Distributed Systems

Niccoló Didoni

September 2021

Contents

Part I

Introduction to distributed systems

Chapter 1

Distributed systems

1.1 Definitions

Before talking about distributed systems and the challenges their development has, we have to define what a distributed system is.

Definition 1 (Distributed System). *A distributed system is a collection of independent computers that appear as a single system.*

From this definition we can understand that for a user, and sometimes for a programmer, a distributed system should be transparent, hence it should not display the information regarding the implementation of the system. A distributed system should always look like a single machine, even if it works thanks to the collaboration of multiple machines connected through a network and maybe located in different parts of the globe.

Another definition of distributed system is

Definition 2 (Distributed System). *A distributed system is a system in which hardware and software located at networked computers communicate and coordinate their actions only by passing messages.*

This definition is a little more precise than the previous one and points out two important characteristics of a distributed system

- Computers in a distributed system **are connected through the network**, so, differently from concurrent systems (in which more processors works in parallel looking like one faster processor), distributed systems do not share a clock signal. This could rise problems when synchronising the processes.
- Computers in a distributed system communicate **only via messages** (typically TCP messages) exchanged via the internet.

1.2 Middle-ware

Distributed systems are very complex then to deal with such complexity the developers can use libraries and daemons that hide the complexity of distribution. Such aid are called middle-ware.

1.3 Characteristics

A distributed system

- Is **concurrent**. Even if every component of the system is single threaded, the whole system is multi threaded because it uses more than one thread (e.g. if three computers running a single thread each make a distributed system, the system runs on three different thread, so it is a multi threaded system).
- Does not have a **global clock**. If distributed systems are multi-threaded then they need synchronisation. Differently from parallel systems, distributed systems do not share a common clock signal so it is not possible to use the classic methods of synchronisation like semaphores and locks. Finding others methods to handle concurrency is one the most important challenges in distributed systems.
- Has **independent fails**. When a centralised application fails because of a hardware problem, the developer cannot do nothing to recover, because the machine which should run the recovery code is broken. However in a distributed system more machines can balance the computational load, then when a machine fails, the other machines can share the tasks allocated to the broken computer. As an example let's think of a web server, when a user wants a certain HTML page, the request is handled by the nearest server. If this server is currently broken, the request is flawlessly redirected to another web server without the user knowing anything.
- Is **heterogeneous**. In a distributed system every component might use different programming languages, protocols or processors depending on his tasks (e.g. a component of the system might save numbers in big endian, another in small endian. When the two computers communicate this has to be kept in mind).
- Is **open** to the other computers on the network.
- Is **secure**. This property comes from the previous point, in fact if a system is open to every machine on the network we must ensure that only those allowed can communicate with the system.
- Is **scalable**. A system's dimension can be increased or decreased without changing the structure of the system.
- Is capable of **handling failure** without the user knowing that an error occurred somewhere in the system.
- Is **transparent**. This means that the user doesn't have to know that an application is part of a more complex system.

Part II

Distributed systems models

Chapter 2

Architectures

When developing a software an UML is usually enough to model its structure and behaviour. On the other hand a distributed system requires more different models to describe the architecture and the behaviours of such a complicated system. In particular the main models are

- The **software architecture**.
- The **run-time architecture**.
- The **interaction model**.
- The **failure model**.

2.1 Software architecture

The software architecture focuses mainly on the software engineering part of the system.

The software architecture of a distributed system can be

- **Network OS** based.
- **Middleware** based.

2.1.1 Network OS based

This architecture is based on the operative systems' network primitives. This architecture is not transparent, in fact we have to use different primitives depending on the type of OS and of the machine on which it runs. In other words this architecture is not platform independent and it's up to the user developing an application that looks like a single machine.

2.1.2 Middle-ware based

This architecture is based on the middle-ware. A middle-ware is a set of libraries and daemons that abstract the complexity of the network OS primitives. As an example Java's Remote Method Invocation is a middle-ware because it allows to execute the methods of a class which is physically allocated on another machine without caring how Java does that (we do not have to establish a TCP connection or use sockets, Java handles it automatically).

Using a middle-ware allows the user to be mostly platform-independent and therefore it is easier to create a system that looks like a single machine.

2.2 Interaction model

The interaction model describes the behaviour of the distributed system. A normal program can be described with an algorithm (i.e. a sequence of operations) but the behaviour of the system cannot be described only with the algorithms running on each process of each component because each part of the system interact in a complex way (by exchanging messages) with the other parts.

To describe a distributed system we have to use a distributed algorithm, that is a description of the steps taken by each process including the transmission of messages between the processes. A distributed algorithm doesn't only focuses on the algorithm of each process and on the messages exchanged but also on

- The rate at which each process proceeds.
- The performance of the communication channel.
- The different clock drift rates.

These properties are important because processes exchange messages and if a process is slower, its messages leave after other messages. But messages are instructions in a distributed system so it's important to know their order just like it's important to know the order of the instructions in an algorithm.

2.2.1 Synchronous and asynchronous systems

Systems can be divided in

- **Synchronous systems** in which the rate of execution, the performance of the channel and the clock drift rates are not known but they have well known bounds (i.e. we fix a bound on the time to execute each instruction or on the time between when a message is sent and when it's received).
- **Asynchronous systems** in which there are no bounds for process execution, message transmission and clock drift rates. For asynchronous distributed systems we can't say if a distributed algorithm is correct (i.e. we can't prove that an algorithm executes correctly).

In reality it is not possible to fix some bounds, in fact most of times the constraints are satisfied but it may happen that some actions exceed the time limits. For this reason all systems are asynchronous and therefore we can't say if an algorithm is correct. To solve this problem we can assume a system is synchronous because the chance that a system exceeds the time limits is very low and we try to demonstrate that a certain algorithm is valid for such synchronous approximation.

The Pepperland example

To highlight the problems related to asynchronous systems we can use the Pepperland example. Consider two divisions of soldiers leaded by two generals. Each division is on top of an hill and the division are divided by a valley in which we can find an enemy division. The generals have to attack the enemy division but they can succeed only if the attack together so the generals have to agree on who has to lead the attack and at what time the charge will take place. The generals can communicate using messengers.

The first problem can be solved even in an asynchronous system (for example the generals could use a random number and the leader is the one that selects the bigger number). On the other hand it's impossible to agree on the time to charge because there is no guarantee on the time the message needs to cross the valley.

2.3 Failure model

The failure model describes the failures that may happen to the processes and to the communication channels. Failures are divided in

- **Omission failures**.
- **Byzantine (or arbitrary) failures**.
- **Timing failures**. Timing failures are valid for synchronous systems only.

The failure problem is important because some protocols perform better when only some types of failures can occur.

In an asynchronous system it's impossible to detect failure because there is no way to know if a message is actually lost or it's just taking a long time to reach the destination.

2.3.1 Omission failures

An omission failure verifies when something that should have happened doesn't happens. Omission failures are valid for all types of systems.

Omission failures for communication channels Omission failures for channels means that a message sent (or that should have been sent) by a component of the system gets lost. Omission failures for channels can be further divided in

- Send omission if the message doesn't reach the cable.
- Channel omission if the message doesn't reach the other end of the cable.
- Receive omission if the message is lost at the receiving point.

In all these cases the final effect is the same.

Omission failures for processes Omission failures for processes means that a process crashes during execution. For processes, omission failures are more common that byzantine failures. Omission failures for processes can be divided in

- **Fail-safe** failures.
- **Fail-stop** failures, i.e. crashes that can be detected.
- **Fail-silent** failures, i.e. crashes that can't be detected.

2.3.2 Byzantine failures

A byzantine failure verifies when it gets executed something that shouldn't have. Byzantine failures are valid for all types of systems.

Byzantine failures for communication channels Byzantine failures for communication channels happen when a component receives a packet that differs from the one that had been send. For communication channels byzantine failures are more common that omission failures. In some protocols byzantine failures, that verify at hardware level, are transformed in omission failures (e.g. if IP finds an error in a packet it discards such packet).

Byzantine failures for processes Byzantine failures for processes happens when unexpected instructions gets executed. For instance this type of failure can be caused by memory corruption. Byzantine failures in processes usually get transformed in omission failures (e.g. after a memory corruption a program usually crashes).

2.3.3 Timing failures

A timing failure verifies when one of the timing bounds defined for the system is violated.

Chapter 3

Run-time architecture

The run-time architecture describes

- Which components are present at run-time (i.e. during a specific execution of a distributed system).
- How these components interact at run-time.

Architectural styles Run-time architecture, other than defining run-time components, defines the type of architecture. The architecture is independent from the run-time distribution of components. For example a web server remains a client-server architecture independently from the number of devices that are requesting a page (run-time architecture). The most common architectures are

- **Client-server**
- **Service Oriented**
- **REST**
- **Peer-to-peer**
- **Object-oriented**
- **Data-centred**
- **Event-based**
- **Mobile code**
- **CREST**

It is also important to specify that a distributed system doesn't have to adopt one of this architecture exclusively but can have the characteristics of more architectures.

3.1 Client-server

The client-server architecture is one of the most used nowadays (the web is based on this architecture).

In the systems that adopt this architecture the components can be divided in two categories

- The **client** that usually represents the user and has an active role in the system (the user makes queries to the server).
- The **server** that has a passive role (replies to the requests of the clients).

With the expansion of the web the client-server architecture has evolved introducing the tiers. A tier represent a machine or a logical function. Tiers are divided in

- **Graphical interface**
- **Application services**
- **Data**

Each tier can be located both on the client and on the server (even in both). For example the user's computer (i.e. the client) could handle only the graphical representation of a web page, while the web server could handle the logic to build the page and the data to build the page. In this case the architecture is a three-tier (because there are a GUI, an application logic and a data tier) client-server.

The a tier could also be divided between client and server, in the previous example the application tier could have been divided between client and server.

The architecture can also be expanded adding more tiers, for instance we can add a script engine that handles the scripting part of the request. In this example we have a 4 tier (a GUI, the application logic, the script engine and the DBMS to handle the data).

Layers and tiers Let us take a second to analyse the difference between layers and tiers.

- Tiers are physical (even if they can be seen as logical) units in which a system can be divided. Tiers can be seen as deployment units, but it's not necessary to have a one on one relationship between a tier and a physical machine (i.e. more tiers can be deployed on the same machine).
- Layers are logical units in which the code can be divided. Layers are usually organised in a stack and communicate using software interfaces.

