.2.1 Eventual Consistency

In the case of eventual consistency, we have a special class of distributed data stores. Here, the workload is mostly composed of reads and few – if any – writes. Data stores use a **weak consistency** policy.

These kind of distributed and replicated databases tolerate a high degree of inconsistency. In the absence of updates, all replicas converge to the same state. This is known as **eventual consistency**.

Some issues with this approach are:

- Unreliable network connectivity and poor performance
- Processes may connect to different copies and submit read and write commands.

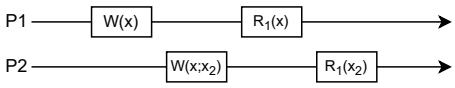
In the following sections, four different types of consistencies will be explained in detail.

7.2.2 Monotonic Reads

A data store is said to provide monotonic-read consistency is the following holds:

> If a process reads the value of data item x, any successive read operation on x by that process will always return that same value or a more recent value.

An example of monotonic-read consistent data store is the following:

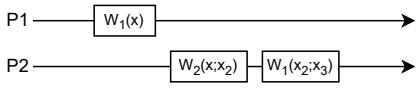


7.2.3 Monotonic Writes

A data store is said to be monotonic-write consistent if the following condition holds:

> A write operation by a process on a data item x is completed before any successive write operation on x by the same process – similarly to FIFO ordering.

An example of monotonic-write consistent data store is the following:

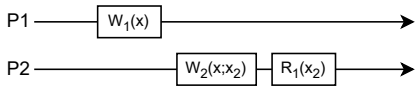


7.2.4 Read your Writes

A data store is said to provide read-your-writes consistency if the following condition holds:

> The effect of a write operation by a process on data item x will always be seen by successive read operations on x mad by the same process.

An example of a data store with read-your-writes consistency is the following:

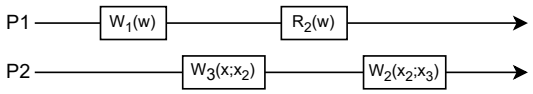


7.2.5 Writes follow Reads

A data store is said to provide writes-follow-reads consistency if the following condition holds:

> A write operation on data item x following a previous read operation on x – by the same process – is guaranteed to take place. Moreover, it will take place on the same or more recent value of x that was read.

An example of a data store that provides writes-follow-reads consistency is the following:



7.3 Replica Management

The objectives of replica management are mainly two:

- Find the best location for a server that can host part of the data;
- Find the best servers for content placement.

7.4 Replica Server Placement

The placement of the replica represents an optimisation problem. As such, we need heuristics to solve it. We need to find the best k locations out of the N possible locations – where $k < N$.

7.5 Content Distribution

First we need to decide what is there to propagate. Several options are possible:

- Propagate only the notification of an update;
- Transfer data from one copy to the other;
- Propagate the update operation to the other copies.

There are **push** and **pull** protocols to deal with update propagation.

- **Push-Based Model**
  Updates the propagated data without replicas asking for updates. This type of protocol is usually used to obtain a high degree of consistency.
- **Pull-Based Model**
  Here the client is the one who asks for updates. This type of protocol is usually used to update caches.

There are also two modes of updates communication. These are:

- **Unicasting**
  To update N servers we need to send N messages. This communication mode is more suitable for pull-based propagation.
- **Multicasting**
  The underlying network will take care of propagating the updates to the clients. This method is very effective when the replicas are in the same LAN (Local Area Network). This mode of communication is more suitable for push-based propagation.

7.6 Consistency Protocols

7.6.1 Primary-Based Protocols

Each data item x in the data store has a **primary** – i.e. an original copy. This primary is responsible for coordinating write operations on x. There are two main classes of primary-based protocols:

- **Remote-Write Protocols**
  This is a primary backup. The write operations are forwarded to a fixed single server.
- **Local-Write Protocols**
  The primary migrates between servers. This is a primary-backup protocol, in which the primary migrates to the process that wants to perform the update.

7.7 Replicated-Write Protocols

Write operations are executed at multiple replicas, not only at the primary. To perform these multiple write operations, we can use one of the following protocols:

- **Active Replication**
  In this case, each replica receives and executes all commands. These commands need to be executed in the same order for all replicas. Lamport's total order protocol- or sequencer-based mechanism is used to totally order the commands.

- **Quorum-Based Protocol**
  The clients request and acquire permission of multiple servers before being able to read and write to a replicated data item. Read quorum N_R and write quorum N_W are:

$$N_R + N_W > N$$

$$N_W > \frac{N}{2}$$

8 Fault Tolerance

A system is said to be **dependable** if it can tolerate faults. This means that the system can provide a service despite faults. A dependable system has the following characteristics:

- **Availability**
  A highly available system is most likely working at any given time;

- **Reliability**
  A highly reliable system will most likely continue to work without failures;

- **Safety**
  A safe system does not do anything wrong (but it may stop);

- **Maintainability**
  Describes how easily a failed system can be fixed.

