Distributed Systems

github.com/martinopiaggi/polimi-notes

2023-2024

Contents

1 Modeling distributed systems

"A collection of independent computers that appears to its users as a single coherent system"

Key Characteristics of DS are:

- **Concurrency**: Multiple processes run simultaneously.
- **Absence of a Global Clock**: There's no single clock that synchronizes all operations.
- **Independent Failures**: Failures can occur independently and can be partial.
- **Heterogeneity**: There's a diversity in hosts, platforms, networks, protocols, languages, etc.
- **Openness**: interoperability through standard access rules, with protocols and interfaces being crucial.
- **Security**: The ease of attaching a node to the system can be exploited maliciously
- **Scalability**: The system should be designed to grow without significantly affecting performance.
- **Failure Handling**: Nothing is entirely reliable. Hosts can fail, links can be unreliable, and distinguishing between the two can be challenging. The system should be capable of detecting, masking, tolerating, and recovering from failures.
- **Transparency**: The system should hide complexities to simplify the tasks of programmers and users.

1.1 Architectural Styles

- **Client-Server**: Layers (tiers)
- **Service Oriented**: built around the concepts of:
  - services: units of functionality
  - service providers
  - service consumers
  - service brokers: list available services reducing system dependency Web Services are a particular implementation of SOA, designed to support machine-to-machine interaction over a network, often using protocols like SOAP and implemented with technologies like HTML. Web service operations are invoked through SOAP, a protocol, based on XML, which defines the way messages (operation calls) are actually exchanged.
- **REST**: Set of principles that define how Web standards are supposed to be used Interactions are client-server and stateless. The REST most import concept is the "stateless" feature. Components expose a uniform interface: `GET`, `POST`, `PUT`, `DELETE`.
- **Peer to Peer**:
  - No distinction between clients and servers.
  - Scalable and decentralized
- **Object-Oriented**:
  - Components are objects
  - RPC (Remote Procedure Code) is used.
  - Enterprise Java Beans is an example of a client-server Object-Oriented interaction.
- **Data-centered**: Components communicate through a common repository (usually passive). First example of this is Linda where data in the shared space is organized in tuples and clients work on it using primitives.

- **Event-Based**: Message based architecture where components are divided into:
  - publishers
  - subscribers
- **Mobile code**: based on the concept to relocating components
  - **Client-Server**: both the code and its execution belong to the server side. The client-side process requests a service from a server-side process. The server then executes the task and sends back the result to the requesting client.
  - **Remote Evaluation**: The service requester possesses the code, while the other side executes it and sends back the required value. This can pose security concerns as the server allows the execution of

external code.

- **Code on Demand**: The service requester hasn't the code. The code is then requested and subsequently executed locally. Clients should be capable of executing the code locally, an example being JavaScript modules.
- **Mobile Agent**: An entity has both the code and data but lacks the capability to process them. These are partially evaluated and sent to the other side for completion.

Mobile Code Paradigm	Execution Location	Code Origin
Client-Server Model	Server	Server
Code on Demand	Client	Server
Remote Evaluation	Server	Client
Mobile Agents	Varies (Client/Server/Other Nodes)	Mobile Agent

1.2 The consensus problem

Consensus in DS is a broad problem which is common in distributed applications. It refers to the challenge of ensuring that multiple entities (or nodes) agree on a single data value, such as a decision or a command. This problem becomes particularly complex when considering the potential for communication failures, unreliable nodes, or malicious actors.

1.2.1 Pepperland Case

The "Pepperland" example is a metaphorical representation of the consensus problem: two (or more) generals with **separated** battalions, which can communicate through messages, needed to reach an agreement to/not to attack at dawn. The coordinated attack problem shows that consensus problem is solvable with the assumption that messengers cannot be captured. Asynchronous and synchronous communication is a decisive variabile in this context: in asynchronous Pepperland there are no bounds on the time required by the messenger. The problem on who will lead the charge is possible in both synchronous and asynchronous way. Regarding an agreement to the simultaneous charge is possible only in synchronous Pepperland. In Synchronous Pepperland, it's possible to determine the maximum difference between charge times:

1. Define the range of message transmission times as min and max.
2. The leader sends a "charge" message, waits for min minutes, and then initiates the charge.
3. Upon receiving the "charge" message, the other general charges immediately.
4. The second division might charge later than the first one, but the delay will not exceed $(max - min)$ minutes.
5. If it's known that the charge duration will exceed this time difference, victory is assured.

If we are in presence of **arbitrary communication failures** the problem is **unsolvable** in both Pepperland: "we cannot know if the message has been lost or it is still arriving".

Aspect/Scenario	Asynchronous Pepperland	Synchronous Pepperland
Bounds on messenger time	No bounds	Bounded
Problem on who will lead the charge	Possible	Possible
Agreement to simultaneous charge	Not Possible	Possible
Maximum difference between charge times	Not possible	$(max - min)$ minutes
In case of arbitrary communication failures	Unsolvable, no guarantees	Unsolvable, no guarantees

2 Communication models

2.1 Middleware

Middleware as a protocol layer is adopted to raise the level of abstraction of communication. Middleware main function is to provide the (un)marshaling of data:

- naming protocols
- security protocols
- scaling mechanisms (such as replication and caching)

Actually the middleware can assume different roles depending on the architecture paradigm: for example in a event based architecture the middleware acts as the event dispatcher, collecting subscriptions and routing event notifications based on these subscriptions. Other possibilieties are:

- Middleware as service broker in a SOA
- Middleware as registry (for naming) in RMI

Middleware can offer different forms of communication:

- Transient/persistent communication
- synchronous/asynchronous communication

Popular combination are:

- transient communication with synchronization after processing
- Persistent communication with synchronization at the request submission.

2.2 Remote procedure call

The ***Interface Definition Language*** (IDL) abstractly describes a function's interface, separating it from its implementation. IDL is important when we talk about **Remote Procedure Call (RPC)**, which is a protocol that one program can use to request a service (function) from a program located in another computer on a network without having to understand the network's details. RPC it's based on a sort of middleware which from signature creates the code to manage all the transport and remote decoding stuff:

- serialization
- marshalling: hosts may use different data representations (e.g., little endians vs. big endians) and proper conversions are needed. For example a data structure representation or even a integer can be completely different from source/destination machine.

Variants of RPC:

- **Lightweight RPC** is a variant of RPC designed for local environments with fewer threads and processes. It eliminates the need to listen on a channel. Instead of actual packets, it utilizes shared memory to serialize and share data between processes. The receiving process waits for the data to become available in the shared memory, processes it, and returns the result. This approach is more efficient and eliminates resource waste compared to traditional RPC.
- **Asynchronous RPC** has several variations with different semantics.
- Some frameworks also support **batched** and **queued RPC**.

The main challenge of RPC is binding the client to the server, which involves determining the location of the server process and establishing communication with it.

2.2.1 Examples RPC

- **Sun's solution**: daemon process that bind calls and server ports. It solves only the second problem, since the clients must know in advance where the service resides. Sun RPC includes the ability to perform

batched RPC.
- **DCE's solution**: Sun's solution but with a directory server to allow us to fully decoupled client-server such that the client has to known only the address of the directory service, goes to the directory service, list the set of functions that are available, search for the function that it must bind this happens at the binding time.

2.2.2 RMI: Remote method invocation

RMI (Remote Method Invocation) is the natural extension to RPC: same thing but OOP.

> "Actually, under the hood, there is a big difference to copy just data structures and objects which have methods!"

In RMI, there is a natural separation between the interface and implementation of an object. This distinction not only exists logically but also physically, as the object's implementation resides on a remote machine. This setup provides the illusion of referencing a remote object as if it were local, simplifying the interaction between objects located in different locations.

The use of a proxy **enables passing objects by reference** in RMI. This proxy serves as a representation of the remote object and allows the client to invoke methods on the remote object as if it were a local object. In RMI it is not feasible to pass objects by copying due to the presence of **methods** in the objects. For example, in a multilingual environment would be necessary to translate the methods code while copying the object (impractical task). Therefore, RMI's approach of passing objects **by reference** using a proxy is a more effective solution.

2.2.3 Passing parameters approaches

Parameters are embedded in the request and can be passed in three ways:

- **By value** (or **by copy**): A copy of the parameter's value is created and passed to the function. Changes made to the parameter within the function don't affect the original value (like in C when passing data types).
- **By reference**: The memory address of the parameter is passed to the function, allowing direct access and modification of the original value. This is impossible in DS since they don't share a common memory.
- **By copy/restore**: a copy of the original parameter is created and passed (like in *by value* approach) but the changes made to the parameter after the call are saved in the "original variable": this produce apparently the same effect as "by reference".

	By Copy	By Copy/Restore	By Reference
RPC	Easy (data structures accessible)	Easy (data structures accessible)	Impossible
RMI	Exception: Java RMI (due to JVM for Java <-> Java)	Impossible	Easy (proxy of the object)

2.3 Message oriented communication

RPC/ RMI you use invocations of methods and there is full synchronization whereas MOC is based on message and it is typically **asynchronous**. Possible models of message-oriented communication are message passing, message queuing and publish/subscribe. Each of these models offers different ways of managing and sending **messages** within a system.

2.3.1 Message passing

Message passing is a common form of message-oriented communication. We can distinguish these 4 main approaches to message passing:

- **Stream sockets**: the server accepts connection on a port, the clients connect to the server (TCP: connection oriented)
- **Datagram sockets**: both client and server create a socket bound to a port and use it to send and receive datagrams (built on UDP)
- **Multicast sockets**: IP multicast is a protocol used to transmit UDP datagrams to multiple recipients simultaneously.
- **MPI**: an higher level version of sockets designed explicitly for clusters of machines. Still low-level and very lightweight.

2.3.2 Message queuing

A much higher level than sockets, point-to-point, persistent (first form of **persistent** communication that we encounter in this course) message oriented communication model. Main features:

- Messages are sent to queues to a **server's queue**.
- The server **asynchronously** retrieves the requests from the queue, processes them, and returns the results to the client's queue.
- Queues are identified by symbolic names, eliminating the need for a lookup service.
- Queue managers act as relays to manipulate the queues.
- Main idea is to promote **decoupling**: thanks to persistency and asynchronicity, clients need not remain connected
- Queue sharing simplifies **load balancing**

2.3.3 Publish-Subscribe

Event-based architectural style where there is a **event dispatcher component** (a middleware) which collects subscriptions and is responsible for routing events to matching subscribers. Subscriptions can be either **subject-based** or **content-based**.

Communication in this style is:

- **Transient**: It is not stored for future reference.
- **Asynchronous**: Events can be sent and received without waiting for a response.
- **Implicit**: It does not require direct interaction between sender and recipient.
- **Multipoint**: Multiple subscribers can receive the same event.

The **dispatcher** in a system can either be **centralized** or **distributed**. It is often the **bottleneck of the system**, especially when dealing with content-based matching, as it is more difficult to match based on content rather than simply matching the topic name.

2.3.3.1 Acyclic graph topology When the topology of the distributed dispatcher is an **acyclic graph** (graph of brokers) three approaches exists:

- **Message forwarding**: each broker stores only subscriptions coming from directly connected clients (so subs are cheap since only interests the closer node). Messages are forwarded from broker to broker and eventually delivered to clients by the closer node only if they are subscribed. a lot of subscriptions few publications better using message forwarding.
- **Subscription forwarding**: complementary approach of message forwarding where the subscription propagates through the network. And now if you send a message, the message only gets delivered along the minimal route that goes from the sender to the destination. Subscription is not propagated along route where it's already propagated before. This approach is ok if you have subject based subscriptions,

while it's not ok if you have content based subscription because in principle you could have a subscriptions that covers another subscription. This approach is better if you don't subscribe too much and you publish a lot so few subscriptions a lot of publications better using sub subscription forwarding .

- **Hierarchical forwarding**: the graph is organized as a spanning tree which routes the messages only in the correct branches.