3.2 Service oriented

The service oriented architectural style is built around the concepts of

- **Service**. A service is a loosely coupled unit of functionality.
- Service **provider**. A service provider exports a service.
- Service **consumer**. A service consumer binds to a provider to use a specific service.
- Service **broker**. A service broker collects the services offered by the providers and allows the consumers to find such services. In particular providers publish services to the broker and consumers can search and find such services on the broker.

Implementation The service oriented architecture is used by many systems like **OSGI** (Open Grid Services Infrastructure), JXTA, Jini and Web Services.

3.2.1 Web Services

A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network.

Interface The interface of a Web Service is described using WSDL (Web Service Description Language). WSDL is a machine readable language.

Operations The operations executed by a Web Service are invoked through a XML-based protocol called SOAP (Simple Object Access Protocol). In particular SOAP defines the way messages are exchanged. SOAP is based on HTTP but other protocols can be used too. Namely other machines can interact with a Web Service using SOAP.

Rules UDDI (Universal Description Discovery and Integration) describes the rules that allows Web Services to be exported and searched through a registry. A registry is the service broker described by the service oriented architectural style.

3.3 REST

REpresentational State Transfer is a set of principles that defines how the Web should be. The key goals behind REST are

- **Scalability**.
- Offer **general purpose interfaces**.
- Offer **independent deployment components**. The architecture should also be open to very large deployments.
- To allow an **easy introduction** of intermediate components that can be used to add or enhance some features and functionalities like security or latency reduction.

REST is based on the following key ideas

- It is based (it's a specialisation) on the **client-server architecture**.
- **Interactions are stateless**, in fact they happen in a client-server way but the servers (i.e. the machines that offer a service) should be stateless. In other words if a client does the same interaction it always get the same value. Another important property is that there should also be no state maintained in the connection.
- The data within the response to a request should be implicitly or explicitly labelled as **non cacheable** or **cacheable**.
- If the system is layered, **each layer should be strict** (should have visibility only on the layer below).

- Client must support **code on demand**. In other words clients should accept code coming from servers and should execute such code (e.g. JavaScript).
- Components should expose a **uniform interface**. In other words the through which we interact should be the same independently from the component.

Resources

A resource is something that can be identified, named, addressed, handled, or performed on a server. Resources aren't only documents or files but also services or functions. For instance say we want to implement a calculator. We can access the sum functionality with the following request

```
http://localhost/calculator/add?a=10&b=20
```

to sum the numbers 10 and 20. The URL is the resource, in particular it is a functionality.

When a user requests a resource to a server, it sends back a textual representation of the resource. For this reason we can say that resources are logical elements and we only interact with some **representations** of the resources. For instance in our example

- The resource is the adding functionality.
- The representation of the resource is the result of the computation (i.e. the number returned by the interaction). Such representation can be in different formats (for instance two different devices could get the data encoded in different ways).
- The representations might be in different formats but they all refer to the same concept which is the url through which we access the functionality.

Messages exchanged by the components of the system should be self descriptive, that is they should encode the entire result of the resource.

3.3.1 Interface

An important characteristic of REST is that it should expose an uniform interface. The interface has to satisfy the following constraints

- Each resource must have an **id** (usually an URI) and everything that has an id is a valid resource (including a service).
- REST components **communicate by transferring a representation** of a resource. The format of the representation can be chosen dynamically in a set of standard data types (e.g. XML, JSON or HTML). A representation of a resource contains both data and metadata. The representation can be in the same format of the raw data or in another derived format, in both case this information is hidden to the user.
- **Messages** have to be **self descriptive**.
- **Clients move from a state to another** each time they process a new representation, usually linked to other representations through hypermedia links.

The advantage of having a uniform interface is that we can uniform caching. In fact the cache simply has to check if a url and its parameters are already in the cache and returns the corresponding result if present. If all REST systems use urls and parameters as interface, then we only need to develop one cache that can work with every system, independently from the application (the only important thing is that the system uses the url-parameters interface).

The same reasoning can be applied to other types of modules like security modules.

3.4 Peer-to-peer

In previous architectures there's always been a distinct distinction between who offers and who requests something. In a peer-to-peer architecture there are no fixed roles and every component can either act as a client or as a server.

Peer-to-peer has many advantages over the client-server architecture in fact

- The client-server architecture doesn't scale well and usually requires specific types of machines for each role.
- The server is a single point of failure.
- Peer-to-peer takes advantage of the fact that common-purpose computers are getting more and more powerful and have access to broadband connectivity.

Each component of the network is a peer and it offers some type of resource like data, files or processing power that is directly exchanged between the peers. In some cases the network need a central server to coordinate the peers. The main server, however, doesn't handle the resource transfer (it offers secondary services).

3.5 Object oriented

The object-oriented architecture is similar to the peer-to-peer architecture in the sense that there's no specific role for each component. The difference is that in this case each component of the system is an object, that is an entity that encapsulate both a state and some operations (an API) to change the state. Object-oriented systems are stateful because the objects have a state.

Invoking methods on different machines Object-oriented systems work just like a normal program where objects call methods of other objects with the difference that objects are located on different machines. Usually to achieve such result we need to use a middle-ware (e.g. RMI in Java) that abstracts the communication part needed to invoke methods on different machines.

Advantages Object-oriented architecture inherit all the advantages of peer-to-peer architecture and add the advantages of object oriented programming like

- **Encapsulation**.
- **Information hiding**.
- **Inheritance**.

3.6 Data-centred

The data-centred architecture is different from the other architectures because in this case there is no direct interaction between components. In this case components interact through a logical area (a data space, which is actually implemented as component but it's just a logical area). Components can send data, retrieve a copy of the data or remove data from the data space. The data space is like a shared memory between multiple distributed components.

Synchronisation The access to the data is usually synchronised so that at any moment only one component can interact with the data space. This behaviour also allows components to synchronise themselves. In fact if a component has to wait if it requests a data which is not available or is been accessed by others.

Middle-ware The actual implementation of this architecture uses the client-server paradigm but, from the point of view of the programmer the actual implementation is hidden by a middle-ware so he/she can see the server as a data space.

Disadvantages The data-centred model has some disadvantages

- The shared data space has to be implemented.
- The shared data space can become the bottleneck and the single point failure of the system.
- The model is only proactive, in fact components have to search for the data and never get informed when new data arrives (unless one searches a data and gets blocked, in this case the system shared space notifies the component when the data it was look for is available). To solve this problem some systems have introduced the possibility to add requests of data to the shared space. When such data is available the data space notifies the component (but it doesn't send the data).

3.7 Event-based

In event-based systems a client can

- **Subscribe** for an event. When a component subscribes to an event it describes the type of event it would like to receive.
- **Publish** an event. When an event gets published it gets forwarded to all components that have subscribed to such event.

Every component can be both a subscriber and a publisher.

Comparison with data-centred architectures The main similarities between data and event based architectures are that in both cases

- The communication is completely **message based**.
- The communication is **implicit** and **anonymous**, in fact the publisher has no idea of how many subscribers are there. The same thing happens in data-centred systems, in fact when a component adds a resource it doesn't know who will access that resource.

There are though some differences between the two architectures, in particular

- The event-based architecture is **asynchronous**.
- The event-based architecture is **multicast**, in fact the publisher directly sends the message with the event to each subscriber without passing through a shared space.
- In data-based systems the data is persistent (the only way to remove it is to delete it). Suppose a component wants a resource. In data-based systems such component can get the resource when it gets added to the shared space or after some time. In event-based systems instead the component can only get an event only if it has subscribed to it before its publication, thus the data is **transient**. In other words, after a component publishes an event such event is deleted.

Central server This architecture can be actually implemented with a central server that receives all the subscriptions and publications and forwards the messages. This implementation is hidden to the programmers using a middleware and has the same problem of the data-centred approach (single-point of failure, bottleneck).

3.8 Mobile code

Mobile code is a style that can be added to other architectures that allows to relocate the software components of a distributed application at run-time.

In other words with mobile code software and hardware aren't a single, glued component anymore. For instance instead of asking to a server a certain resource we could move an entire process to the server and execute the code (i.e. move the process of the client to the server) to compute such resource directly on the server and than come back to the server.

Paradigms Mobile code can be divided in three main paradigms. To show these paradigms we will use the client-server architecture but the same mechanisms works for any couple of components (with no specified role).

- In **remote evaluation** the client has the code but doesn't have the data and the computational power to execute it so the client sends the code to the server that executes the code with its data and returns to the client the result of the computation.
- In **code on demand** the client has the data execute some code (and the computational power) but doesn't have the code. The client has to ask the code to the server and then it can execute the received code locally with its own local data.
- In **mobile agent** the process running on the client is moved to the server. In this case not only the code and the data is moved but also the stack and the heap. The execution on the client is interrupted and the resumed from the same point on the server.

Advantages and disadvantages Mobile code has many advantages (e.g. flexibility) but also many cons. The main disadvantage is security, in fact when a component sends code it could also be malicious code that can be freely executed on another machine.

Mobility Systems can offer

- Strong mobility if they can transfer both code and the state of the computation (i.e. the process). Systems with strong mobility can implement all the paradigms.
- Weak mobility if they can only transfer code. Systems with weak mobility can only implement on-demand code and remote evaluation.

3.9 CREST

CREST extends REST using mobile code. In this architecture instead of exchanging representations, the components exchange computations. In other words a component takes a function and its data (i.e. parameters, local variables) and passes it to another component so that it can resume the execution from where it had been interrupted.

Part III

Naming

Chapter 4

Naming

Names allow us to reference an entity. Naming services associate a name to an entity (i.e. given a name obtain the reference to the relative entity).

Entities can be accessed via one ore more access points. An access point is an entity characterised by an address (i.e. a special name). The address of an entity could change (e.g. IP addresses can be dynamically assigned) so we need to use location independent names (i.e. names that do not rely on the address used to access the entity). For instance google.com (the name) doesn't change even if the IP address (the address) of the related server changes.

Names can be divided in

- **Global** names if they refer to the same entity no matter where are they used. An example of global names is the absolute path of a file in a file-system.
- **Local** names if the entity the refer to depends on where they are used. For instance a relative path in a file-system is a local name.

Another important distinction is between

- **Human friendly** names that can be easily interpreted and read by humans.
- **Machine friendly** names that can be easily used and interpreted by machines.

Resolving a name Resolving a name means to obtain the address of a valid access point of an entity given its name. For instance a DNS is system whose main task is to associate a domain name to an IP address.