Faults that affect systems can be of several different types:

- **Transient**
  Faults that occur once and then disappear;

- **Intermittent**
  Faults that occur and vanish several times;

- **Permanent**
  Faults that continue to exist until the faulty component is replaced.

8.1 Failure Models

There exist several different failure models, for example:

- **Crash Failure**
  A server halts but was working properly before it stopped;
- **Omission Failure**
  It can either be a receive or send omission. A **receive omission** is when the server does not receive a request. On the other hand, a **send omission** is when the server has done some work but cannot send the message;
- **Timing Failure**
  The response lies outside of a real-life time interval;
- **Response Failure**
  The server's response is incorrect;
- **State Transition Failure**
  A request made by a server reacts unexpectedly;
- **Arbitrary/Byzantine Failure**
  This is the most difficult failure to handle. It can happen when a piece of software has bugs or when a server has been hacked.

8.2 Failure Masking

The system failures described in the section above can be masked thanks to the use of redundancy. Redundancy can be of different types:

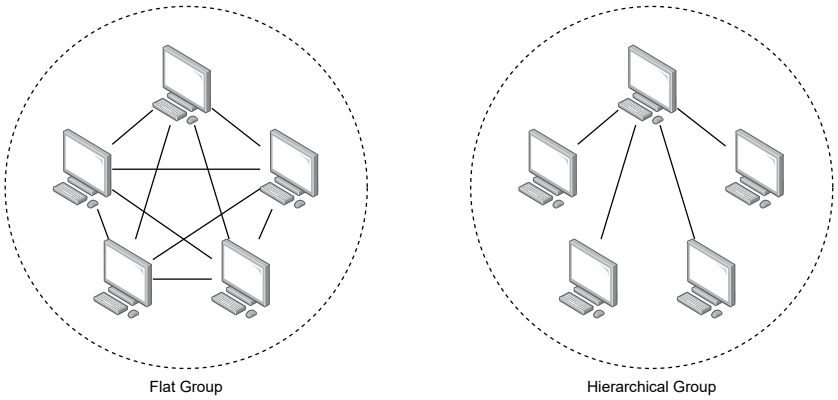
- **Information Redundancy**
  Extra bits are added to a message in order to allow for recovery (e.g. Hamming Code);
- **Time Redundancy**
  An action is performed multiple times;
- **Physical Redundancy**
  Extra equipment or processes are added to better tolerate failures.

8.3 Process Resilience

The idea of process resilience is to group identical processes together. When a message is sent to the group, all processes that are part of that group will receive it. These groups are also dynamic, this means that processes can join and leave groups, as well as being part of multiple groups at the same time.

8.3.1 Flat vs Hierarchical Groups

The terms **flat** and **hierarchical** describe the internal structure of a group, as well as how the work is carried out in those groups.



To remember group membership, we have two approaches:

- **Centralised**
  One node has the list of all group members;
- **Distributed**
  The group state is saved at each group member.

In these cases failure masking is done through **replication** and **agreement**. Agreement is used to elect a coordinator and to decide whether to commit a transaction or not. However, **failure detection** is vital for a distributed system to be truly fault-tolerant.

8.4 Consensus in Faulty Systems

In a fault-tolerant process group, each non-faulty process executes the same commands – and in the same order – as every other non-faulty process.

Non-faulty processes need to reach consensus on which command to execute next.

8.4.1 Flooding-Based Consensus

Let's consider a process group $P = \{P_1, \ldots, P_N\}$ with fail-stop feature semantics. A client now contacts P_i requesting it to execute a command. Every P_i maintains a list of proposed commands. The flooding-based consensus algorithm works as follows:

1. In round r, P_i multicasts its known set of commands $C_{r,i}$ to all the other processes;

2. At the end of round r, each process P_i merges all received commands into a new $C_{r+1,i}$;

3. The next command cmd_i is selected through a globally shared deterministic function.

$$cmd_i \leftarrow select(C_{r+1})$$

8.5 Reliable Client-Server Communication

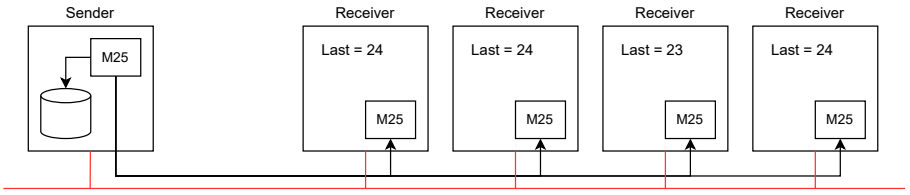
TCP masks omission failures by handling lost messages with acknowledgments. In case of other types of failures, we can employ RPC semantics to deal with such failures. RPC deals with:

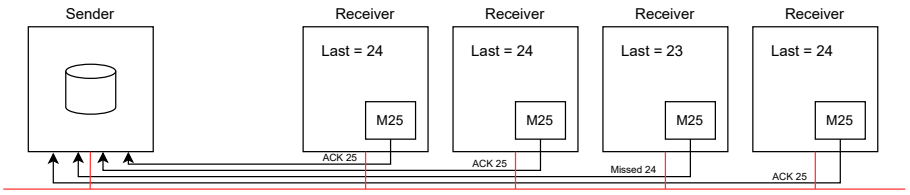
- **Client unable to locate server**
  This failure can be handled by throwing an exception at the client.;
- **Request from client is lost**
  The client can start a timer and resubmit the request if there is no response;
- **Server crashes after receiving request**
  We could either give up immediately and report the error – **at-most-one** semantics, or we could keep trying until a response is received – **at-least-one** semantics;
- **Reply from server is lost**
  The client could try to resend the request if no response is received after some time.