2.3.3.2 Cyclic graph topology When the topology of the distributed dispatcher is an cyclic graph (graph of brokers) **DHT** approach exists. **DHT approach**: nodes are organized in a **hash table**. To subscribe for messages having a given subject S:

- Calculate a hash of the subject H_s
- Use the DHT to route toward the node succ(Hs)
- While flowing toward H_s leave routing information to return messages back

To publish messages having a given subject S

- Calculate a hash of the subject Hs
- Use the DHT to route toward the node succ(Hs)
- While flowing toward succ(Hs) follow back routes toward subscribers

And since the DHT do routing even in cyclic graphs, so even using and leveraging, taking advantage of the fact that the network is cyclic, you may implement, efficiently implement, publish, subscribe, routing on top of the DHT by distributing the routing subject-based system DHT can be used. Other approaches proposed in literature and basically never used:

- **Per Source Forwarding (PSF)**:
  - A spanning tree is built to keep a forwarding table organized per source.
  - Each publisher sends messages to all known brokers, which then forward to interested subscribers.
- **Improved Per Source Forwarding (iPSF)**:
  - A spanning tree is built similar to PSF but leverages the concept of indistinguishable sources.
  - Publishers send messages only to brokers with interested subscribers.
  - Advantages include smaller forwarding tables and easier construction.
- **Per Receiver Forwarding (PRF)**:
  - A spanning tree is built for each receiver (subscriber).
  - Subscribers inform brokers of their interests, and brokers request messages from publishers based on these interests, pulling messages based on demand.

2.4 Stream-oriented communication

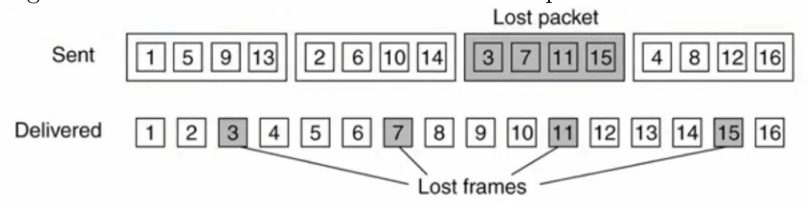
Stream-oriented communication is a different model of communication that focuses on streams. Streams are sequences of messages of the same type where their timeliness is important. An example of this is a video stream, where each frame is connected to the next frame. Transmission modes:

- **Asynchronous**: The data items in a stream are transmitted one after the other without any further timing constraints (apart ordering)
- **Synchronous**: There is a max end-to-end delay for each unit in the data stream
- **Isochronous**: There is max and a min end-to-end delay (bounded **jitter**)

Jitter refers to the variation in the time between packets arriving, caused by network congestion, timing drift, or other unpredictable factors. Which is our best protocol against jitter? There is no specific protocol, IP protocol is the best effort protocol and does not guarantee quality of service. However, it does have a **Differentiated Services** field in its header, which is used by nobody and even ignored by most of internet providers. So, since the network doesn't provide anything specific, everything must be done at the application layer:

- **Buffering**: packets are not instantly delivered to the receiver but first they are stored in a buffer.

- **Forward error correction**: is a mechanism used to detect and correct errors in data transmission. Unlike backward error correction, which involves going back to a previous state to correct errors, FEC is a technique where errors are detected and corrected in real-time without needing to resend the data.
- **Interleaving**: instead of sending packets that include consecutive frames, you send packets that mix together different frames. This means that if a packet is lost in transit, consecutive frames will not be lost.



3 Naming

In a DS, entities such as hosts, processes, threads, disks, and files need a mechanism is needed to be correctly mapped to their addresses. Names are used as references for these entities, allowing them to be accessed by other entities. These names can be classified as global or local, as well as human-friendly or machine-friendly.

3.1 Flat naming

In a **flat naming** schema, names are "flat", meaning they are simple strings without structure or content. It can be implemented with:

- **Simple solutions**: designed for small scale envs - **broadcast - multicast - forwarding pointers**: a particular technique to solve mobile nodes which consists in leaving reference to the next location at the previous location
- **Home-based approaches**: A single home node is used to track the location of the mobile unit. The assumption is that the home node is stable and can be replicated for reliability. However, this approach adds an extra step and increases latency. This system lacks geographical scalability, as the entity may be physically close to the client but still relies on the home node.
- **DHT**: DHT is a good solution for flat naming because it's strongly distributed. An example is **Chord** which is a protocol and algorithm for a peer-to-peer distributed hash table.
- **Hierarchical approaches**: based on organize the the instances into a tree in which nodes can represent subdomains (sub-trees). Leaves contains entities. This approach doesn't scale well since higher nodes needs bigger database to store all "children".

3.1.1 Chord

Nodes and keys are organized in a logical ring and it's based on the idea that the routing table is used to "go" far as possible, as close as possible (in terms of nodes visited) to the node you are searching without overshooting. The highlights of Chord are:

- The "finger table" permits a **logarithmic search**: the item with key k is stored by the largest node in the successor column where there are all nodes $(id + 2^i)$-th node for incrementing i.
- The number of bits used by the DHT corresponds to the number of rows in the routing tables.
- The search space is typically much larger than the current number of computers connected, allowing for the addition of new nodes. This means that with a given ID there is no computer connected in most cases

3.1.1.1 Chord considerations The use of Chord in a network provides two key benefits:

- Chord finger tables increase efficiency by allowing nodes to bypass large sections of the network during lookups.

- Finger tables enable scalability as the network grows and more nodes join. These tables dynamically adjust to ensure that search time remains logarithmic, even with a larger number of nodes.

Drawbacks:

- A major drawback is the maintenance overhead. Nodes need to regularly update their finger tables to accommodate new nodes joining or existing nodes leaving the network.
- Implementing and managing finger tables can be more **complex** when compared to simpler Peer-to-Peer (P2P) architectures.

3.2 Structured naming

Structured naming systems organize names in a **namespace**, which is a labeled **graph** composed of leaf nodes (representing named entities) and directory nodes (with labeled outgoing edges pointing to different nodes). Names are referred to through path names, which can be absolute or relative. The namespace for large-scale systems is often distributed among different name servers, usually organized hierarchically. The namespace can be partitioned into logical layers:

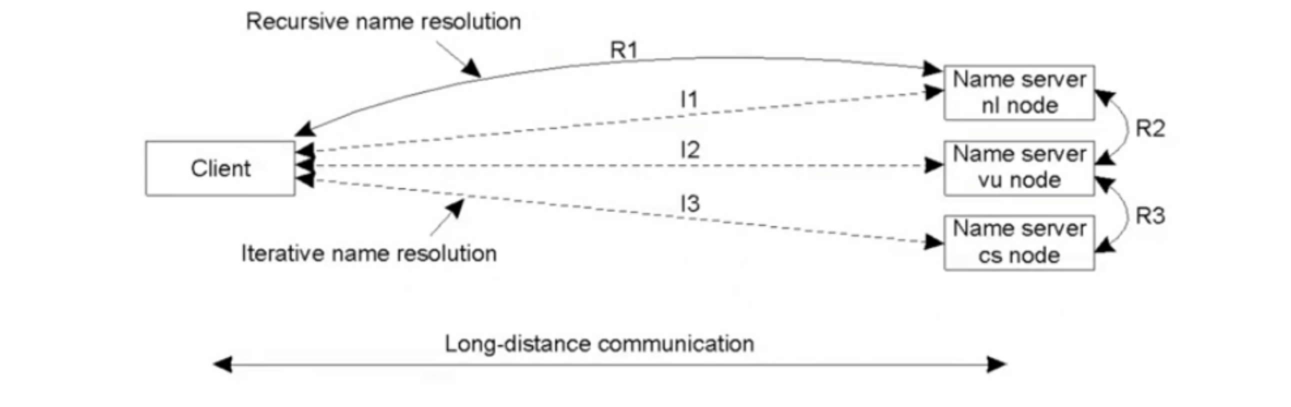
- global level
- administrational level
- managerial level

Item	Global	Administrational	Managerial
Geographical scale of network	Worldwide	Organization	Department
Total number of nodes	Few	Many	Vast numbers
Responsiveness to lookups	Seconds	Milliseconds	Immediate
Update propagation	Lazy	Immediate	Immediate
Number of replicas	Many	None or few	None
Is client-side caching applied?	Yes	Yes	Sometimes

3.2.0.1 Name resolution techniques

Two main techniques of name resolution are **iterative** and **recursive**:

- Iterative ns involves resolving the name by first asking the root and then its sub-nodes to find the responsible resource. Pros of this technique include:
  - reduced communication costs
  - more effective caching along the resolution chain. However, it may lead to higher demand on each name server.
- Recursive ns starts from the root and then recursively forwards the request to its sub-nodes. This technique does not involve caching, which can result in slower lookups. Additionally, each name server in this resolution method is responsible for a single request, increasing demand on each server.


Recursive name resolution
R1
I1
I2
I3
R2
R3
Client
Name server nl node
Name server vu node
Name server cs node
Iterative name resolution
Long-distance communication

The DNS structure is a practical example of a structured naming system, where the namespace is hierarchically organized, and caching and replication are used to enhance efficiency. DNS namespace is organized as a hierarchically rooted tree with separate authorities for each domain. Each name server is responsible for a specific zone. Global servers are mirrored and queries are routed by IP anycast (not broadcast, it's a variant of unicast where is used only a node, chosen by the network among the nodes). Caching and replication are widely used, with stability in the global and administrative layers while more frequent changes in the managerial layer.

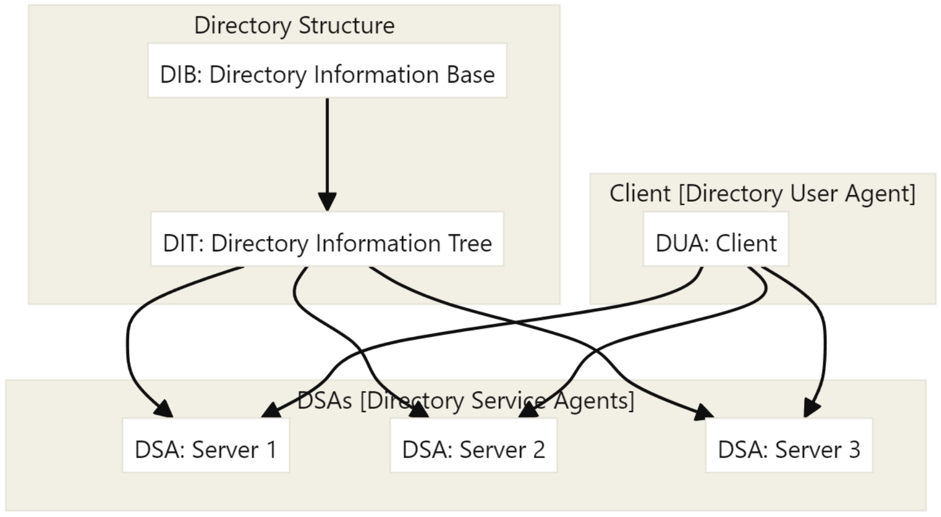
While moving a host within the same domain only requires updating the domain name servers, if i's moved to a different domain, DNS servers can provide the new IP address or create a symbolic link to the new location but may affect lookup efficiency.

The structured naming system, implemented using a hierarchy of servers, is more efficient than a flat naming system because it allows for more effective caching and reduced communication costs due to its organized structure. The hierarchical organization of the namespace in structured naming systems, as exemplified by the DNS, enables faster lookups and better scalability. In contrast, flat naming systems, while simpler, can suffer from inefficiencies due to their lack of structure, especially as the scale of the system increases.