Identifiers When taking into account mobility (i.e. entities that can move and change address) it becomes really hard to associate a name with its address, in fact a name could reference an entity that has more addresses. We can use identifiers to face this problems, because

- Identifiers never change during the lifetime of an entity.
- Each entity has exactly one identifier.
- Every entity has a different identifier.

With identifiers the problem of resolving a name is divided in two parts

1. Associate a name to the identifier of an entity. This task is executed by the **naming service**. Notice that many names can refer to the same identifier.
2. Associate the identifier with its address (i.e. locate the entity). This task is executed by the **location service**. Notice that an entity may have many addresses.

These two phases are shown in Figure 4.1.

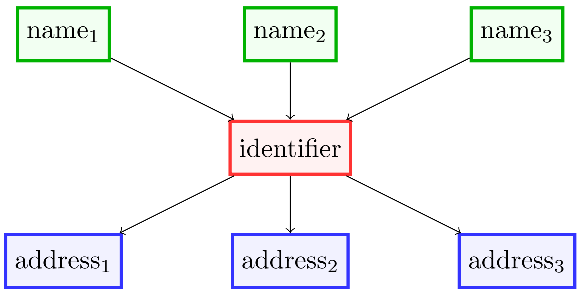


Figure 4.1: Resolving a name with identifiers.

4.1 Naming services

Names can be classified in three categories depending on the schema they are generated with.

- **Flat naming**.
- **Structured naming**.
- **Attribute based naming**.

4.1.1 Flat naming

In flat naming, names are simple (hence flat) strings. In other words such names have no fixed structure. For instance postal addresses and names are **not** flat names (a name is always made of a name and a surname).

Name resolution

A flat name can be resolved using

- **Broadcast or multi-cast**. The naming service sends the name to every entity and only the entity that recognises its name replies with its address. This process is very simple but doesn't scale well. For instance this type of name resolution is used in the ARP protocol.
- **Forward pointers**. This name resolution technique fits well when an entity changes location. Assume an entity E is on the machine 1 with address A_1. When a service S wants to access E it can use A_1. If E changes its location (let's say to 2) and its address becomes A_2, the service has no way to know its new address. What happens is that the machine 1 keeps a reference (a forwarding pointer) to the new location of E. When S wants to access E, it uses A_1, which

isn't the address of E anymore but the machine 1, knowing the new location of E, redirects the request of S using A_2. When the machine 2 receives the requests it acknowledges S and sends its new address so that now S can reference E using A_2. This solution allows to handle mobility but increases latency.

- **Distributed Hash Tables**. This name resolution technique uses hash tables where names (keys) are associated to addresses (values). The hash table is divided between many devices connected one to each other to improve performance.
- **An hierarchical approach**. This technique builds a tree in which the leaves contain the addresses and the intermediate nodes contain pointers to other intermediate nodes or to leaves. Every node N knows which addresses are present in the sub-tree rooted in N (i.e. every node is responsible for a sub-domain). A service can query any node (intermediate or leaf). Upon receiving a query, a node forwards the request upwards (unless the name is in its sub-tree) until a node recognises that the name is part of its sub-domain. At this point we can follow the pointers to the leaf where the address associated to the name is. This system works well when the name is close to the leaf. To improve performance the nodes can cache some information. In any case, this solution is still better than having a single device that handles all the requests.

4.1.2 Structured naming

A structured name is organised in a fixed structure (i.e. in a name space). The name space can be described using a graph made of

- **Leaf nodes**. A leaf node represents a named entity. Each resource is referred with a path name.
- **Directory nodes**. A directory node has some pointers to other nodes. Each directory node knows which pointer has to be followed given a certain path name.

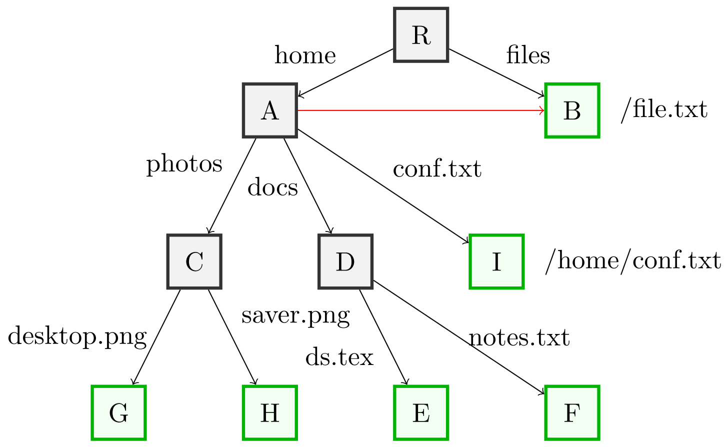


Figure 4.2: A name space (directory nodes in black, leaves in green and hard links in red).

A classical example of a structured naming system is a file system. A file system can be represented as a graph that has the root folder as root node. Every directory is a directory node and every file is a leaf node. The name space isn't necessarily a tree (even if its structure is mainly the structure of a tree), in fact we can also have

- **Hard links** that connect a node to another node that is not its son.
- **Symbolic links** in which the identifier or the address of an entity is replaced with the absolute path name.

Name resolution

Name resolution can be done using a tree (different from the graph used to represent name space). Every level of the tree is responsible for a part of the name and the leafs of the tree contain the address of the name.

Every node of the graph

- Can be putted on a different machine.
- Has a symbolic reference to another node that is responsible for another part of the name.

For instance in DNS every node is responsible for a different domain of the url (to get the address of deib.polimi.it first we ask to the node that knows the it domain. It gives us the link to the node responsible for the polimi domain and so on until we get the address of the name).

The resolution of the name can be done

- **Iteratively**. In this case we ask to the root node to resolve a name. If the root node doesn't know the address of the name it gives us the link to a node that may know the address. We have to repeat the same operation with the node just obtained until we get the address.
- **Recursively**. In this case we ask the root node to resolve a name. If the root node doesn't know the address related to the name he forwards the request to a node that might resolve the name. This operation is executed recursively until the name is resolved. Finally the root node answers with the resolved address.

4.1.3 Attribute based naming

An attribute based name is a structured name that can be seen as a record with fields (attributes). Attribute based naming systems are usually called directory services.

Attribute	Value
Nation	Italy
Region	Lombardy
City	Milan
Name	Politecnico di Milano

Table 4.1: A record of an attribute based name

Operations

This type of names allow two different types of operations

- **Lookup**. This operation, given a full name, returns the corresponding address. With the previous name types we could only do lookups.
- **Search**. This operation, given some attributes, returns a list of addresses that satisfy the given attributes (like in SQL). To perform this operation we need to have a database.

Hierarchical approach

If we interpret the labels of a name in a hierarchical way we could build a graph and distribute the addresses in such graph. In this way we increase efficiency. This solution only works if the attributes are organised in an intelligent way. For instance an intelligent way to organise a name space with the scheme defined in 4.1 would be to put the Nation attribute at the highest level of the hierarchy (Nation→Region→City→Name).

LDAP

Distributed attribute based naming services, when implemented, can be combined with structured naming. An example is the Lightweight Directory Access Protocol (LDAP). An LDAP directory is made of records (also called directory entries) whose form the directory's Directory Information Base (DIB). Each record is made of a collection of $< attribute, value >$ pairs in which

- Each attribute has a type.
- A value can be a single value or multiple values (i.e. an array).

An example of a record is shown in Table 4.1. Each record has a unique name, generated as the sequence of values of that records. For instance, the unique name of the record in Table 4.1 is

```
/Nation=Italy/Region=Lombardy/City=Milan/Name=PolitecnicoDiMilano
```

The directories can be used to create a tree like in structured name-spaces where each directory node is responsible for the sub-tree in which all nodes contain a specific value for a certain attribute. For instance a directory node can be the root of a sub-tree in which all records have `Region=Lombardy`.

4.2 Removing referenced entities

In many distributed application it is important to remove the references to the entities that aren't used anymore (like Java's garbage collector that deletes the objects with no references). There exist many ways to achieve such goal, some of the most used are

- **Black and white graphs**
- **Reference counting**.
- **Weight counting**.
- **Reference listing**.
- **Mark and sweep**.

4.2.1 Black and white graphs

This method is based on a graph in which every node is an entity and is marked as unchecked (it is coloured in white). The algorithm is the following

1. We start from a set of entities that are considered alive.
2. Starting from the alive entities we follow the arches and mark every node we encounter as checked (black).
3. After checking all possible arches (if we reach a black node we stop because we have already been there), all the nodes that are still white have to be eliminated because there is no way to reach such entities starting from an active entity (otherwise they would have been coloured black).

This algorithm is very efficient on a single machine because

- The system can stop the execution of the program to execute it.
- The algorithm has access to the whole memory.

but doesn't perform well in distributed applications.

4.2.2 Reference counting

In reference counting every objects O counts how many processes or objects have a reference to O.

When a process P_1 passes a reference to the object O to a process P_2

1. P_1 notifies O that a process asked a reference.
2. O increases the counter of references.
3. O sends an acknowledgement to P_2 to confirm that the counter has been incremented.

On the other hand when a reference is removed from O, then O has to be notified so that it can decrease its counter. If the counter goes to 0 the object O can garbage itself.

Requirements

This algorithm behaves correctly only if the following properties are verified

- **Reliability**, in fact the processes have to inform the object every time a reference is requested or removed. In a distributed system processes exchange such information via messages, therefore we have to assure that such messages reach the destination exactly one time (it is easy to ensure that a message reaches the destination at most one time or at least one time but it is hard to ensure that the message is received exactly once).
- Handling of **race conditions**.

Racing conditions Let's try to understand why racing conditions may occur. Let us consider

- A process P_1 that has a reference to object O.
- A process P_2.

In this situation the value of O's counter is 1 because only one process (i.e. P_1) has a reference to O. Say that P_1 doesn't tell O that it has send O's reference to P_2. Consider now that P_1 wants to copy and send its reference to P_2 and after that it wants to delete its reference. Here is a possible sequences of events that leads to an inconsistent state because of racing conditions

1. P_1 sends O's reference to P_2.
2. P_2 receives O's reference from P_1.
3. P_2 sends a message to O to notify that a reference has been added.
4. P_1 deletes its reference to O.
5. P_1 sends a message to O to notify that a reference has been cancelled.
6. O receives P_1's message and decrements its counter. Since the counter is 0, O is deleted.
7. At this point P_2's message should be received by O, but it has already been removed.