8.6 Reliable Group Communication

8.6.1 Basic Reliable-Multicasting Schemes

These schemes are important mechanisms used to implement replication in groups. A message sent to a group should be received by all members of that group. A basic implementation of such an algorithm is the following:





There are some issues with this solution, namely:

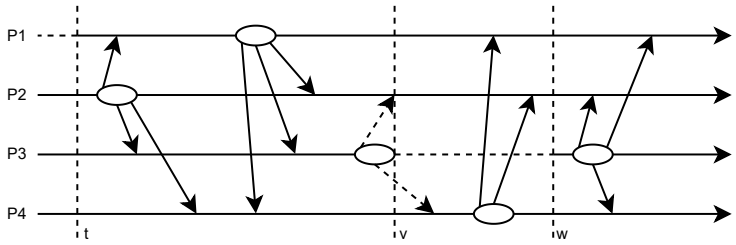
- If there are N receivers, the sender must send N messages;
- The sender might be swamped with ACKs in the feedback step of the protocol – second image above;
- Negative ACKs are only sent when the receivers miss a message;
- There is a risk of feedback implosion.

8.6.2 Atomic Multicast

This is the basis for state machine replication. Here, messages are sent to all non-faulty processes and delivered in the same order. This multicast system ensures consistency and forces reconciliation.

8.6.3 Virtual Synchrony

All processes in the same view must agree on which messages are to be delivered in that view. Virtual synchrony ensures that the message and the view-change message are delivered in the same order by all group members.



The steps performed in the above visualisation are the following:

1. $P1$ joins the group at time t;
2. $P2$ performs a reliable multicast by sending multiple point-to-point messages;
3. $P1$ performs a reliable multicast by sending multiple point-to-point messages;

4. $P3$ crashes at time v while performing the reliable multicast. This partial multicast is discarded entirely;
5. $P4$ performs a reliable multicast by sending multiple point-to-point messages;
6. $P3$ rejoins the group at time w;
7. $P3$ performs a reliable multicast by sending multiple point-to-point messages.

8.7 Message Ordering

This approach defines how different multicasts are ordered.

8.7.1 Reliable Unordered Multicast

No guarantees are given concerning the message ordering.

P1	P2	P3
send M1	receive M1	receive M2
send M2	receive M2	receive M1

8.7.2 Reliable FIFO-Ordered Multicast

The incoming messages from the same process are delivered in the order in which they are sent.

P1	P2	P3	P4
send M1	receive M1	receive M3	send M3
send M2	receive M3	receive M1	send M4
	receive M2	receive M2	
	receive M4	receive M4	

8.7.3 Reliable Causally-Ordered Multicast

Messages are delivered such that potential causality between different messages is preserved. This multicast protocol has two properties:

- **FIFO Order**
- **Local Order**
  For example, if a process delivers $M1$ before multicasting $M2$, then no process will deliver $M2$ before $M1$.

8.7.4 Reliable Totally-Ordered Multicast

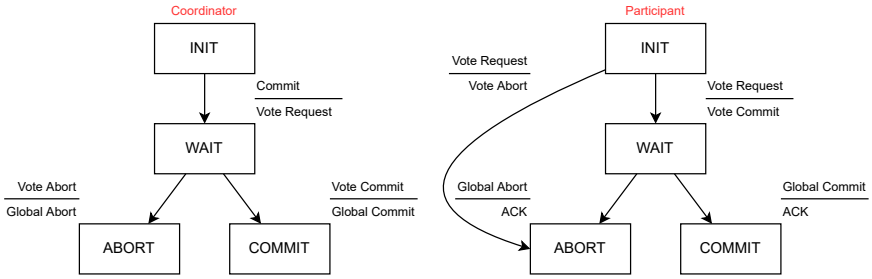
All messages are delivered in the same exact order.

8.8 Distributed Commits

One-phase commits protocol do not work in distributed systems with reliable multicasting. For this reason, more sophisticated protocols are used.

8.8.1 Two-Phase Commit

The protocol is structured as follows:



Some characteristics of this protocol are the following:

- The protocol may block if the coordinator crashes – **single point of failure**;
- To avoid blocking on the participants, we can use timeouts;
- In some of the states, a process is blocked waiting;
- Until the coordinator recovers, all of the processes are blocked.