3.3 Attribute-based naming

The idea of attribute-based naming is to add a set of attributes (which code properties) to the name of entities. Attribute based naming systems are usually called **directory services** and they are usually implemented by using **DBMS**. **LDAP** (Lightweight Directory Access Protocol) is the dominant protocol used for directory services. LDAP directories consist in a collection of directory entries which is called Directory Information Base (**DIB**). Each record in the DIB has a unique name composed of naming attributes. In large-scale directories, the DIB is typically partitioned according to the Directory Information Tree (DIT) structure and:

- Servers are known as **Directory Service Agents** (DSA)
- Clients as **Directory User Agents** (DUA)


Directory Structure
DIB: Directory Information Base
DIT: Directory Information Tree
Client [Directory User Agent]
DUA: Client
DSAs [Directory Service Agents]
DSA: Server 1
DSA: Server 2
DSA: Server 3

3.4 Removing unreferenced entities

- In some cases, objects need to be removed from the system since unreachable from any other node.
- This is typically done by creating a graph
- Automatic garbage collection is common in conventional systems.
- Distribution of systems complicates the process due to lack of global knowledge and network failures.

Different approaches:

- **Reference counting**: Every object keeps track of the number of references it has. When an object is created, its reference counter is set to 1. When a reference is no longer needed, the counter decreases. The problem with this is **race conditions** when passing references between processes.
  - **Weighted reference counting**: Tries to circumvent the race condition by communicating only **counter decrements**:
    - It uses two counters, a partial and total one.
    - Removing a reference subtracts the proxy partial counter from the total counter of the skeleton.
    - The advantage over a "standard counting" is that creating a proxy doesn't require you to reform the skeleton. You only reform the skeleton when you destroy the proxy.
    - When the total and partial weights become equal garbage collection happens.
    - The only problem with this is that only a fixed number of references can be created. Still exists some techniques like "asking more weights form the proxy" or "create a recursive weighted reference counting" to solve this problem.
- **Reference listing**: it is used a list to track the identities of the proxies. Easier maintenance in the event of network failures since "pinging" clients to check if they are alive is possible. However, race conditions can still occur when copying references.
- **Mark-and-sweep**: Tries to identifies disconnected groups of objects from the root set using a graph and turning the problem into a "search/graph exploration". There are issues such as the requirement for a stable reachability graph and potential scalability problems.

4 Synchronization

In a DS there isn't a single clock

This chapter explores distributed algorithms for various synchronization tasks such as:

- synchronizing physical clocks
- simulating time using logical clocks
- preserving event ordering
- achieving mutual exclusion
- conducting leader election
- collecting global state and detecting termination
- managing distributed transactions
- detecting distributed deadlocks

In all of this **time** is a fundamental critical concept. In ds ensuring that all machines perceive the same global time is a critical challenge. Computer clocks are actually timers and to achieve synchronization, several factors need to be considered:

- **clock drift rate**, which is a constant value determined by the timer. For most quartz crystals, this drift rate is around 1 second per day, meaning that they can drift by approximately 11.6 seconds every 11.6 days.
- **Clock skew** refers to the difference in drift rates between two clocks. If two clocks are drifting in opposite directions, they will accumulate a skew equal to twice the product of the drift rate and the elapsed time.

To maintain synchronization, a resynchronization process is needed. There are two main approaches to achieving synchronization:

- The first approach is to synchronize all clocks against a single clock, typically one that has external and accurate time information. This ensures accuracy for all clocks, as they are aligned with the reference clock.
- The second approach is to synchronize all clocks among themselves, ensuring that they all agree on the same time. At the very least, time monotonicity needs to be preserved, meaning that time should always move forward and not jump backwards or stall.

Before dive in synchronization algorithms one important issue to note is that if a client recognizes that its own time is ahead of the correct time, it should **never switch its clock back in time**: it should be obvious .. switching the clock back can cause errors in running applications. Instead, the client should delay its clock until it reaches the synchronization point. For example, if the clock should be 11:59:59 but is at 12 o'clock, the client can delay its clock by going half the speed for two seconds until it reaches perfect synchronization.

4.1 Synchronization algorithms

4.1.1 GPS

The GPS algorithm is highly efficient in providing device positioning and clock synchronization. It operates through triangulation from a set of satellites whose positions are known. By measuring signal delay, the distance can be determined accurately. However, a challenge arises from the necessity of synchronizing the clocks between the satellites and the receiver, due to the inevitable clock skew. This limitation hinders the effectiveness of GPS in indoor environments, where GPS signals cannot be received reliably. Although GPS is a viable option, it may not perform optimally under normal circumstances.

In order for GPS to work, the clocks of the satellite and the station need to be perfectly synchronized. The satellites emit signals that are perfectly synchronized among themselves because they have atomic clocks on board. By measuring the flight time of the signal, the receiver can determine its position. To achieve this, at least four satellites are needed to provide four equations to solve for four variables (x, y, z, and time).

The precision of GPS is around 10 meters in distance, which requires highly precise clock synchronization. The main source of error comes from the time it takes for the signal to go from the GPS to the receiver's computer.

However, it is possible to synchronize the clocks of multiple computers with a precision in the order of nanoseconds using GPS, although this may not be feasible in certain cases where computers are located indoors.

4.1.2 Simple algorithms: Cristian's (1989)

How do you synchronize if you don't have a GPS on board of every station? One of the simplest algorithm is the Christian's algorithm.

In the time synchronization process, clients periodically send requests to the time server. However, there are certain problems:

- the time might run at a different speed on the client machine. To avoid this, a gradual change is introduced.
- the non-zero time it takes for the message to travel to the server and back. To account for this, the round-trip time is measured and adjusted. The adjusted time, denoted as , is calculated as the sum of the current clock time and half of the round-trip time.

Multiple measurements of the round-trip time are taken and then averaged to improve accuracy. However, even with these adjustments, there is still some error in the obtained time due to the delay between the request and response. Indeed, the assumption here is that the network is symmetric in terms of latency: meaning the time of the request is nearly the same as the response time. However, this assumption is an oversimplification that works if the flight time is very short compared to the desired precision.

4.1.3 Berkeley (1989)

The second approach in Berkeley Unix differs from Christian's algorithm in that the time server is active instead of passive: it collects the time from all clients, averages it, and then retransmits the required adjustment.

This approach synchronizes the machines with each other instead of against a single machine, which is reasonable when there is no assumption that one machine is more correct than the others.

4.1.4 Network Time Protocol (NTP)

This protocol was designed for UTC sync over large-scale networks and it's actually what is used today for large-scale networks like the internet and uses servers organized in a hierarchy. At the top of the hierarchy are machines with **atomic clocks** that synchronize other machines, down to the client machines. The synchronization method depends on the network:

- **Multicast** (over LAN): on LAN broadcast communication is typically used, where the NTP server periodically broadcasts the current time and the receiving machine synchronizes based on that time.
- **Procedural-call mode**: similar to Christian algorithm
- **Symmetric mode**: for higher levels that need the highest accuracies

The transmission times of messages m and m' gives this: $o = o_i + (m' - m)/2$ where o_i is an estimation of the time offset (between the two clocks), and d_i represents the accuracy of this estimation.

4.2 Logical time

In some applications, it is not necessary to have accurate absolute time. Instead, what is important is the ordering and causality relationships of events.

4.2.1 Scalar clocks

Lamport invented a simple mechanism by which the happened before ordering can be captured numerically using integers to represent the clock value. Each process p_i keeps a logical scalar clock L_i :

- L_i starts at zero
- L_i is incremented before p_i sends a message

- Each message sent by p_i is timestamped with L_i
- Upon receipt of a message, p_i sets L_i to: $MAX(\text{ timestamp }_{msg}, L_i) + 1$

The idea behind Lamport clocks is that they can serve as an approximation of the "happens before" relationship between events. In Lampard clocks, if one event a happens-before another event b ($a \rightarrow b$), then the scalar clock of a is less than the scalar clock of b . Note that the converse is not always true. Just because the scalar clock of "a" is less than that of " b " does not mean that $a \rightarrow b$. This because scalar clocks capture a partial ordering of events, not the full causal relationship.

4.2.1.1 Other versions of Lamport Clocks

1. **Lamport Clocks with Process IDs for Total Order Guarantee:**
   - **Objective:** Establish a total order of events across distributed systems.
   - **Mechanism:** Each process maintains the logical clock + each process IDs: IDs are used like tie-breakers for events with identical timestamps, in this way "nothing happens concurrently".
2. **Lamport Clocks for Total Ordering Multicast:**
   - **Objective:** Ensure total ordering of multicast property, which says "every process receive the message in the same order".
   - **Mechanism:** Ever
   1. A message is sent in multicast, with the logical timestamp of the sender
   2. When a process receives a message, it is put in a local **queue**, ordered by timestamp
   3. The receiver multicast an ACK to the other processes
   4. A message is delivered to the application only when it is at the highest in the queue and all its acks have been received.

4.2.2 Vector clocks

The problem of scalar clocks is that $e \rightarrow e' \Rightarrow L(e) < L(e')$ but the reverse does not necessarily hold, e.g., if $e \| e'$.The solution are Vector clocks. Basically is the same a scalar clocks but all process has/sends a vector, in which for each cell $V[j]$ there is a value associated with the process j (so N values for N processes).

Rules:

- $V_i[i]$ is the number of events that have occurred at P_i, initially $V_i[j] = 0$ for all i, j
- If $V_i[j] = k$ then P_i knows that k events have occurred at P_j
- P_i attaches a timestamp $t = V_i$ in all messages it sends (incrementing "its value" $V_i[i]$ just before sending the message
- When P_i receives a message containing t, it sets $V_i[j] = \max(V_i[j], t[j])$ for all $j \neq i$ and then increments $V_i[i]$ (basically it updates its own vector according to the received vector using *max* function).

A vector clock defines a perfect **isomorphism** with respect to the happens before relationship. Position i-th of the vector clock of each process represents the corresponding number of events that occur at process i-th.

4.2.2.1 Vector clocks for causal delivery We want to order questions and replies, messages in causal order, not in a total order.

Using vector clock, we can order events according to causality, not exactly causality. In order to do so, we need a variation of vector clock. Causal delivery: if two events are causally related, everybody must see the message in the same order. A slight variation of vector clocks can be used to implement causal delivery of messages in a totally distributed way.

We can use vector clocks:

- Variation: increment clock only when sending a message. On receive, just merge, not increment
- Hold a reply until the previous messages are received:
  - $ts(r)[j] = V_k[j] + 1$

    - $ts(r)[i] \leq V_k[i]$ for all $i \neq j$

4.3 Mutual exclusion

Mutual exclusion is required to *prevent* interference between processes in a distributed system.

- **Safety property:** says that at most one process exits the critical section at a time.
- **Liveness property**: all requests to enter/exit the critical section eventually succeed (no deadlock, no starvation)
- **Optional**: if one request happened-before another, then entry is granted in that order

4.3.1 Centralized solution

The simplest solution is to have a server to coordinate access to a resource. This server emulates a centralized solution where it manages the lock using a token, which allows a process to access the resource. Requests and releases for resource access are obtained through messages to the coordinator.

This solution is easy to implement and ensures mutual exclusion and fairness. However, it has some drawbacks: - the server can become a performance bottleneck - the server is a single point of failure

Actually the best option is this: just replicate the central server to address its limitations.

4.3.2 Mutual exclusion with Lamport scalar clocks

To request access to a resource, process P_i multicasts a resource request message m, with timestamp T_m, to all processes (including itself). A resource request is granted to a process when its request has been acknowledged by all other processes.

Upon receipt of m, a process P_j :

- If it does not hold the resource and it is not interested in holding the resource, P_j sends an acknowledgment to P_i
- If it holds the resource, P_j puts the requests into a local queue ordered according to T_m (process ids are used to break ties)

- If it is also interested in holding the resource and has already sent out a requests, P_j compares the timestamp T_m with the timestamp of its own requests
    - If the Tm is the lowest one, Pj sends an acknowledgement to Pi
    - otherwise it put the request into the local queue above

This protocol satisfies:

- the **safety** property
- **liveness** property, since it is guaranteed that each request will eventually be acknowledged.
- **optional** property as it guarantees access to the resource in Lamport clock order, which respects the happened-before order.