At the end of this sequence we can notice that P_2 still has a reference to O, but O has been deleted. This happens because P_1's delete message and P_2's add message are concurrent and the former is executed before the latter.

4.2.3 Weight counting

In distributed systems the reliability property cannot be ensured. Weight counting comes in hand to reference names in unreliable systems. In this algorithm every object has a skeleton made of two counters

- A **total counter**.
- A **partial counter**.

and every process has a proxy with just a partial counter that points to an object (the proxy stores the reference to an object). When a process P_1 asks for a reference to an object O, the partial counter of the object is divided in half and the new value of the counter is copied to the partial counter of P_1. After this operation both P_1 and O have the same value in the partial counter.

When a process P_2 asks P_1 the reference to O, P_1 doesn't have to notify O but simply has to halve its partial counter and copy the new value to the partial counter of P_2. This way the sum of the partial counter of every process and of the object is equal to the total counter of the object. The proxy inside P_2 points directly to O.

When P_2 removes its reference to O, the total counter of O's skeleton is reduced by the partial counter of P_2's skeleton. O is deleted when its total and partial counter are equal.

The only problem with this architecture is that we could reach a state in which the value of the partial counter cannot be halved. To solve this problem we can create an internal skeleton. For instance if the partial value of process P_1 is 1 and P_2 asks for a reference, P_1 could create an internal skeleton with new values for the total and partial counter. The new internal skeleton behaves like the skeleton of an object. P_1 passes a reference to its skeleton and when P_2 has to reference O, it has to pass through P_1 (it doesn't have a direct reference to O anymore).

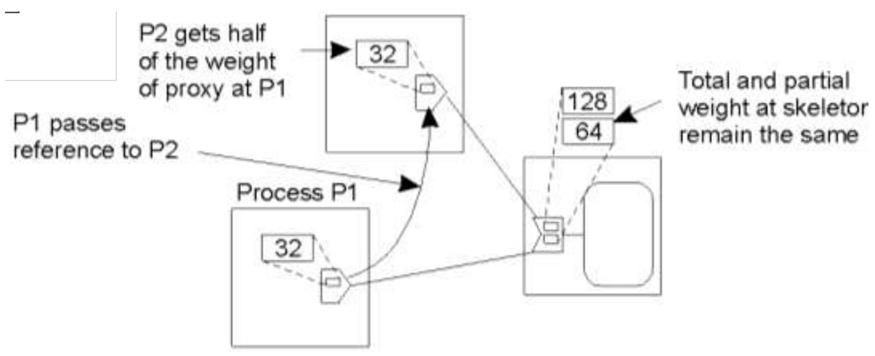

P2 gets half of the weight of proxy at P1
32
P1 passes reference to P2
Process P1
32
128
64
Total and partial weight at skeletor remain the same

Figure 4.3: An example of weighted references.

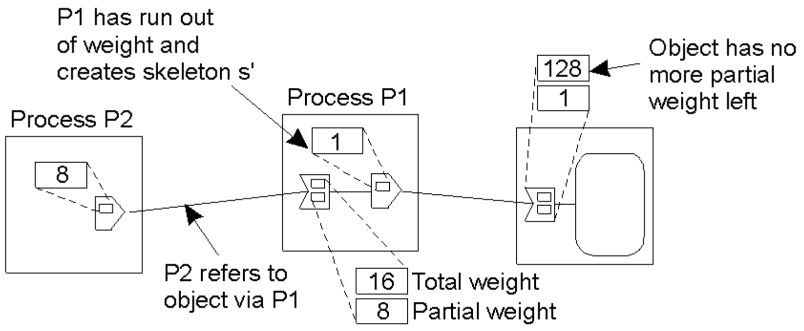

P1 has run out of weight and creates skeleton s'
Process P2
8
Process P1
1
128
1
Object has no more partial weight left
P2 refers to object via P1
16 Total weight
8 Partial weight

Figure 4.4: An example of weighted references with an internal skeleton on a process.

4.2.4 Reference listing

In reference listing every object saves the identities of every process that have a reference to such object. The list of identifiers is a set.

This methods mitigates the problem of non reliable communications, in fact if we keep a list of all processes that have a reference we can discard all the messages that asks for a reference and come from a process that is already in the set of processes with a reference (i.e. we cannot get more than one reference request message).

Reference listing still suffers from race condition problems when a process $P1$ passes a reference O to another process $P2$.

4.2.5 Mark and sweep

The mark and sweep algorithm is similar to the black and white algorithm. The only difference is that when the algorithm visits a node, it colours it grey. The node is painted black only when the algorithm has visited all the adjacent nodes (in practice only when it has forwarded the request to all its adjacent nodes).

Distributed mark and sweep

Mark and sweep can also be run in a distributed system. In this case a local process is connected via a proxy to the skeleton on another node's process. The distributed algorithm works this way

1. A process P and its proxies are marked grey when an object O in process P is reachable from a root in process P.
2. When a proxy q is marked grey, a message is sent to the associated skeleton.
3. When the skeleton acknowledges the proxy, the proxy is turned black. When all proxies are black, the process is coloured black.
4. Once all objects in a node are black or white, the garbage collector can eliminate all white nodes.

Part IV

Communication

Chapter 5

Fundamentals

5.1 Protocols

Protocols describe the way two components communicate on a network.

Structure Protocols have a layered structure in which each layer focuses on a specific problem communicates with the layers above and below. Each layer offer some service to the layer above (i.e. each service uses the services offered by the level below).

The OSI model is an example of layered protocols.

Implementation Since many layered protocols use packets to communicate, the layered protocols are implemented using a mechanism called encapsulation in which each protocol takes the message of the lower level protocol, adds its header and gives the message to the layer above.

Middleware In the OSI model the middleware layer usually sits above the transport layer and below the application layer. Middleware can add functionalities to the transport protocol like

- Naming.
- Security.
- Scaling mechanisms like replication and caching.

The communication offered by middleware can be

- **Transient** if the communication only happens if the components that are communication are active at the time of communication. UDP is an example of transient communication.
- **Persistent** if the communication can happen even if one of the components is not active. For instance data-centred systems offer persistent communication, in fact after a component has added a file to the shared space, another component can retrieve it at any time after the upload (even if during the upload it wasn't active). Persistence may happen on the destination machine or on the middleware.

Another distinction is between

- **Synchronous communication** in which there is a synchronisation point between the two end-points or between the sender and the communication layer. Synchronisation can happen in various forms. For instance the client could stop its execution and wait for an acknowledgement that could be sent at different levels (e.g. the local middleware, the server's middleware, the server's process).
- **Asynchronous communication** in which there is no synchronisation. For instance UDP implements an asynchronous communication because it simply sends a frame without caring if and when it arrived at destination. In general communication is asynchronous when the sender continues its execution immediately after sending a message.

Transient communication These four categories aren't exclusive, that is we can have transient synchronous communications (like phone calls), transient asynchronous communications (like UDP). In particular we can distinguish four types of transient communication

- **Pure asynchronous communication** in which the message sent from A to B is never acknowledged.
- **Receipt-based synchronous communication** in which the message sent by A is acknowledged by B when it arrives at the destination machine (but not at the process). In this case B could also handle the request some time after it has been received. A stops the execution and resumes it only when the message is acknowledged.
- **Delivery-based synchronous communication** in which the message sent by A is acknowledged by B when it is received by the destination process.
- **Response-based synchronous communication** in which the message sent by A is acknowledged by B only after being processed and executed by B.

Chapter 6

Remote procedure call

Local procedure call In local procedure call when we invoke a function the stack is populated with the parameters of the function, the return address and the local variables of the function. The parameters can be passed

- By **value**.
- By **reference**.
- By **copy and restore**.

Remote procedure call Remote procedure call uses the same semantics of local procedure call but the caller and the callee aren't on the same machine. The function invoked by the caller is executed on another process (on the callee machine).

Middleware The remote procedure call mechanism is hidden to the programmer using a middleware. When a programs calls a remote procedure

1. The middleware on the client generates a local procedure that has the same interface of the remote procedure. The implementation of the local procedure is different from the actual remote implementation.
2. The client procedure serialises the parameters (transforms them in a packet) and sends the packet using a lower level protocol.
3. The server's middleware has another procedure that reads the packet and extracts the parameters.
4. The server's middleware invokes the actual procedure on the application layer.
5. The result of the computation is passed to the server's middleware that transforms it in a package and sends it over the network.
6. The client middleware reads the packet and extracts the result that is returned to the caller on the application layer.

When the client invokes a method, it is actually calling the middleware method, in fact the actual procedure is executed on the server. The client only receives the result of the computation that happens on the server.

Automatic middleware generation An important part of remote procedure call is that the middleware should be generated automatically. To automatically generate both the client and the server layers of the middleware we need

- The **number** and **type** of parameters.
- The **language** in which the code on the two machines is written (the language can be different on the two components).

6.1 Serialisation

Serialisation is an important aspect of remote procedure call and can be achieved combining

- **Serialisation** to transform (i.e. flatter) the data in a byte stream.
- **Marshalling** to adapt the basic types of a language to the basic types of another language.

Middlewares that operate on different languages use a unified, platform-independent language called Interface Definition Language (IDL) to describe the signature of the procedures. IDL separates the interface from the implementation (it is not an programming language, it is an interface definition language). IDL can also map itself in target programming languages (e.g. C, Python, Java).

6.2 Passing parameters by reference

Usually middleware do not offer the possibility to pass a parameter by reference but in some cases it is possible to pass a parameter by **copy and restore**. In this solution parameters are passed by copy and when the remote procedure ends its execution, if some parameter passed by copy changed, it is passed to the client that updates its local copy of the parameter. The final result is similar to passing by reference but not he same (copy and restore breaks aliasing).

6.3 Binding client and server

A problem with remote procedure call is finding which server and which process offers a procedure. The easy solution is to hard-code such information in the code of the middleware. This solution is very inconvenient and can cause a lot of problems.

Binding service To solve the binding problem a middleware could offer an additional service that handles the binding between client and server. In this case the socket isn't explicitly provided but can be found dynamically. Such server should find out

- Where is the server process.
- How to establish communication with the server.

Portmap In some implementations the second problem is solved using a daemon process called `portmap` that binds calls and server (i.e. ports). This service is only offered to local clients (i.e. the callers, the machines that want to execute a remote procedure invocation). In this solution however the client code should explicitly define the procedure-address bindings.