4.3.3 Token ring solution

Processes are logically arranged in a ring, regardless of their physical connectivity. Access to a shared resource is granted through a token that is passed along the ring in a specific direction. When a process does not require access to the resource, it forwards the token to the next process in the ring. To gain access to the resource, a process keeps hold of the token. Once a process has finished using the resource, it releases it by passing the token to the next process in the ring.

4.4 Leader election

In many distributed algorithms, a coordinator or special role is required. One example is server-based mutual exclusion. The problem arises when there is a need for a consensus on selecting a new leader when the old leader is no longer available, either due to failure or applicative reasons. The minimal assumptions for this scenario are: - the nodes are distinguishable, as without this distinction, it is not possible to perform selection. - the processes have knowledge of each other and their respective IDs. However, they do not have information about which processes are up and running or which ones have failed.

4.4.1 The bully election algorithm

The election algorithm works as follows: when a process P notices that the current coordinator is not responding, it initiates an election sending an `ELECT` message to all other processes with higher IDs. If no one responds, P wins the election and sends a `COORD` message to the processes with lower IDs. If someone with higher IDs responds, P doesn't win and in a recursive way, the other processes with higher IDs perform the algorithm.

4.4.2 A ring-based algorithm

In a ring topology among nodes, when a process detects a leader failure, it sends an `ELECT` message containing its ID to the next closest alive neighbor. The process receiving the election message follows these steps:

- If the process is not already in the message, it adds itself and propagates the message to the next alive neighbor.
- If the process is already in the message, the message type is changed to `COORD`, and the modified message is recirculated.

When arrive the `COORD` message, it means it has circulated around the entire ring. It then takes the list of IDs from all the processes and selects the greatest, lowest, or desired ID as the new leader (all processes obv must choose the same criteria).

After another round of message propagation, the leader will be elected. Multiple messages may circulate simultaneously but will eventually converge to have the same content.

4.5 Capturing global state

Capturing global state is a problem that one application of this case is the problem that we already encountered, the problem of creating a **snapshot of a system**. Capturing the global state of a distributed system is not as straightforward as it would be with a global clock. **Since a global clock is not available, we have to rely on recording the state of each process at different times.**

A cut C of a system S composed of N processes $p_1, ..., p_n$ can be defined as the union of the histories of all its processes up to a certain event. C is consistent iff for any event e it includes, it also includes all the events that happened before e.

4.5.1 Distributed snapshot (Chandy-Lamport)

The Chandy-Lamport algorithm selects a consistent cut. Any process p may initiate a snapshot by:

1. Recording its internal state
2. Sending a token on all outgoing channel to signal the start of the snapshot
3. Start recording local state (messages arriving on every incoming channel)

Upon receiving a token, a process q :

- Stop recording incoming message on the channel the token arrived along
- If not already recording local snapshot
    1. Records its internal state

2. Sends a token on all outgoing channels
3. Start recording a local snapshot

The end of the snapshot is "natural": at some point the token have arrived on all its incoming channels. Things to note:

- the application runs continuously during recording. If it receives a token from a channel which is being recorded, it will only pause the recording but continue processing the messages throughout the entire operation.
- Once the snapshot is complete, the collected data can be sent to a single collector that can reconstruct the global state of the system based on the individual process snapshots.

4.5.2 Dijkstra-Scholten distributed termination

In a **diffusing computation**, initially all processes are idle except the **init** process, which after the reception of a message/signal starts the distributed computation. The termination condition is that when processing is complete at each node and there are no more messages in the system the computation is terminated.

The **Dijkstra-Scholten** termination detection algorithm basically consists into creating a **spanning tree** to ensure that each successor is unique to only one node, without creating dangerous cycles. The core ideas:

- each node keeps track of the nodes it sends messages to: its children.
- If a node was already awake when the message arrived, it is already part of the tree and should not be added as a child of the sender.
- When a node has no more children and is idle means that it's a leaf node: it tells its parent to remove it as a child.

4.6 Distributed transactions

Different transaction types exist:

- **Flat**: happening on a single database, relatively easy to guarantee ACID
- **Nested**:
  - Each transaction consists in multiple sub-transactions in different (completely independent) DBs.
  - If a sub-transaction fails, then the entire original transaction fails
- **Distributed**:
  - Flat transactions on **distributed data**: the difference between nested transaction is that same the multiple DBs are part of the same database.
  - A single transaction should be able to read and write to all the data in all the distributed data stores
  - Need distributed locking

The main idea is to use a transaction manager which works as a "frontend" and interact with multiple schedulers which guarantee the legality of the transactions. The schedulers properly schedule conflicting operation using mainly two approaches:

- **Locking with 2PL** leads to serializability but maybe can cause a lock. 3 types:
  - Centralized **lock** manager
  - Primary: multiple lock managers
  - Distributed: distributed lock managers, so necessity to **synchronize** the locking over multiple hosts
- **Timestamp ordering** locks anything, perform everything with timestamps according to some criteria which are mainly divided into **pessimistic** ordering and **optimistic** ordering. The optimistic approach is different from the pessimistic one since it relies on the assumption that the conflicts are rare: maximum parallelism, eventually rollback. Not widely used in DS. Actually, not widely used in general.

4.6.1 Pessimistic timestamp ordering

Serializability without risk of deadlocks using a timestamp to each transaction (e.g., using logical clocks).

Some rules:

- We refer to the write timestamp of the committed version of x as $ts_{wr}(x)$ (That of the last transaction which write x)
- Same but with read timestamp $ts_{rd}(x)$ (that of the last transaction which read x)
- When **scheduler** receives write (T, x) at time $= ts$
  - If $ts > ts_{rd}(x)$ and $ts > ts_{wr}(x)$ perform tentative write x_i with timestamp $ts_{wr}(x_i)$
  - else abort T since the write request arrived too late
- When scheduler receives read(T, x) at time $= ts$
  - If $ts > ts_{wr}(x)$:
    * perform read and set $t_{rd}(x) = \max(ts, t_{rd}(x))$
  - else abort T since the read request arrived too late

4.6.2 Optimistic timestamp ordering

Based on the assumption that conflicts are rare, the strategy here is to simply proceed with the transaction and handle conflicts at a later stage:

- data items are marked with the start time of the transaction. When the transaction is ready to commit, a check is performed to determine if any items have been changed since the start of the transaction.
- If changes are detected, the transaction is aborted.
- Otherwise, it is committed.

Things to remember about optimistic timestamp ordering:

- Deadlock-free, allows maximum parallelism.
- Under heavy load, there may be too many rollbacks.
- Not widely used, especially in DS

4.6.3 Detecting distributed deadlocks

The 2PL locking can produce **deadlocks** (obv the timestamp approach can't produce deadlocks). Distributed deadlocks can be addressed mainly:

- Ignore the problem: most often employed, actually meaningful in many settings
- Detection and recovery: typically by killing one of the process
- Prevention
- Avoidance: never used in (distributed) systems, as it implies a priori knowledge about resource usage

Not only fix deadlocks is difficult, but even detecting deadlocks in DS is difficult. The approaches:

- **Centralized** solution consists in a **coordinator** which collects messages from each machine which maintains the own resource graph for its own resources. Then the coordinator with a "god-view" can "see" if there are deadlocks. The problem with these approach is that the timing of message arrivals can lead to false deadlocks.
- **Distributed** detection system (**Chandy-Misra-Haas**) where each process is allowed to request multiple resources simultaneously and send messages containing the tuple (initiator, sender, receiver). If any of the process detects a cycle, a deadlock is found.

A smarter solution to prevent deadlocks is to design the system in a way that makes it impossible for deadlocks to occur using a **distributed prevention** approach. It's possible to do this using **global timestamps** with two possible approaches:

- wait-die algorithm

- wound-wait algorithm

5 Fault tolerance

Another very broad and very typical system problem is fault tolerance. A **fault** can cause an error which eventually can cause a failure. For example a fault can be to not check zero division which gives the error division by zero.

Faults can be classified into:

- **Transient** faults which occur once and then disappear.
- **Intermittent** faults appear and vanish without any apparent reason.
- **Permanent** faults continue to exist until the failed components are repaired.

There are different types of **failure** models:

- **Omission** failures which can occur in processes
- **Timing** failures, which only occur in synchronous systems, happen when one of the time limits defined for the system is violated (for example, bounds on request's latency).
- **Byzantine** failures can occur in processes, where intended processing steps may be omitted or additional steps may be added. The "key characteristic" of byzantine failures is that weird results are produced.

Fault tolerance refers to the ability of a system or network to handle failures without causing a complete disruption. One of the main techniques used to achieve fault tolerance is redundancy. Redundancy can be implemented in different ways to mask failures:

- **Information** redundancy: instead of sending only the packet, we send the packet and also some additional information that may help in recovering a byzantine failure
- **Time** redundancy: TCP if does not receive an ack, tries to re-send the message later
- **Physical** redundancy: be redundant in physical layer, so use multiple channels

5.1 Protection against process failures

Redundant process groups are groups of processes which can collectively handle the work that should be done by an individual process. In this way, even if some processes fail, the remaining healthy processes can continue to work. Using process groups can be challenging to manage the membership of distributed groups. Multicast join/leave announcements are necessary but the problem is that if a process crashes, it will not notify others, which can complicate maintaining the group's integrity: if multiple processes crash or leave, the groups structure may need to be rebuilt.

In general, if processes fail silently, then $k + 1$ processes allow the system to be k-fault-tolerant . In case of Byzantine failures matters become worse: $2k + 1$ processes are required to achieve k-fault tolerance (to have a working voting mechanism). Byzantine means that you need other servers (the majority) to check that the first one has produced a weird result.

5.1.1 FloodSet algorithm

The FloodSet algorithm is used for fault tolerance in distributed systems. Each process begins with a variable called `W`, which is initialized with its own start value. In each round, each process sends its `W` to all other processes and adds the received sets to its own `W`. This process is repeated for `k + 1` rounds, where `k` is the maximum number of faults the system can tolerate (`k`-fault-tolerance): so for example to be 5-fault-tolerant, we need 6 steps to reach an agreement.

Each process may stop in the middle of the send operation.

After `k + 1` rounds a decision is made based on the size of `W`: - If the cardinality of `W` is 1, no problem at all - If the cardinality of `W` is greater than 1, it must be use a previously decided common function/criteria on all processes to make the decision on which value to take.

A more efficient implementation consists to broadcast `W` only if the process learns of a new value.

5.1.2 Lamport's algorithm intuition

In case of byzantine failure, things get more complex: the processes can both stop or exhibit any arbitrary behavior like sending arbitrary message or performing arbitrary state transitions. This problem has been described by Lamport in 1982 in terms of armies and generals.

During the initial round, all processes exchange their respective values. However among them there is a traitor which sends unusual values. Starting from the second round onwards, each process adopts the value that is most commonly shared among them.

Lamport (1982) showed that if there are m traitors, $2m + 1$ loyal generals are needed for an agreement to be reached, for a total of $3m + 1$ generals.

Fischer, Lynch, and Paterson proved that in general, for an **asynchronous** system, even a single failure is enough for not being able to reach a consensus: "**Impossibility of Distributed Consensus with One Faulty Process**" (**FLP** Theorem). Basically the FLP theorem says that every protocol that may come to your mind must be a **synchronous** protocol: so it must be a protocol that somehow fixes the bound in the sending of messages, fixes the bound in the processing speed of the chat-optic processes, and fixes the bound in the maximum delay jitter between the values cross.

5.2 Reliable group communication

Here the are the various alternatives for **reliable group communication** when processes are reliable but links are not.

5.2.0.1 Basic approaches to reliable group communication

- **ACKs**, where each recipient sends an acknowledgement after receiving the message. If an acknowledgement is not received, the sender resends the message. However, this can lead to an "ack implosion" if all recipients send acknowledgements simultaneously.
- **NACKs**, where ach recipient sends a NACK after a random delay indicating which packet was missed. This **optimistic** approach is more scalable as it prevents multiple NACKs from being sent simultaneously and reduces the likelihood of overwhelming the sender.
- **Hierarchical Feedback Control** is an evolution of using just ACKs and NACKS. The receivers are grouped together, with each group having a coordinator. These groups form a **tree structure** with the sender at the root. Coordinators manage acknowledgments and retransmissions within their groups and communicate with their parent coordinator if necessary. The challenge lies in maintaining the hierarchy, especially in the presence of dynamic group membership.

5.2.0.2 Communication in faulty processes Let's define the **atomic multicast** problem: a message is delivered to all non-faulty members of a group or to none, maintaining the same order of messages at all receivers.

Close synchrony cannot be achieve in the presence of failures: it says that multicast messages can be considered as instantaneous events and processes which receive the messages see events in the same order.

Virtual synchrony is the weaker model which replace the close synchrony:

- Only messages sent by **correct processes** are processed by all correct processes.
- Crashed processes are removed from the group and must rejoin later, reconciling their state.

- Messages sent by failing processes are either processed by all correct members or by none.
- The guarantees on the receiving order are only for relevant messages

Virtual synchrony model takeaways: - distinguishes between receiving a message and delivering it; messages are buffered by the communication layer delivered when certain conditions are met. - The concept of **group view** is crucial in this model: it's the set of processes that are considered part of the group at the time of sending a message. - Changes to the group view can be seen as another form of multicast messages (they must be consistent) - All multicast must take place between view changes

5.2.0.3 Message ordering Virtual synchrony is complex. On top of the previous explored protocol (or analogue protocols) we could retain the **virtual synchrony property** applying different orderings for multicast messages can be identified:

- **Unordered** multicasts
- **FIFO-ordered** multicasts: FIFO messaging ensures that messages from a single sender are delivered in the order sent. However, total FIFO ordering across all senders is not guaranteed without additional mechanisms.
- **Causally-ordered** multicasts
- **Totally-ordered** multicasts

5.3 Checkpoints

We are discussing fault tolerance and recovery techniques. There are two types of recovery:

- **Backward recovery** involves going back to a previous state if a state resulting from a crash is undesirable.
- **Forward recovery** involves trying to correct errors without going back to a previous state.

To enable backward recovery, we need to save previous states that can be retrieved later. **Checkpointing** involves periodically saving the state of each process without needing to synchronize with other processes (as synchronization is not practical in distributed systems). When recovering using checkpoints, it is important to ensure that transitioning to different states at different times still makes sense for the entire application: this is achieved finding what we call a "consistent cut" or recovery line. The main challenge is to identify the most optimal **recovery line**, which comprises consistent checkpoints from each process, thereby representing an overall consistent cut: going back to different states at different times but still making sense for the application as a whole.

5.3.1 Independent checkpointing

To implement and discover valid checkpoints, we need to obtain an approximation that allows us to reconstruct the best set of checkpoints for a good **recovery line**.

Before dive in the 2 seen algorithms, let's keep in mind that with $I_{i,x}$ we indicate the x-th interval between two checkpoints on process i . The 2 algorithms:

- **Rollback-dependency graph**:
  - this algorithm works starting from the "final" / "crashed" state view of the overall system
  - each dependency between the interval of checkpoints is translate it into a dependency between the ending checkpoint of the starting interval and the ending checkpoint of the arrival interval (ending to ending).It's like shifting towards the "right" all messages between processes, aligning them to the ending checkpoints of each interval.
  - from the final situation, you "mark" all the checkpoints reachable from the crashed process following the dependency relations you "wrote".
- **Checkpoint-dependency graph**:
  - First of all the dependencies are added from the **starting** state of the interval to the **final** state of the other interval (where the arrow arrives).

  - Now the **iterative** part:
    - You select the set with the checkpoints nearest the crash: this is called “hypothesis set”:
      - if there are no dependencies **inside** the hypothesis set a recovery line is found.
      - otherwise the **arrival checkpoint** of the dependency which is **inside** the hypothesis set is **discarded**. Then new iteration of the algorithm is done, selecting the **new** set: which is made from the previous checkpoints a part the one discarded.

Those are 2 **centralized** algorithms that should bring to the same result and are performed by a single central coordinator after a distributed collection of checkpoints from the processes. How works the distributed collections? We will see it in Synchronization chapter when we talk about Chandy-Lamport Distributed snapshot algorithm.

6 Distributed agreement in practice

6.1 Distributed commit

We will discuss the concept of Distributed Commit, specifically focusing on **atomic commitment**. In a DS, the challenge arises when we want to commit operations to databases that are partitioned.

Atomic commit refers to a transaction which refers to the **atomicity** ACID property: the transaction is either completely successful or completely unsuccessful, no intermediate state is possible.

Consensus	Atomic commit
One or more nodes propose a value	Every node votes to commit or abort
Nodes agree on one of the proposed value	Commit if and only if all nodes vote to commit, abort otherwise
Tolerates failures, as long as a majority of nodes is available	Any crash leads to an abort

- Termination:
  - If there are no faults, all processes eventually decide (weak)
  - All non-faulty processes eventually decide (strong)

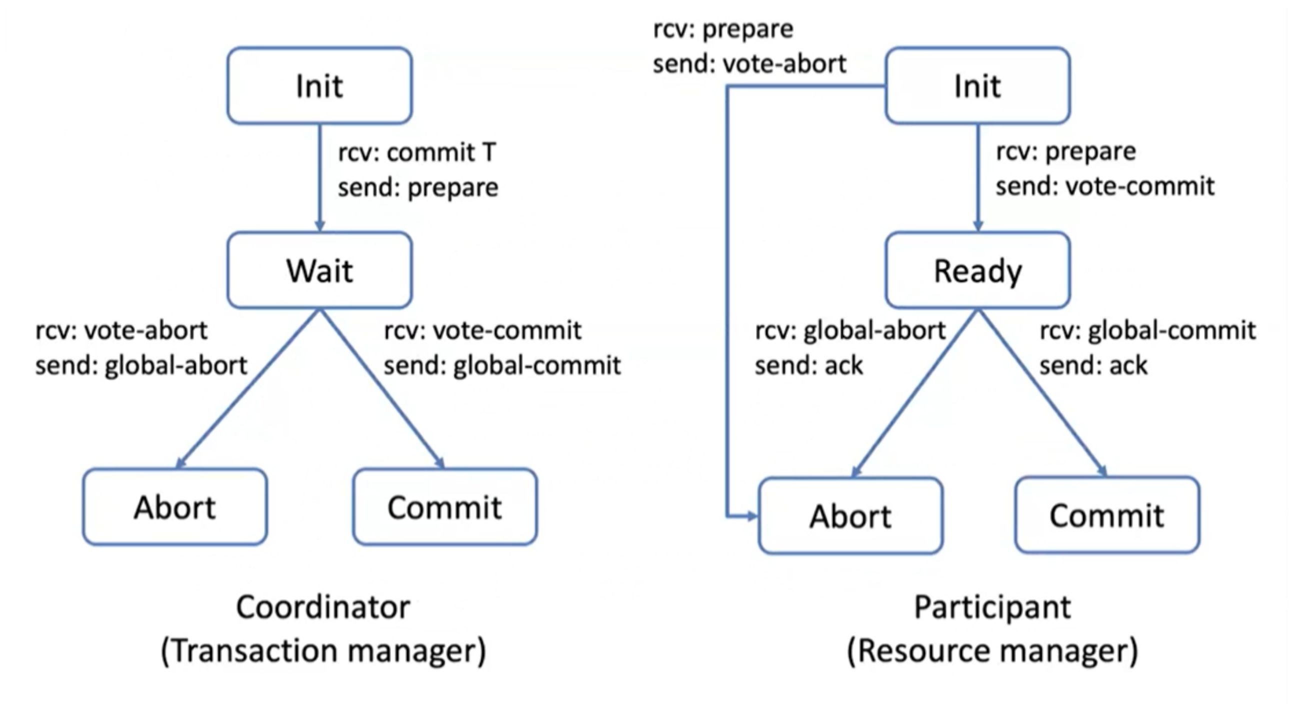
There are two different commit protocols:

- Two-phase commit (**2PC**): sacrifices liveness (blocking protocol)
- Three-phase commit(**3PC**): more robust, but more expensive so not widely used in practice

General result (FLP theorem): you cannot have both liveness and safety in presence of network partitions in an asynchronous system.

6.1.1 2PC

2PC is a blocking protocol which satisfies the **weak** termination condition, allowing to reach an agreement in less than $f + 1$ rounds.



6.1.1.1 2PC failure scenarios

1. Participant Failure:

- **Before Voting:**
  - If a participant fails before casting its vote, the coordinator can assume an **abort message** after a timeout
- **After Voting (Before Decision):**
  - If a participant fails after voting to commit, the coordinator cannot proceed to a global commit without confirmation from all participants.

2. Coordinator Failure:

- **Before Vote Request:**
  - Participants waiting for a vote request (in `INIT` state) can safely abort if the coordinator fails.
- **After Vote Request (Before Decision):**
  - Participants in the `READY` state, having voted but awaiting a global decision, cannot unilaterally decide. They must wait for the coordinator's recovery or seek the decision from other participants.

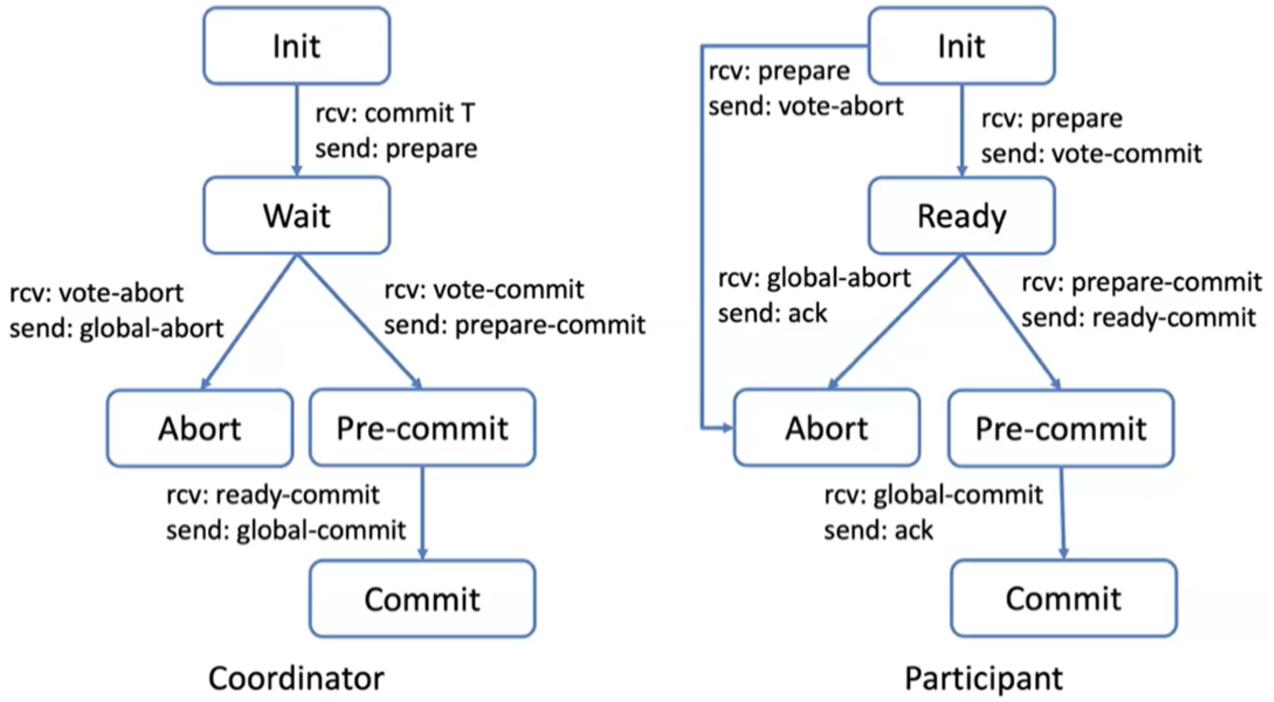
3. Consensus Inability:

- **Coordinator fail before decision make an indeterminate State:**
  - If the coordinator fails before sending a commit or abort message and any participant is in the `READY` state, that participant cannot safely exit the protocol.
  - Nothing can be decided until the coordinator recovers!
  - 2PC is vulnerable to a single-node failure (the coordinator)
  - Participants in the `READY` state are left in **limbo**, unable to commit or abort, demonstrating the inability of 2PC to reach consensus in the event of certain failures.