DCE Portmap In other implementations a directory server is added. When the server's code starts, it registers the remote procedure to the directory server (i.e. the server adds a procedure-socket couple to the directory server). The client can ask the directory machine the socket for a certain procedure.

6.4 RPC optimisation

6.4.1 Dynamic activation

Until now the server was waiting for a packet to activate a procedure. This mechanism isn't efficient and wastes a lot of resources. This problem in UNIX is solved using the `inetd` daemon that listens for connection on a set of ports and when receives a packet it invokes the server process (this way the job is done by a single daemon and not by the server process). The `inetd` daemon finds the map between a service and a process in a configuration file in the folder `/etc/services`.

6.4.2 Lightweight RPC

Remote procedure invocation could be also used to put in communication two processes on the same machine, in fact processes can't share memory (only threads can). This operation is very useful but has a drawback, in fact the packet containing the parameters should go back and forth on the network stack (which take some time) without any reason (the two processes are on the same machine).

Operating system facilities The operating system can share memory between processes because there's a single memory and the OS can access it all. Therefore to solve the network stack problem we can rewrite the middleware using the operating system facilities to share memory instead of the network stack.

Cut and paste When we cut some data from a process and paste it to another process we are actually using a lightweight remote process call.

6.4.3 Asynchronous RPC

When calling procedures that returns `void` the client can resume execution after the server acknowledges the start of the execution. Asynchronous RPC can also be implemented with non `void` procedures. In this case the caller resumes execution after the server acknowledges the start of execution and when the result is returned the caller is interrupted and can handle the return value.

Chapter 7

Remote method invocation

Remote method invocation is the equivalent of remote procedure call for an object-oriented programming language. The difference is that in OOP the caller has to invoke a method on a remote object.

In RMI a client object is an object that has the same methods of the remote object but those methods serialise the parameters and send them to the server instead of executing the remote code. The procedure is the same as in RPC but the syntax is different.

The object the client interacts with is a proxy object that has the same interface of the object on the server. On the server the method is invoked by the skeleton.

The reference to the procedure is replaced with the reference to an object on which it's possible to call methods remotely.

Passing parameters by reference In RPC it's possible to exploit copy and restore to pass parameters by reference but it much easier to pass by copy. In object-oriented programming it's more difficult to pass by copy (especially for multi-language implementation of RMI) because an object isn't just a set of attributes but also includes methods and we can't move code between different programming languages. On the other hand in RMI it's easier to pass by reference in fact when an object $O1$ wants to call a method on an object $O2$ on another machine it creates a proxy bound to the socket of the remote machine that can access and execute the code of $O2$. The proxy has a reference to $O2$ like in a object-oriented programming. So the two components have proxies (i.e. references) to remote objects.

RMI perfectly recreates passing by reference but it's very complex.

An example in Java Consider a list `AraryList<Person> people` of `Person` on a local machine and that we want to add to such list a `Person p` that is on a remote machine (`people.add(p)`). To execute such operation

1. A proxy Person object is created locally. Such object is a proxy and therefore has a reference of the socket of the remote server where we can find the actual object.
2. The proxy Person gets added to the local list.

When we want to call a method of the person we have just added we can use the syntax

```
people.get(0).wave();
```

What really happens is that

1. The person is actually a proxy object so it has a network reference (and not a normal object reference) and the code of the method `wave()` isn't the one of the real remote object.
2. The middleware executes the local `wave` code that serialises all the parameters and the local variables and sends them to the remote server for execution.

This way we can execute code even if the two machines use different languages, in fact the local proxy object simply says to the remote object to execute a method with a certain name and parameters but doesn't care about the code.

In other words when we pass objects through the network we do not pass the actual objects, instead we pass proxy objects that have a network reference to the actual object.

Examples Two famous implementation of RMI are

- **Java RMI**. Java RMI is completely based on Java (thus it's platform dependent and works only on Java). This implementation supports passing parameters by reference or by value even in case of complex objects. In the latter case the class has to implement the `Serializable` interface. Java RMI allows code on-demand, too.
- **OMG CORBA**.OMG CORBA is multilanguage and multiplatform. This implementation support passing parameters by reference or by value. Since CORBA is multilanguage, if objects are passed by value it is up to the programmer to guarantee the same semantics for methods on the sender and receiver sides (sender and receiver might use different languages or platforms).

Chapter 8

Message oriented communication

Remote method invocation and remote procedure call are synchronous and support only point-to-point interaction. On the other hand, message oriented communication

- Usually is fully asynchronous.
- Supports multi point interaction.
- Is based on the concept of one-way message. In RMI and RPC messages aren't one way, in fact when a component calls a method, the other answers with the return value of such method (the caller sends and receives data).
- Supports persistent communication.

8.1 Reference model

In message-oriented communication the middleware is implemented with brokers that

- Receive the messages sent by the sender.
- Operate on the message, changing it if necessary.
- Send the message to the broker of the receiver of the message.
- Deliver the message to the final application.

Some examples of this type of model are sockets used for TCP and UDP and multicast sockets.

8.2 Message oriented communication models

8.2.1 MPI

MPI rises the level of abstraction offered by the sockets that are low level artefacts.

In MPI communication takes place within a group of processes because it was born to handle high performance applications that involve a large number of machines (e.g. a data centre). Each process in a group has a local identifier. A sender or a destination is therefore represented by a the pair

```
groupID, processID
```

A message can be sent to a single destination (i.e. to a single entity represented by a group-process pair) or to all the processes in a group (broadcast communication).

MPI primitives The high level primitives that MPI offers are

- MPI_bsend allows a process to append a message to a local send buffer.
- MPI_send allows a process to send a message and wait until it has been copied to some buffer in the middleware. The buffer could be local or remote. This type of send delegates synchronisation to the middleware.
- MPI_ssend allows a process to send a message and wait until the message has been delivered to the destination machine (but not to the destination process).
- MPI_sendrecv allows a process to send a message and wait for the reply.
- MPI_isend allows a process to pass a reference to a message (actually to a buffer) and continue execution. This type of send is completely asynchronous.
- MPI_issend allows a process to pass a reference to a message (actually to a buffer) and wait until the message has been delivered to the destination machine (but not to the destination process).
- MPI_recv allows a process to retrieve a message. If there is no message the process is blocked and waits for an incoming message.
- MPI_irecv allows a process to retrieve a message. If there is no message the process continues execution.

MPI also offers primitives that allow broadcast communication

- MPI_Bcast allows to send a message to every process in a group.
- MPI_Reduce allows to apply an operator to the data sent to a process. For instance if all the processes send a message with a number and the reduce operation is a sum, the receiver gets the result of the sum of all the messages sent by the others.
- MPI_Scatter allows to divide a list of messages and send a section of the list to each process in the group. For instance in a group with 5 processes if a process wants to send a list of 10 messages, every process receives 2 messages.
- MPI_Gather allows to collect the messages sent from other processes of the group

8.2.2 Message queuing

Message queuing is a

- **point-to-point**.
- **persistent**.
- **asynchronous**.

form of communication. In message queuing message are sent to a queue (and not directly to the destination). A process can send or get a message from a queue implemented in the middleware. The queues are part of the middleware and not of the application. Besides public queues, each component has an input queue and an output queue. Queues are persistent in fact if a process adds a message to a queue, another process could get it even after some time (in general after being added).

Coupling In message queuing there's no coupling between sender and receiver, in fact the sender only has to add a message to a queue and it doesn't send the message directly to the receiver. In this sense sender and receiver aren't coupled. In other words the communication is mediated by a queue.

Primitives Each process can

- **Attach** to a queue. This operation attaches the local queues of the process to the public queues.
- **Put** a message to a queue to append a message to the queue.
- **Get** a message from a queue to obtain an remove the first message of the queue.
- **Poll** the queue to get the first message. If there is no message the process continues the execution.
- **Notify** a queue to add a callback so that when a message is added to the queue the process gets a notified.

Client-server architecture Message queuing can be complex because it uses messages but it has huge advantages. Consider a client-server architecture that uses message queuing. In this example clients put requests to the queue and the server get the requests. This architecture can easily be scaled in fact we can simply add a new server that fetches the requests from the queue without modifying nothing else. This can be done because the components of the system aren't coupled.

Implementation of the queues The public queues used in message queuing can be putted on normal components or on specific components that only handle queues. Queues can also be enriched by adding some code that can transform the messages on the queue.

8.2.3 Publish-subscribe

Publish-subscribe is a way to implement the event-based architecture. Each component can do two operations

- **Subscribe** to a message. When a component subscribes to a message it describes the messages it wants to receive.
- **Publish** a message.

Communication The communication in the publish-subscribe model is

- **Multi-point**, in fact a single message can be sent to many components (i.e. different components can subscribe for the same message).
- **Transient and asynchronous**, in fact only the subscribers that have subscribed for a certain message before it is published receive the notification. In other words the middleware doesn't stores the message until consumed.
- **Indirect** because the publisher doesn't know who will receive the message. The publisher doesn't send the message to a specific person.
- **Receiver based** because it's not the sender that chooses the destination, but it's the destination that subscribes for an message.
- **Without coupling** because the publisher doesn't sends the message directly to the receivers.
- **Scalable** because a different number of components can subscribe to a message.

Subscription language Components have to use a common language to describe the messages they want to subscribe for. The subscription language can be

- **Subject-based** (or topic based). In this case the subscriber describes the subject of the message. The subject of the message is a string (A message is made of a subject, or topic, and a content).
- **Content-based**. In a content-based subscription language the subscriber can explicitly describe the content of the message. In a message the content is usually made of records, so the subscriber can describe the pattern of the record it wants to receive.

Content-based languages are more expressive but they are also more complex to analyse.

Event dispathcer

The messages are usually handled by the event dispatcher that

- Receives and records the subscriptions.
- Matches the messages published with the subscriptions.
- Sends the messages matched to the subscribers.

The event dispatcher is ideally a single component, this means that it can be the bottleneck of the system. To solve this problem many systems have distributed dispatchers (i.e. the dispatcher is divided in multiple processes on different machines). Distributed dispatchers can be divided in

- Acyclic dispatchers if the dispatchers are connected in a way that doesn't generate cycles.
- Cyclic dispatchers if the dispatchers are connected in a way that generates cycles.

Dispatchers have to exchange messages so that they can reach the correct subscriber. The algorithms used to forward the messages are different in the two cases.