6.1.2 3PC

3PC is designed to overcome the blocking nature of 2PC by introducing an additional phase, which allows the protocol to avoid uncertainty even in the case of a coordinator failure.

- **Phase 1 (prepare):**
  - The coordinator asks participants if they can commit. Participants respond with their agreement or disagreement.
- **Phase 2 (prepare commit):**
  - If all participants agree, the coordinator sends a pre-commit message. Participants then prepare to commit but do not commit yet.
- **Phase 3 (global commit):**
  - Finally, the coordinator sends a global-commit message to finalize the transaction.



3PC reduces the risk of blocking by ensuring that participants can make a safe decision even if the coordinator fails, provided they have reached the pre-commit phase. 3PC is a non-blocking protocol, which satisfies the strong termination condition but **may** require a **large number of rounds** to terminate. The good thing is that with no failures, only 3 rounds are required. Also **timeouts** are crucial in 3PC to ensure progress in the presence of failures. Participants will proceed to the next phase or abort based on timeouts.

6.1.2.1 3PC failure scenarios

- **Coordinator Failure:**
  - If the coordinator fails before sending the pre-commit message, participants who have not received this message can safely abort.
  - If the coordinator fails after sending the pre-commit message but before the do-commit message, participants enter the 'uncertain' phase but will decide to commit once a timeout occurs, as they know that all participants were ready to commit.
- **Participant Failure:**
  - Participant failures are handled similarly to 2PC, but with the additional pre-commit phase providing a buffer that can prevent the system from entering a blocked state.

Termination Protocols in 3PC:

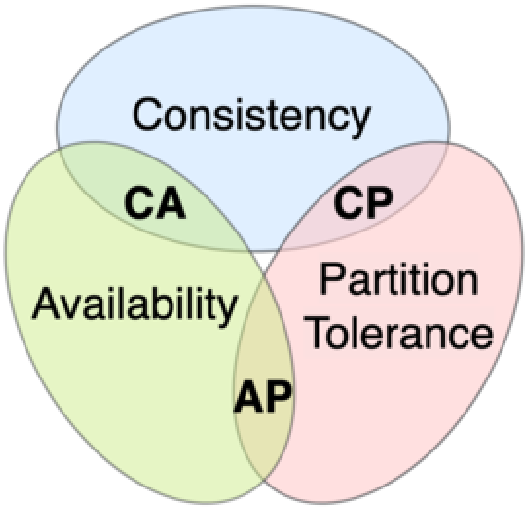
- **Coordinator Termination:**

    - If the coordinator recovers from a failure, it performs a termination protocol to determine the state of the participants and decide on the next step.
- **Participant Termination:**
    - Participants also have a termination protocol to follow if they recover from a failure, which involves communicating with other participants to determine the global state.

6.1.3 CAP theory

Any distributed system where nodes share some (replicated) shared data can have at most two of these three desirable properties - **C**: consistency equivalent to have a single up-to-date copy of the data - **A**: high availability of the data for updates (liveness) - **P**: tolerance to network partitions

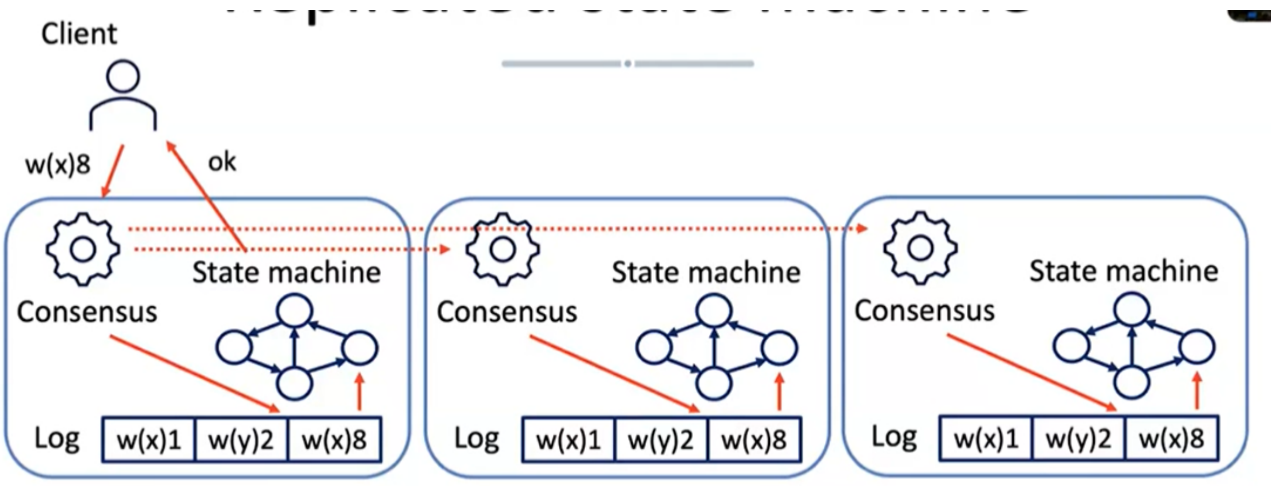
In presence of network partitions, one cannot have perfect availability and consistency.



For the *hodlers* out there we can say that the blockchain **trilemma** is a concept that was derived from CAP Theorem.

6.2 Replicated state machine

A general purpose consensus algorithm enables multiple machines to function as a unified group. These machines work on the same state and provide a continuous service, even if some of them fail. From the viewpoint of clients, this group of machines appears as a single fault-tolerant machine.



The idea is that the client connects to a leader which writes the operations onto a log. This log contains a sequence of operations that need to be propagated to the other nodes in the system through a consensus protocol. The ultimate objective is to maintain a **coherent view of the system** regardless of failures and network issues.

6.2.1 Paxos

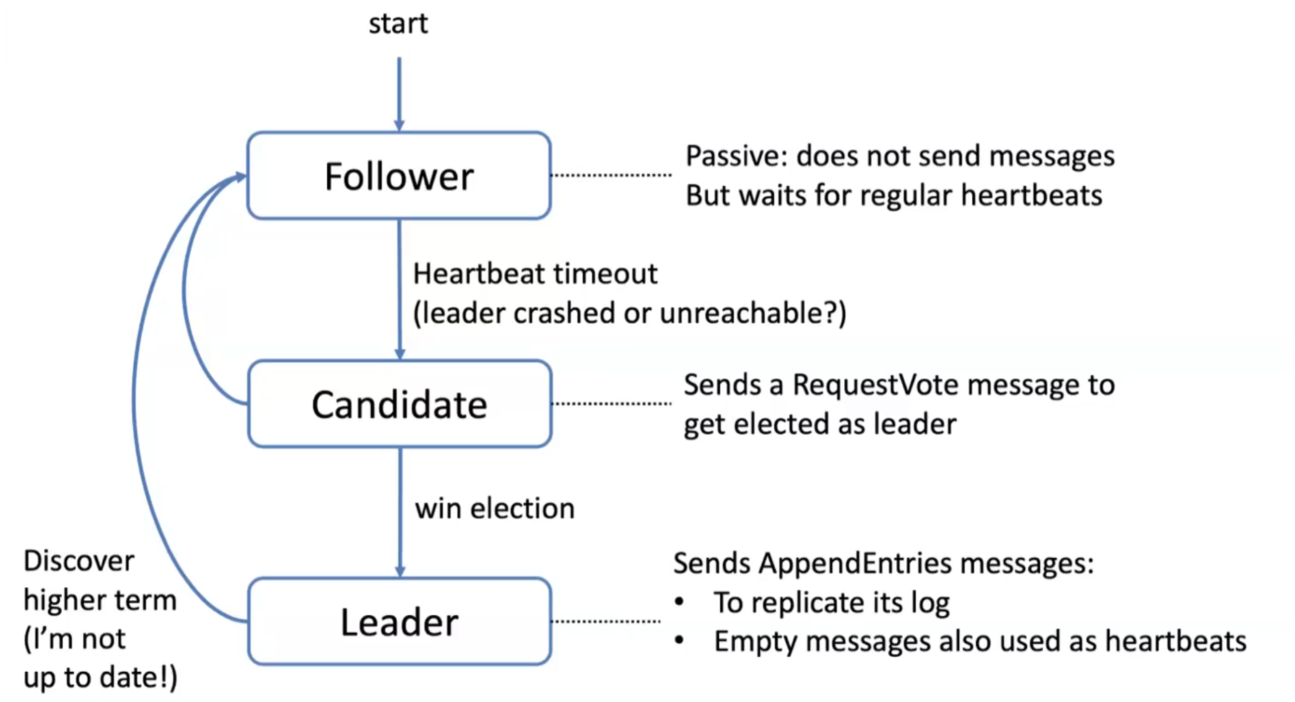
Paxos, proposed in 1989 and published in 1998, has been the reference algorithm for consensus for about 30 years. However, it has a few problems.

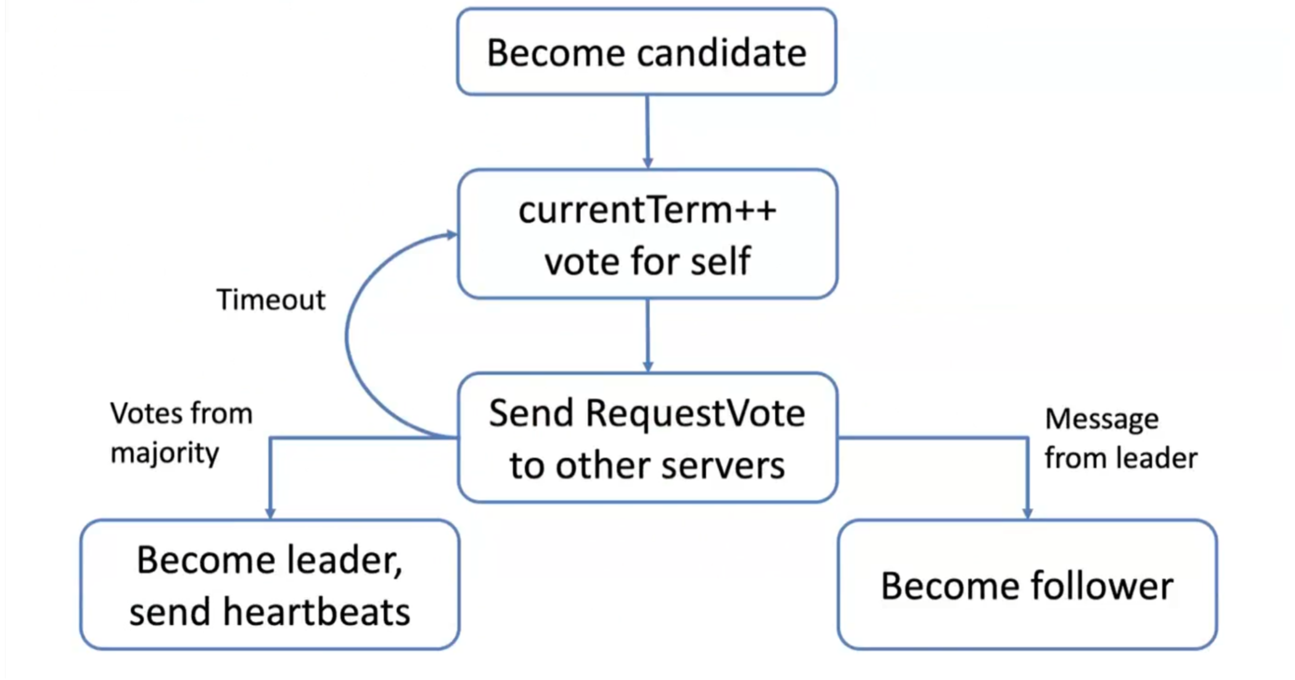
- allows agreement on a single decision, not on a sequence of requests. This issue is solved by multi-Paxos.
- Paxos is difficult to understand, making it challenging to use in practice.
- No reference implementation of Paxos: here is often a lack of agreement on the details of its implementation.