Message forwarding In acyclic dispatchers it's easier to handle the messages. The first way to handle the messages is **message forwarding**

1. When a component subscribes to a dispatcher, such dispatcher doesn't forwards the subscription to the dispatchers nearby.
2. When a message is published the network of dispatchers have to propagate it to every node to allow the message to reach its destinations.This happens because each dispatcher doesn't know if its neighbour have a subscription for such message.

In message forwarding every dispatcher saves only a limited number of subscriptions (i.e. the local ones, the ones it receives directly), therefore it's easier to look for a match when a message arrives. On the other hand the cost of forwarding is high because every node always have to forward a message. Since the network is acyclic there's no risk that a message travels forever.

Subscription forwarding Subscription forwarding is the the second type of algorithm for acyclic dispatchers. In this case the subscriptions are always forwarded. In this case the message match is expensive because each node contains the subscriptions of many nodes but the cost of forwarding is low because the message is forwarded directly to the subscriber (each node knows to which node to send the message).

Hierarchical forwarding Hierarchical forwarding is an algorithm to route messages in acyclic dispatchers. In this case the network is seen as a tree (this can happen because the network is not cyclic). To have a tree the network has to have a root.

The algorithm works this way

1. When a component subscribes to a messages the subscription is forwarded only in the direction of the root (i.e. up the tree).
2. When a component publishes a message it is propagated only in the direction of the root and goes to the leaves only if a match is found in one node.

Distributed hash tables To handle message routing in cyclic networks we can use distributed hash tables. An hash table is a data structure made of key-value pairs that provides two operations

- An operation to add a key-value pair.
- An operation to get the value associated to a key.

A distributed has table is a set of processes that together offers the operations to add a pair and to get a value. Each machine keeps a portion of the hash table and when an operation is executed, either the machine knows the result or it asks other machines.

The distributed hash table uses

- The topics as keys.
- A list of the components that have subscribed to that topic as value.

When a component subscribes to a topic it does an add operation on the DHT (i.e. the component is added to the list that corresponds to the topic). When a component publishes a message it does a get operation using the topic of the message as key and the message is sent to all the components of the list.

This solution can only be used for subject-based (topic-based) languages. For content-based languages every node has a table that associates a source and a predicate to the next node, so that when a message arrives it is confronted with predicate and the source and if they match the message is forwarded to the next hop.

Per source forwarding Per source forwarding (PSF) is a **content-based routing method** for cyclic dispatchers to route an event from its source to a node (i.e. a subscriber). In PSF

- Each source defines a shortest path tree (SPT) to reach every other node in the system.
- Each node has a forwarding table that contains
  - For each source, the next hop in the SPT.
  - For each next hop, the conditions that have to be true to forward an event to that hop.

Improved per source forwarding Improved per source forwarding (iPSF) is based on and uses the same concepts of PSF and adds the concept of indistinguishable sources to reduce the dimension of the forwarding tables and the time needed to build them. Two sources A e B with SPT $T(A)$ and $T(B)$ are indistinguishable from a node n if n has the same children for $T(A)$ and $T(B)$ and reaches the same nodes along those children.

Per receiver forwarding Per receiver forwarding (PRF) is a **content-based routing method** for cyclic dispatchers. In PRF

- The source of a message computes the set of receivers and adds them to the header of the message.
- At each hop the set of recipients is partitioned. This means that for each next hop we remove the receivers that can't be reached via that hop.
- Each node has two different tables
  - The unicast routing table that associate the next hop to each receiver.
  - A forwarding table with the predicate for each node in the network.

Distance Vector Distance Vector (DV) is a protocol for **building path in a cyclic network**. In particular, it builds a minimum latency spanning tree using packets and their responses. The responses are used to update the spanning tree and allow to create a local view of the tree at each node. Namely, each node sees its neighbours in the tree.

Link State Link State (LS) is a protocol for **building path in a cyclic network**. In particular, it allows to build a Shortest Path Tree (SPT) using different metrics. The tree is build using packets that carry information about the known state of the network. To ensure that every node knows any new information, a packet is forwarded as soon as some node acquire new information. Differently from Distance Vector, every node discovers the topology of the whole network. SPTs are calculated locally and independently by every node.

8.2.4 Complex event processing

Complex event processing is an evolution of publish-subscribe, in fact the dispatcher in CEP can also match complex subscriptions.

A complex subscription is defined by a rule language that allows to describe complex events using raw events. For instance a complex event could combine two events that happen at a fixed distance of time (a smoke event followed by a heat event after 30 seconds both from the same area generates a fire event).

In CEP

- Subscribers subscribe to complex events.
- Publishers publish raw event.
- The dispatcher uses the raw events to generate complex events.

This type of communication architecture can be very useful in a network of sensors that publish raw data. Such data can be analysed by the dispatcher and send to us (the subscribers) that are interested in a complex analysis of such data. Currently subscription languages have a syntax similar to SQL.

Chapter 9

Stream-oriented communication

Stream-oriented systems handle data streams (i.e. sequences of data units). In such systems time can or cannot impact on correctness.

Stream-oriented communication can be

- **Asynchronous**
- **Synchronous**
- **Isochronous**. In isochronous communication there is a maximum and a minimum end-to-end delay. For instance video streaming is a type of isochronous communication.

Middleware In stream-oriented systems there's usually

- A **source of data** (e.g. compressed multimedia data).
- A **streaming server** that takes the data from the source and provide such data to the client in segments.
- A **destination client**

9.1 Quality of Service

Non functional requirements in stream-oriented communication are called Quality of Service and are very important in data streams. Common non functional requirements are

- The required **bitrate**.
- The **maximum delay to setup a session**.
- The **maximum end-to-end delay**.
- The **maximum variance in delay**.

Client and server has to agree on the Quality of Service (QoS). Usually the clients asks for a specific QoS and the server can accept it. Once the QoS is defined, client and server have to control it.

9.1.1 Differentiated Services

The IP protocol is a best effort protocol, thus it doesn't ensure to satisfy any QoS. This problem can be fixed using the Differentiated Services field (also Type Of Service) in the header that allows to specify the priority of a packet. The header contains

- 6 bits for the **Differentiated Services Code Point** (DSCP) field.
- 2 bits for the **Explicit Congestion Notification** (ECP) field. This field encodes the Per Hop Behaviour (PHB). The PHB can be
  - Default.
  - Expedited forwarding.
  - Assured forwarding (further divided into 4 classes).

This field is usually ignored by some routers so it can be useless. In other words Internet can help improving QoS so everything has to be implemented at the application layer.

9.1.2 Improving Quality of Service

QoS can be improved at the application layer using

- **Buffering** that allows to save the packets in a buffer and start to show the data to the user only after the buffer has accumulated a decent number of chunks. Buffering allows to overcome the fact that the packets could arrive with different time spans.
- **Forward correction error** that is used in those systems that have to guarantee deliver in time. Forward error correction tries to correct the packet. This is possible because the packet was sent with additional information. For data streams backward error correction (i.e. if there's an error the receiver asks the sender to send the a copy of the message) cannot be used because it worsen QoS. The disadvantage of this method is that the package is bigger so the system needs more bandwidth.
- **Interleaving data** to mitigate the impact of lost packets. In this method the chunks of data are interleaved so that if a packet is lost the chucks lost are at the same distance (so it's more difficult to notice) and not in a sequence. For instance the first packet could contain chunks 1, 4 and 7, the second packet chunks 2, 5 and 8 and the third packet chunks 3, 6 and 9.

  ```
  [1 4 7][2 5 8][3 6 9]
  ```

Synchronisation Typically stream-oriented systems handle multiple streams that have to be synchronised. To synchronise multiple streams we can

- Include **more streams in one packet**. In this case the synchronisation happens on the sender side. The main disadvantage of this method is that when a packet gets lost, multiple streams get lost.
- Use **different packets for each stream** and add a timing information so that the streams can be synchronised on the receiver side. IN this case we can choose different QoS for each stream and we can decide how to synchronise the streams when a packet gets lost.

Part V

Synchronisation

Chapter 10

Introduction

Many distributed systems have to synchronise concurrent activities done by multiple processes and machines. Synchronisation can also be done in a non distributed system using the machine clock and memory. On a distributed system things get harder because we don't have neither a global clock nor a shared memory. Moreover, we have partial failures, in fact in a single-machine system if a component of the program fails, the whole program crashes, but in distributed systems a part could fail while the other keep working. In fact, in distributed systems we want the system to tolerate failures.

Algorithms To synchronise distributed systems we have to study algorithms for

- Synchronise physical clocks.
- Simulate a time using local logical clocks.
- Mutual exclusion.
- Leader election.
- Collecting global state.
- Termination detection.
- Distributed transactions.
- Detecting distributed deadlocks.

Chapter 11

Time and clocks

Time can be used to synchronise activities, in fact timestamps can be used to check if a certain action has to be executed (or has been executed) before another one. For instance when we compile a C file using `make`, it compares the timestamp of the object file and the source file to decide whether to compile or not (the file is compiled only if it has a newer timestamp than the object file because it means that after crating the object file, the source file has been edited).

Timestamps It's harder to work with timestamps in distributed systems, in fact two machines might have not synchronised timestamps, so even if an action α happens before another action β (in terms of absolute time) it might seem that α has happened after β because the clocks that generate the timestamps are not synchronised. Consider, for instance, two processes A and B whose current timestamps are $\tau_0^A = 103$ and $\tau_0^B = 100$, respectively. A does an operation α at the timestamp τ_0^A (subscript indicates the global time) and B does an operation β after one tick (i.e. at timestamp $\tau_1^B = 101$, when the timestamp of A is $\tau_1^A = 104$). The operation done by B has an earlier timestamp than the operation done by A but it happened after the operation done by A because A happened at global time $t = 0$ while B happened at global time $t = 1$. The problem here is that, initially, the clocks weren't synchronised.

11.1 History and time measurement standards

Initially time was measured as a fraction (1/86400) of the mean length of a day (astronomical time or Coordinated Universal Time, UTC). This measure is rather inaccurate so scientists decided to use the number of oscillations of an atom of Caesium 133 to measure one second ($1\text{sec} = 9,192,631,770$ oscillations). This convention is called International Atomic Time (TAI).

Leap seconds UTC isn't as precise as TAI, in fact it depends on the rotational speed of hearth. Scientists have discovered that earth is slowing down so days are lasting longer and therefore seconds measured using the astronomical time are getting longer. To compensate this difference between TAI and UTC, when the difference between the two measures is more than 800 ms (i.e. UTC has accumulated 800 ms of delay) a UTC second is skipped. Such skipped second is called **leap second**.

11.2 Physical clock synchronisation

A computer clock is a timer, that is a device that generates a tick with a specific frequency (like a metronome that tells how many ticks have passed).

Drift rate In principle the frequency is perfect but in reality clocks have little changes in frequency called drift d. This means that the real frequency f_r can actually be a value between the ideal frequency f_i plus or minus the drift).

$$f_i - d \leq f_r \leq f_i + d$$

To evaluate the drift of a timer we can use the drift rate ρ (that is a characteristic of the timer) that measures the difference between two clocks with the same frequencies due to the drift of each clock. For instance the drift rate of an ordinary quartz clock is $10^{-6}s/s$ (s/s means that 10^{-6} seconds are lost every second). This means that if two machines use an ordinary quartz clock, their timestamps could differ 1 second every 12 days (this result has been obtained transforming the seconds at the denominator in days, in particular $1\ s = \frac{1}{86400}\ days$).

Maximum skew The maximum skew δ is the maximum difference between the timestamps of two clocks and depends on the physical characteristics of the timer and the time Δt we wait between two measures.

If two clocks are drifting in opposite directions (i.e. one clock has a smaller real frequency, the other a bigger real frequency), then the maximum skew in an interval Δt is

$$\delta = 2\rho\Delta t$$

From this formula we can determine when two machines have to be synchronised. In particular a synchronisation is needed at least every

$$\Delta t = \frac{\delta}{2\rho}$$

seconds. Thanks to this formula we can set how much skew we want to allow and calculate at which intervals we have to synchronise the system.

11.2.1 GPS synchronisation

GPS needs 3 satellites to get the position of a device. In particular each satellite

1. Sends a packet with a timestamp t_0 to a device.
2. The device answers with another packet with a timestamp t_1.
3. The satellite uses the difference between t_1 and t_0 to get the delta time Δt needed to go from the satellite to the device. Such difference can be used to compute the distance between the satellite and the device (c being the speed of light).

$$d = \Delta t \cdot c$$

Three satellites aren't enough to precisely locate a device, in fact the clock on the device is much less accurate with respect to the clock on the satellite (satellites use atomic clocks that use Atomic Time), so the timestamp of the device isn't precise (it's not in sync with the clock of the satellite). If the timestamp is not accurate than Δt is not accurate and therefore the distance d isn't accurate either.

To precisely locate a device we need a fourth satellite that calculates the skew between the satellite and the device and synchronises the clock of the device with the clock of the satellite. If the clocks weren't synchronised, the device couldn't have been located correctly. GPS can locate a device with a precision of some meters (less than a hundred), this means that the precision of the synchronisation is of nanoseconds (i.e. the delay of the device is of nanoseconds).

If two devices have GPS, than they can synchronise themselves with the atomic clock of a satellite (the atomic clocks of the satellites are synchronised, in fact they all use TAI), so both devices are synchronised.

Finding the device delay Let us now understand more precisely how four satellites are able to compute the delay between the device clock and the TAI. Let us call

- Δ_r the unknown difference between the device's clock and the satellites' clock.
- x_r, y_r, z_r the unknown coordinates of the device.
- T_i the timestamp of the message sent by satellite i.

Let us now consider a device that receives a message sent by a satellite. The device receives the message at global time T_{now} but it replies with its time T_r. In other words the unknown Δ_r can be written as

$$\Delta_r = T_r - T_{now}$$

When the device receives all 4 messages it can compute the distance as

$$\begin{aligned} d &= c\Delta_i \\ &= c(T_r - T_i) \\ &= c(T_{now} - T_i + \Delta_r) \\ &= c(T_{now} - T_i) + c\Delta_r \end{aligned}$$

The first addendum has to be equal to the Euclidean distance (because $c(T_{now} - T_i)$ is the actual distance between the satellite and the device), thus we obtain

$$c(T_r - T_i) = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2} + c\Delta_r$$

Since every satellite sends a message, we obtain 4 equations (one for each i) in 4 unknowns (x_r, y_r, z_r and Δ_r). The system can be solved to obtain the device's coordinates and Δ_r.

Issues GPS is the best solution to synchronise clocks but it also has some disadvantages, in fact

- It isn't always available (for instance underground or in some building)
- In some case it has to be added to the machines, in fact GPS is not a standard equipment on every device.

11.2.2 Cristian's algorithm

Cristian's algorithm is based on a time server, that is a server that synchronises the clocks of the other processes.

This algorithm doesn't synchronises the devices with the global time (i.e. with the TAI) but synchronise the devices between them using the time of the time server. In other words it's not important what the time is, but it's important that every machine on the system agrees on what time it is.

Algorithm The algorithm works in the following way

1. At regular intervals a device sends a synchronisation request to the time server.
2. The time server replies with the correct time.

This algorithm works well when messages travel fast with respect to the required time accurancy.

This first algorithm allows the device to accumulate some error in fact from the time when the server checks the current time to the time the response device receives the packet, some time passes so that the time written in the packet isn't the time at which the packet arrives to the device. For instance consider the following sequence

1. The packet arrives at the server.
2. At 14:00:01 (server time) the server checks the time.
3. At 14:00:02 (server time) the server sends the packet.
4. At 14:00:05 (server time) the device receives the packet.

The device synchronises its clock using the value 14:00:01 that is in the packet but the real time is 14:00:05.

To solve this problem the device computes the time ΔT that passes between when he sends the packet and when he receives the answer, and adds $\frac{\Delta T}{2}$ to the time received.

$$t_{sync} = t_{received} + \frac{\Delta T}{2}$$

This approximation is reasonable only if the the time needed to go from client to server is similar to the time needed to go from server to client. In other words the network (and the time to elaborate the packet) has to be homogeneous. On the other hand the speed of the network isn't relevant (in fact we only need to know the time difference from when the packet is sent to when it's received). If the time difference ΔT is much smaller than the time accuracy requested by the client, the ΔT is discarded.

The new algorithm can be written in the following way

1. The device calculates the time T_0 (local time).
2. The device send a synchronisation request to the time server.
3. The time server replies with the correct time t_{server}.
4. The device calculates the time T_1 (local time).
5. The device synchronises its clock with

$$t_{server} + \frac{T_1 - T_0}{2}$$

11.2.3 Berkley algorithm

The Berkley algorithm is similar to the Cristian's algorithm, in fact both use a time server to synchronise the clocks. In this case the server doesn't sends its time but it collects all the requests coming from the clients. Every request contains a timestamp (which is the local timestamp of the client) and the server averages the timestamp. The average is sent back to every client that synchronises its clock using such average.

For instance if a client sends a request with time t_c =8:00:10 and another client sends a request with time t_γ = 8:00:20, the server calculates as average time t_a = 8:00:15 and sends it to the clients. Sometimes the new time is sent in terms of adjustments with respect to the time sent in the packet (in the example the first receives a +5 seconds while the second client a -5 seconds).

11.2.4 Network Time Protocol

Network Time Protocol (NTP) was designed for synchronising UTP time for large distributed systems, in fact it was designed for devices connected to the Internet (every OS nowadays has a NTP server running). The goal of NTP is to synchronise the clocks of multiple devices against UTC with an error in the order of some tens of milliseconds.

Principle of working NTP is implemented using a set of servers organised in a hierarchical structure (a tree or a forest). The root node (or the root nodes in case of a forest) is a machine that knows global time (e.g. it has an atomic clock, or uses GPS) and at the leaves there are the clients (i.e. our devices, in fact every computer in this case is a server because it runs an NTP server) connected to the Internet.

Every node of the tree asks the time to the node at the upper layer and gives its time to the nodes of the lower level. In other words each layer

- Synchronises with the upper layer.
- Provides synchronisation for the lower layer.

A server can provide synchronisation in different ways depending on the type of the network, in fact

- If the client (i.e. the one that asks for synchronisation) is on the same LAN of the server (i.e. the one that provides synchronisation) then the server **multicasts** its time at certain intervals because the latency can be neglected.
- If client and server are on different networks than **procedure call**, a protocol similar to Cristian's algorithm, is used.
- If the nodes are close to the root **symmetric mode** is used. This synchronisation protocol ensures a more accurate clock synchronisation.

Procedure call and symmetric mode The procedure call algorithm is a variation of the Cristian's algorithm used to synchronise clocks in NTP.

In this case the algorithm uses three packets (instead of two like in Cristian's algorithm), in particular

1. The lower layer sends a packet to the upper layer to asks for the time. The packet contains the local time T_{i-3}.

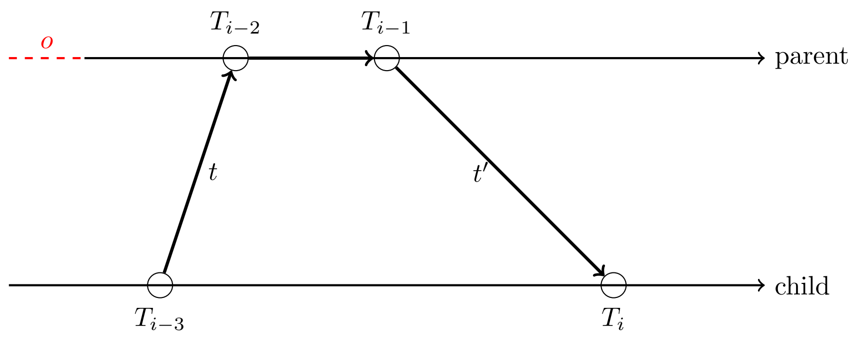


Figure 11.1: The synchronisation between two nodes in NTP.

2. The upper layer replies with a packet containing the time T_{i-3} received, the local time T_{i-2} at which the packet arrived and the local time T_{i-1} at which the packet has been sent.
3. The lower level receives the packet at time T_i.