6.2.2 Raft

Raft is a consensus algorithm designed for managing a **replicated log**. It's used to ensure that multiple servers agree on shared state even in the face of failures.

- **Server States:**
  - Servers can be in one of three states: leader, follower, or candidate.
  - The leader handles all client interactions and log them
  - The log is then replicated and passive followers respond to the leader's requests.
  - Candidates are used to elect a new leader if the current leader fails.
- **Leader Election:**
  - In the event of a leader crash, a leader election process is initiated
  - Raft uses randomized timeouts to prevent multiple parallel elections from occurring simultaneously
  - If a follower receives no communication from the leader within a certain timeframe, it becomes a candidate and starts a new election.
- **Log Replication:**
  - Followers append entries to their logs only if they match the leader's log up to the newest entry.


start
Follower
Passive: does not send messages
But waits for regular heartbeats
Heartbeat timeout
(leader crashed or unreachable?)
Candidate
Sends a RequestVote message to
get elected as leader
win election
Discover
higher term
(I'm not
up to date!)
Leader
Sends AppendEntries messages:
• To replicate its log
• Empty messages also used as heartbeats



6.2.2.1 Log matching consistency

Log matching consistency is a property that guarantees the consistency of log entries across different servers. To achieve this Raft divides time into **terms** of arbitrary length: - Terms are numbered with consecutive integers - Each server maintains a current term value - Exchanged in every communication - Terms identify obsolete information

This property ensures that if log entries on different servers have the same `<index,term>`, they will store the same command.

Furthermore Raft implements **leader completeness**: once a log entry is committed, all future leaders must store that entry. **Servers with incomplete logs cannot be elected as leaders**. Meanwhile, in terms of communication with clients, it's guaranteed that clients always interact with the leader: this because when a client starts, it connects to a random server, which communicates the leader for the current term to the client.

6.3 Blockchains

Blockchains can be seen as **replicated state machines**, where the state is stored in the replicated ledger, which acts as a **log** and keeps records of all operations or transactions. Also blockchains can be modeled inside a "byzantine environment": misbehaving user (byzantine failures) attempting to **double spend** (or actions inconsistent with the state) their money by creating inconsistent copies of the log. Overall, the choice of approach depends on the trade-offs between search expressivity, performance, and network fragility.

7 Peer-to-peer

"Take advantages of resources at the edges of the network"

Alternative paradigm that promotes the sharing of resources and services through direct exchange between peers. Very popular in the early 2000s with hundreds of file-sharing applications developed.

7.0.1 Napster

Napster was the first P2P file sharing application that allowed users to download and share music for free over the Internet. Napster used small computers on the edges to contribute although they relied on the **centralized server** for lookup purposes (which some argued made it not a pure P2P system).

The main operations in Napster were: - joining - publishing - searching - fetching.

Napster's simplicity and `O(1)` search were advantageous. However, the **centralized** server for searching was a drawback since it was a single point of failure and a single point of control.

7.0.2 Gnutella

Gnutella has the advantage of being a fully decentralized network, eliminating the need for central coordination. This means that search costs are distributed among the network using a **query flooding** algorithm. When a node joins the Gnutella network, it connects to a known "anchor" node and sends a `PING` message to discover other nodes. These nodes respond with `PONG` messages, providing new connections to the joining node. The topology of the Gnutella network is **random** and constantly changing based on flooded queries. To join the network, a node needs to know the address of at least one anchor node. Peer nodes are used for both resource searching and recursively routing and flooding queries. To prevent congestion and endless queries, each query packet has a `HopToLive` field that decreases with each hop. Once this field reaches zero, the query stops. This parameter helps in limiting the scope of the search and prevents infinite loops or excessive network traffic.

7.0.3 KaZaA

KaZaA was created in 2001 from a Dutch company called Kazaa BV. Kazaa introduces a level of organization in the network by differentiating between regular nodes and "supernodes". In networks with a large number of nodes, simple query flooding (in Gnutella style) can quickly become inefficient and generate excessive traffic. Hierarchical query flooding can mitigate these issues by leveraging the capabilities of supernodes.

- Pros:
  - Tries to consider node heterogeneity
  - Bandwidth
  - Host computational resources

    - Host availability
    - Kazaa rumored to consider network locality
- Cons:
    - Still no real guarantees on search scope or search time
    - proprietary, not open source

7.0.4 BitTorrent

- Allows many people to download the same file without slowing down everyone else's download.
- Downloaders swap portions of a file with one another, instead of all downloading from a single server.
- **Pros**:
    - the tracking of the upload/download ratio gives peers an incentive to share resources. This mechanism in practice works and prevent free-riding.
    - **Cons**:
        * according to Approximate Pareto Efficiency if two peers get poor download rates for the uploads they are providing, they can start uploading to each other and get better download rates than before. But Pareto efficiency is actual a relatively weak condition.
        * central tracker server needed to **search** . The protocol main goal is indeed the sharing.

7.0.5 Freenet - Secure Storage

Freenet primary goal is **sharing** with an emphasis on secure storage and communication. The routing is intelligent, with a focus on the "small world" network paradigm. This means that while it doesn't exactly fit traditional P2P models, it might be closest to a mix of structured topologies and hierarchical systems (because of its emphasis on anonymity and its unique routing techniques). Main features:

- **Search Scope**:
    - **Join**: New nodes contact known nodes and receive a unique ID.
    - **Publish**: Files are routed towards nodes that store files with closely matching IDs.
    - **Search**: Uses a hill-climbing search with backtracking.
    - **Fetch**: Once a query reaches a node containing the desired file, the file is sent back.
- **Performance**: Intelligent routing ensures relatively short query durations, but no provable guarantees are given. Anonymity features may complicate metrics and debugging.
- **Security & Anonymity**: Emphasis on anonymity and security. Messages are forwarded and sources modified to ensure the anonymity of communication. Uses cryptography extensively for security.

Overall Freenet: **Pros**: - Intelligent routing ensures relatively short queries. - Provides anonymity. - Small search scope. - **Cons**: - No provable guarantees. - Anonymity features can complicate debugging and measurements.

7.0.6 Final comparison of Peer to Peer architectures

Compare the following approaches in terms of search expressivity and performance:

Example	Search expressivity	Performance	Fragility
Napster	**Centralized Search**	**Optimal** in terms of number of messages due to centralized nature	**Weak**: central server susceptible to takedown
Gnutella	**Broad expressivity** since local data can be searched, and users can specify the type of search they prefer (even based on content)	**Query Flooding** leads to inefficiencies	**Robust** compared to centralized

Example	Search expressivity	Performance	Fragility
Kazaa	**Limited**	Hierarchical Query Flooding optimizes search by node distinction (supernodes) but lacks real guarantees on search scope or time	**More robust** due to supernodes (assumptions about **supernode higher stability**)
Chord	**Limited** to the structure and rules of the DHT system (no queries based on content, only by key)	Structured Topology (DHT) gives efficient and predictable search times with logarithmic message number	**Robust**: queries can be forwarded to **successors if some nodes are offline**
Freenet	**Small search scope** as similar files are stored on the same node	Anonymity can complicate metrics	Designed for **robustness and anonymity**: decentralized and dynamic routing.

8 Replication and Consistency

Replication is useful for:

- more **performance** by allowing workload sharing and **reducing latency** for individual requests. It can also enhance **availability** by replicating data closer to users.
- more **fault tolerance** through **redundancy**

However, one of the main challenges in replication is ensuring **consistency** across replicas: changes made to one replica need to be propagated to all others, which can result in conflicts. The objective is to maintain consistency while **minimizing communication overhead**. Ideally you want the illusion of a single copy, but actually it's impossible and we have to rely on a **consistency model** is a **contract** between the processes and the data store.
Consistency models can be divided based on the promises made by the contract/protocol:

- **guarantees on content**: maximum difference between versions of different replicas
- **guarantees on staleness**: timing constraints over propagation to all replicas
- **guarantees on the order of the updates**: constraints of possible behaviors in the case of conflicts

Consistency protocols are responsible for implementing **consistency models**. These protocols are designed with various strategies in mind to handle different assumptions:

- **Single Leader Protocols**:
  - One replica is designated as the leader.
  - Clients send write requests to the leader
  - followers update synchronously, asynchronously or semi-synchronously:
    * sync if write operation completes only after the leader has received a confirmation from all followers. Safer but can lead to high latency.
    * async when the leader store the ops and all the follower updates happen asynchronously.
    * Hybrid solution is to consider an operation completed after confirmation from at least k replicas.
  - Single leader protocols are widely adopted in distributed databases like PostgreSQL, MySQL, etc.
  - No write-write conflicts; read-write conflicts still possible. **Multi Leader Protocols:**
  - Writes are carried out at different replicas concurrently.
  - No single entity decides the order of writes, leading to potential write-write conflicts.
  - Often adopted in **geo-replicated** settings. Contacting a leader that is not physically co-located can introduce prohibitive costs

– Multi leader protocols are more complex but can handle conflicts in many scenarios.
– Remember that client always contact a single node! **Leaderless Protocols:**
– Client contat multiple replicas to perform write/reads
– Clients contact multiple replicas for writes/reads.
– **Quorum-based** protocols are used to avoid conflicts, similar to a voting system where we need the majority of replicas to complete a write
– Leaderless replication is used in modern key-value/columnar stores like Amazon Dynamo, Riak, Cassandra.

8.1 Data-centric consistency models

It is quite difficult to have a precise definition of **consistency** in the context of data-centric models. Here we consider as a contract dictating the guarantees on content, staleness, and update order.

Consistency	Description
Strict	Any read on data item x returns the value of the most recent write on x
Linearizable	All processes must see all shared accesses in the same order. Operations behave as if they took place at some point in (wall-clock) time.
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time.
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order.
Eventual	Updates are guaranteed to eventually propagate to all replicas, assuming no new updates are made to the given data item.

From the CAP theorem, it is not possible to simultaneously achieve both consistency and availability. Consistency refers to all replicas in a system having the same data at any given time, while availability refers to a system's ability to continue operating despite failures. Strong consistency models such as linearizability offer strong consistency but may have higher latency. On the other hand, weaker models like eventual consistency are less costly.

8.1.1 Strict consistency

"Any read on data item x returns the value of the most recent write on x"

All writes are instantly visible and the global order is maintained. However, determining the "most recent" write is only possible within a single processor machine and in a DS (without a global time) is ambiguous.

8.1.2 Linear consistency (linearizability)

Linearizability is a strong form of consistency, it ensures that all operations appear to occur instantaneously and exactly once at some point between their invocation and their response. It's a **real-time** guarantee.

"The system is **sequentially consistent** and also if $ts_{OP_1}(x) < ts_{OP_2}(y)$ then operation $OP_1(x)$ precedes $OP_2(y)$ in the operation sequence"

If one operation appears to happen before another from any global perspective, then every process in the system must agree on this order.

Linearizability is useful in scenarios where the application logic requires a certain ordering between operations to be enforced, and all writes become visible (as if they were executed) at some instant in time, maintaining a global order

The difference between **strict consistency** and **linearizability** is subtle and often **theoretical** because, in practice, strict consistency is generally not achievable. Linearizability is often the strongest **practical** consistency model implemented in distributed systems.

8.1.3 Sequential consistency

Processes can agree on a sequence of operations, regardless of the real-world clock time. This agreed-upon sequence preserves the semantics of the writes and reads. Although the DS itself may not have a real clock, we can imagine ourselves outside the system, observing the real order of operations. Let's assume that the x-axis of the schedule represents the definition of real time.

> "The result is the same as if the operations by all processes were executed in some sequential order, and the operations by each process appear in this sequence in the order specified by its program"

The schedule is sequential, but ... At this point in time B thinks x is already $1, C$ thinks x is still 0. If they communicate (through a different channel), they break the **illusion of a single copy**.

In practice:

- Use a single coordinator (single leader replication):
  - Sequential consistency **limits availability** since it's necessary to contact the leader (which might be further away from the client) which must propagate synchronously the update to the replicas to achieve fault-tolerance
  - **No tolerance for network partitions**: in case of net. part. clients are blocked or leader is blocked to contacts followers
- **Distributed agreement**: the use of leaderless protocols which are quorum-based where for each operation it's necessary a **quorum** (*quò · rum* is the quotient, in numbers or percentages, of the votes cast or of the voters, required for an election or resolution to be valid) of the servers which agrees on the version number of a resource. With NR (NW) number of replicas that the clients contact to read (write) and N the number of all replicas:
  - $NR + NW > N$ ensures that the sets of replicas involved in read and write operations overlap. It means that at least one replica is common between read and write sets, helping **to avoid read-write conflicts**.
  - $NW > \frac{N}{2}$ ensures that more than half of the replicas must agree for a write operation to be committed; **to avoid write-write conflicts**

In practice, the quickest way to determine if a sequence of operations is sequentially consistent is to identify a valid **interleaving** of operations that is acceptable. It's important to note that an interleaving is valid only if all processes observe the same sequence of operations. When dealing with multiple variables, you need to consider each one individually, as various cases and patterns can emerge.

8.1.4 Causal consistency

Causal consistency is a weaker form of consistency when compared to linearizability or sequential consistency but provides a **balance between availability and consistency**.

> "Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in any order at different machines."

Causal consistency indeed weakens sequential consistency based on Lamport's notion of happened-before and it's not a total order but a **partial order**. This means that only causally related operations need to be ordered with respect to each other, while concurrent operations can appear in any order). Causal consistency is favored in DSs because it is easier to guarantee with smaller overhead and is easier to implement compared to stronger consistency models. Remember that causal order is **transitive**: it's important to know for understanding how causal relationships are established across different operations.

In practice, in the exercises, for each couple of "write-write" and "read-write" in each process, you make a constraint which has to be respected by the reads of all processes.

8.1.4.1 FIFO consistency

> "Writes done by a single process are seen by all others in the order in which they were issued; writes from different processes may be seen in any order at different machines"

Super simple consistency where causality across processes is dropped. It's implied by Causal consistency.

In practice, in the exercises, for each couple of "write-write" in each process you make a constraint which has to be respected by the reads of all processes.

8.1.4.2 Eventual consistency There are scenarios where simultaneous updates are unlikely and where read operations are more prevalent: examples include web caches, DNS, and geo-distributed data stores like Facebook/Instagram. In these types of systems, eventual consistency is often deemed satisfactory, as it guarantees that updates will **eventually** propagate to all replicas:

- Very easy to implement
- Very few conflicts in practice
- Today's networks offer fast propagation of updates

This is widely used in practice, in scenarios where the order of messages is not important. Eventual consistency doesn't imply any fifo/causal/sequential consistency.

8.2 Client-centric consistency models

Client centric consistency models take the pov of a single client who is reading/writing from multiple locations.

Concept	Definition	Example
Monotonic Reads	Subsequent reads by a process will always see the same or newer values.	Reading a forum thread will always show the latest replies, never reverting to older posts.
Monotonic Writes	Writes by a process are completed in order.	Comments on a blog are published in the order they're written by a user.
Read Your Writes	A process will see its own writes in successive reads.	After posting a tweet, you'll see your tweet when you refresh the page.
Writes Follow Reads	Writes by a process reflect the latest read value.	Replying to a message only after you've seen the most recent messages in the conversation.

8.2.1 Client-centric consistency implementations

In this scenario, each operation has a unique identifier (ReplicaID + a sequence number) and there are two sets assigned to each client:

- the **read-set**: write identifiers relevant for the read operations executed by the client

- the **write-set**: the identifiers of the write performed by the client.

These sets can be represented using **vector clocks**, which keep track of the latest read/write identifiers from each replica.

8.3 Design Strategies

8.3.1 Replica placement

Type	Description
Permanent Replicas	Statically configured, used in systems like DNS and CDNs.
Server-Initiated	Created dynamically to cope with access load, moving data closer to clients.
Client-Initiated	Rely on client cache, can be shared among clients for enhanced performance.

8.3.2 Update Propagation

8.3.2.1 What to Propagate

Method	Description
Notification Only	Update performed, only notification propagated; used with invalidation protocols. Best if `#reads << #writes`.
Transfer Modified Data	Transfer the modified data to all copies; efficient when `#reads >> #writes`.
Enable Update Operation	Propagate information for the update operation at other copies (active replication).

8.3.2.2 How to Propagate

Approach	Description
Push-Based	Update propagated to all replicas, regardless of need; preserves high consistency.
Pull-Based	Update fetched on demand; convenient if reads $<$ writes, manages client caches.
Leases	Used to switch between push and pull approaches.

	State of server	Messages sent	Response time at client
Push-based	List of client replicas and caches	Update (and possibly fetch update later)	Immediate (or fetch-update time)
Pull-based	None	Poll and update	Fetch-update time

8.3.2.3 Propagation Strategies

Protocol	Description
Leader-Based	Synchronous, asynchronous, or semi-synchronous propagation.
Leaderless	Includes read repair and anti-entropy processes.

8.4 Case studies

8.4.1 Case study: Spanner

Spanner is a globally-distributed database developed by Google. It is specifically designed to handle very large databases, utilizing a partitioned approach where each partition is replicated.

- **Design**: Spanner is designed for very large databases with many partitions, each of which is replicated.
- **Techniques**: It uses standard techniques:
  - **single-leader replication** with **Paxos** for fault-tolerant agreement on followers and leader
  - **2PC** for **atomic commits**
  - **timestamp protocols** for concurrency control
- **TrueTime**: Spanner's novelty lies in TrueTime, which uses very precise clocks (atomic clocks + GPS) to provide an API that returns an uncertainty range. The "real" time is guaranteed to be within this range, which is crucial for deciding when to commit read-write transactions.
- **Transactions**: Read-write transactions use TrueTime to decide when to commit, waiting until the uncertainty range has certainly passed. This ensures transactions are ordered based on time, achieving **linearizability**. Read-only transactions also acquire a timestamp through TrueTime, allowing them to read the latest value at that time without locking, optimizing frequent read-only operations

8.4.2 Case study: Calvin

Calvin is designed for the same settings as Spanner. It adopts a sequencing layer to order all incoming requests, both read and write.

- **Guarantees**: Calvin provides **linearizability** through a sequencing layer that orders all incoming requests, both read and write.
- **Operation**: It uses a replicated log implemented using Paxos, requiring operations to be deterministic and executed in the same order everywhere.
- **Advantages**: The system reduces lock contention by achieving agreement on the order of execution before acquiring locks and eliminates the need for 2PC because transactions are deterministic (they either succeed or fail in all replicas)

8.4.3 Case study: VoltDB

- **Developer Role**: Developers specify how to partition database tables and transactions. Specifying this allows the database to organize data efficiently based on query hints.
- **Execution**: Single-partition transactions can execute sequentially on that partition without coordinating with other partitions.

9 Big data

Data science relies on the availability of large volumes of data, commonly known as "big data." Big data comes from various sources such as recommender algorithms, genomic data, and medicine. For instance, Netflix's recommendation system utilizes historical viewing data, while Google Translate uses translations of books in various languages. Tech giants have built business models around data. They provide free services to users and monetize the data by selling it to advertisers. Amassing vast amounts of data enables more statistically significant insights and is often more cost-effective than selectively retaining data. Big data is characterized by the "4 V's":

- Volume (the quantity of data)
- Velocity (the speed of data inflow)
- Variety (the range of data types and sources)
- Veracity (the accuracy and reliability of data).

Handling big data effectively requires:

- Automatic parallelization and distribution of data processing
- Fault tolerance
- Status and monitoring tools
- Clean programming abstractions

9.1 Map Reduce

The MapReduce programming model, introduced by Google in 2004, allows application programs to be written using high-level operations on immutable data. The runtime system handles scheduling, load balancing, communication, and fault tolerance.

- In **Map** phase, individual elements are outputted to one or more `<key, value>` pairs.
- In the **Reduce** phase, the values with the same key are reduced to a single value.

The map function is **stateless** and its output depends only on the specific word received as input: **reducers** receive an immutable list (iterator) of values associated with each key. This ensures that the data is read sequentially from the disk only once. Designing an algorithm that updates data only once and avoids excessive state storage can result in improved performance in the given context. Typical applications involve:

- Sequences of map and reduce steps
- Data transformations
- Iterations until convergence (e.g., Google PageRank)

Strengths of MapReduce are:

- Simplified programming model
- Automated management of complex tasks (allocation, synchronization, etc.)
- Versatility
- Suitability for large-scale data analysis

Limitations of MapReduce are:

- Higher overhead compared to High-Performance Computing (HPC)
- Inherent performance constraints
- Rigid structure; each step must complete before the next begins

9.1.1 Support functionalities

MapReduce platforms provide us many functionalities:

- **Scheduling**: Efficient allocation of tasks to mappers and reducers, with consideration for data locality. The master node is responsible for scheduling tasks, monitoring their execution, and reassigning tasks in case of worker node failures.
- **Data distribution**: Optimization of data transfer from mappers to reducers to minimize network bandwidth consumption. The Google File System (GFS) uses a block-based approach to store data files by dividing them into 64MB blocks. Each block is then replicated and stored on three different machines. GFS avoids limitations caused by rack switches and enables a higher read rate by considering location of these replicas and taking into account data distribution.

- **Fault tolerance**: MapReduce handles transparently node failures by re-executing failed tasks. The master node monitors worker nodes and reassigns tasks as necessary.

9.2 Beyond MapReduce

Over the last decade, the MapReduce framework has undergone several advancements with an introduction of new dataflow abstractions. Recent improvements in big data processing include:

- Support for complex transformation graphs
- Transition from batch to stream processing
- Shift towards in-memory and hybrid processing methods

Key features of modern platforms:

- arbitrary number of stages join, filter, groupBy besides map and reduce
- Caching intermediate results if reused multiple times.
- **Apache Spark** adopts a **scheduling** approach which is focused on "*data parallelism*". This optimize **throughput**, reduce overhead and eventually compress data. The scheduling approach of Spark simplifies **load balancing**. Spark can perform in-memory processing and supports arbitrary acyclic graphs of transformations, which can lead to faster execution times compared to the on-disk batch processing of MapReduce.
- **Apache Flink** adopts a **pipelined** approach which basically means "*task parallelism*". This makes Flink ideal for stream processing because its **lower latency** approach. Load balancing is basically impossible since everything is statically decided when the job is deployed and there isn't scheduling.
- **Elasticity** refers to the ability to adjust resource usage dynamically. Elasticity is about **scaling the resources** to meet the demands, while load balancing is about efficiently distributing the workload across the available resources. Elasticity is better supported by systems with dynamic scheduling (like Spark) than by pipelined approaches, because scheduling decisions take place at runtime while elasticity in pipelined approaches is not possible.
- **Fault tolerance** in Big Data processing platforms is achieved in 2 ways:
  - if the processing is **scheduled**: Re-scheduling and **re-compute** if a node fails.
  - if the processing is **pipelined**: **checkpointing** to a distributed file system is the way.

MapReduce can be still an appropriate choice for simple batch processing tasks, where the overhead of Spark's in-memory processing or Flink's streaming capabilities is not justified.