Consider that the first two packet needs a time t and t' respectively to cross the network and that the difference between the clock of the lower and the upper level is o. This means that time T_{i-2} is the sum of the time at which the packet started T_{i-3}, the time t needed to reach the upper layer and the difference o between the two clocks (because T_{i-2} is calculated in the local time of the upper layer while T_{i-3} in the time of the lower layer)

$$T_{i-2} = T_{i-3} + t + o \tag{11.1}$$

Following the same reasoning we can write the time T_i as

$$T_i = T_{i-1} + t' - o \tag{11.2}$$

From these two formulas we can obtain t and t' and calculate the total transmission time d as sum of the send time t and the response time t'

$$d = t + t' = T_{i-2} - T_{i-3} - o + T_i - Ti - 1 + o = T_{i-2} - T_{i-3} + T_i - T_{i-1}$$

Now let us define a new quantity o_i as

$$o_i = \frac{T_{i-2} - T_{i-3} + T_{i-1} - T_i}{2}$$

From 11.1 and 11.2 we can derive

$$T_{i-2} - T_{i-3} = t + o \tag{11.3}$$

and

$$T_{i-1} - T_i = o - t' \tag{11.4}$$

If we replace equation 11.3 and 11.4 in the definition of o_i we obtain

$$o_i = \frac{t + o + o - t'}{2} \tag{11.5}$$

$$= \frac{2o + t - t'}{2} \tag{11.6}$$

$$= o + \frac{t - t'}{2} \tag{11.7}$$

Finally we can highlight o and write

$$o = o_i + \frac{t' - t}{2}$$

Since t and t' are always positive (being time spans) we can try to add t to o and subtract t to o. Respectively, we obtain an upper and a lower bound of o.

$$o_i + \frac{t' - t}{2} - t' \leq o \leq o_i + \frac{t' - t}{2} + t \tag{11.8}$$

$$o_i + \frac{t' - t - 2t'}{2} - t' \leq o \leq o_i + \frac{t' - t + 2t}{2} + t \tag{11.9}$$

$$o_i - \frac{t + t'}{2} \leq o \leq o_i + \frac{t + t'}{2} \tag{11.10}$$

$$o_i - \frac{d_i}{2} \leq o \leq o_i + \frac{d_i}{2} \tag{11.11}$$

Form the last inequality we can notice that o_i is an estimate of the offset o (which is what we want to calculate) and d_i is a measure of accuracy of the estimate, in fact o is between o_i plus or minus a fraction of d_i.

If o_i is small enough we can synchronise the clock of the lower level using o_i, otherwise the algorithm is repeated.

11.3 Logical time

In some cases time isn't enough to synchronise two operations or maybe it isn't relevant because the only important thing is the order of the operations. Ordering is important because it can also indicate causality. Logical clocks aren't related to physical locks and can order operations.

11.3.1 Happens before relationship

Happens before

The happens-before relationship can be defined as follows

- If an event e occurs before event e' we write

  $$e \rightarrow e'$$

  The events e and e' have to happen on the same machine because we need the local physical clock to understand which event happens before. The happens-before relationship respects the order of a machine.

- If e is the event of sending a message and e' is the event of receiving the same message than e happens before e'

  $$e = \mathit{send}(msg) \rightarrow e' = \mathit{recv}(msg)$$

The happens-before relationship is **transitive** and defines a partial ordering between events. The relation is just a partial order because it's not possible to order events that happen on different machine and are not events of sending or receiving a message. The happens-before relationship also describes a potential causal order (if something happens before something else it doesn't mean that the first event caused the second one).

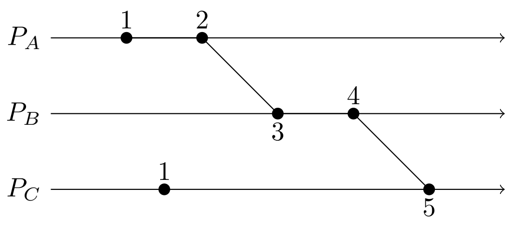


Figure 11.2: An example of scalar clock synchronisation.

Parallel relationship Two events e and e' are in a parallel relationship if they aren't in an happens before relationship, and we write

$$e||e'$$

Scalar clocks (Lamport clocks)

Logical clocks (also called Lamport clocks) implement the happens-before relationship between two events (i.e. something important for our application that we can assume take zero time to happen). Starting from the happens-before relationship we can define our own clock, in particular each process p_i has its own logical scalar clock L_i (also called Lamport clock) represented with an integer number and

- L_i is initialised with 0.
- L_i is incremented before sending a message (and the message is timestamped with L_i).
- When p_i receives a message it updates L_i as

$$L_i = \max(\text{msg timestamp}, L_i) + 1$$

From this definition of Lamport clock we can demonstrate that if an event e happens before an event e' than the Lamport clock of e is smaller than the Lamport clock of e' but not vice versa because there are some events that might not be in a happens-before relationship (for instance events of different machines that aren't messages). In other words the ordering of the Lamport clock is stronger (i.e. if there is an happens-before then we are sure there is a Lamport relationship) than the order of the happens-before relationship.

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

If we add a process number (different for each process) to the Lamport clock than we obtain a full ordering because there aren't ambiguous event anymore. In this case we are also respecting the happens-before ordering.

Totally ordered multicast

Assumptions To achieve totally ordered multicast using scalar clocks we have to assume **reliable** and **FIFO links**.

Protocol Scalar clocks are used, for instance, in totally ordered multicast. Totally ordered multicast delivers messages in the same global order using logical clocks. In totally ordered multicast

- Messages are sent and acknowledged using multicast.
- All messages (including acks) carry a timestamp with the sender's scalar clock.
- Scalar clocks ensures that the timestamps reflect a consistent global ordering of events.

When a node receives a message it

1. Store the message in a queue, ordered according to its timestamp.
2. Sends an acknowledgement (in multicast). The ack isn't queued.

Eventually, all processes have the same messages in the queue. A message is delivered to the application only when it is at the highest in the queue and all its acks have been received (this means that a process has to keep track of the acknowledgements received for every message in the queue). Since each process has the same copy of the queue, all messages are delivered in the same order everywhere.

11.3.2 Vector clocks

Vector clocks are an extended version of scalar clocks (Lamport clocks) that are one on one with the happens-before relationship (the Lamport ordering didn't implied the happens-before relationship).

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

Instead of keeping a single number on each node, we keep a list of numbers at each node. The length of the list is the number of processes in the system (thus we assume that the number of processes is known). Each process p_i has a vector V_i such that

- For node i, the position of i of its vector V_i contains the number of events that have occurred at p_i (calculated with the rules of the Lamport clock)

  $$V_i[i] = \text{number of events happened at } p_i$$

- The position j of the vector V_i (with $i \neq j$) contains the number of events that p_i thinks happened at node j.

  $$V_i[j] = \text{number of events that } p_i \text{ thinks happened at } p_j$$

  In other words $V_i[j]$ is the view of p_i of what happened at p_j.

Vector clock The vector clock V_i of process p_i is handled in the following way

- Is initialised with all zeroes at the beginning.

  $$V_i = [0, 0, \dots, 0]$$

- An event of the local process p_i causes an increment in the number $V_i[i]$.

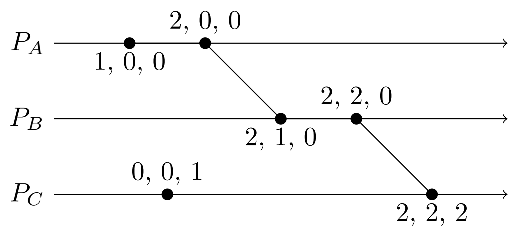


Figure 11.3: An example of vector clock synchronisation.

- When p_i sends a message, it is timestamped with the vector clock of p_i (i.e. V_i). Notice that in this case all the vector is passed, not a single number. Sending a message is also an event, thus before sending a message p_i increments $V_i[i]$.
- When p_i receives a message with timestamp T (that is a vector), it sets every cell j (but the cell i) of its own vector clock with the maximum with the maximum between $V[j]$ and $T[j]$.

$$V_i[j] = \max(V_i[j], T[j]) \quad \forall j \neq i$$

Receiving a message is an event, so after receiving a message p_i has to increment $V_i[i]$

Definitions Here's a list of useful definitions to define an ordering relationship between vectors

- Two vectors V and V' are the same if and only if they are the same at each position.

$$V = V' \Leftrightarrow V[i] = V'[i] \quad \forall i \in 0, ..., length(V) - 1$$

- A vector V is not greater than a vector V' if and only if at each position the elements of V aren't greater than the elements of V'.

$$V \leq V' \Leftrightarrow V[i] \leq V'[i] \quad \forall i \in 0, ..., length(V) - 1$$

- A vector V is smaller than a vector V' if and only if at each position the elements of V isn't greater than V' and the two vectors are not the same.

$$V < V' \Leftrightarrow V \leq V' \land V \neq V'$$

- Two vector clocks V and V' are parallel if and only if V isn't smaller than V' and V' isn't smaller than V.

$$V || V' \Leftrightarrow \neg(V < V') \land \neg(V' < V)$$

Using these rules we can demonstrate that if an event e happens before an event e' (i.e. it exists an happens-before relationship) then the vector clock of e is smaller of the vector clock of e', and vice versa (it's an if and only if relationship)

$$e \rightarrow e' \Leftrightarrow V(e) < V(e')$$

The notation $V(e)$ indicates the instance of the vector V after the event e.

Moreover if two events e and e' are parallel than the related vector clocks are parallel, and vice versa.

$$e \parallel e' \Leftrightarrow V(e) \parallel V(e')$$

In other words the ordering relationship defined by the happens-before relationship is isomorph with the ordering relationship defined by the vector clock.

Vector clock for causal delivery

This protocol uses a variation of vector clock in which an element of the clock is incremented only when a message is send (or when a normal event occurs) and not when a message is received. The protocol is based on the following assumptions

- All messages are sent to every other process (in broadcast).
- The channels are reliable and FIFO ordered.
- It has to preserve the ordering only between messages and replies.
- If a message m_1 arrives before m_2, it doesn't necessarily mean that the two messages are related.

When the process receives a message, it saves it and it doesn't use it until all the previous messages have been received. We are sure that all the previous messages have been received by a process p_i if

- The timestamp at position j (position of the sender) of the message equals the vector clock at position j of the local vector $V_i[j] + 1$ and
$$T[j] = V_i[j] + 1$$
- The timestamps at the other positions are lower than the elements in the vector.
$$T[k] \leq V_i[k] \quad \forall k \neq j$$

For instance consider a process P_3 with $V_3 = [0, 0, 0]$ that receives a message timestamped as $T = [1, 1, 0]$ from P_2. The process P_3 saves the message for later because the second condition seen before doesn't hold, in fact (consider 1-index vector)

- $T[2] = V_3[2] + 1 = 1$, but
- $T[1] > V_3[1]$

This means that P_3 has received a message from a process P_2 that had previously received a message from P_1. To verify that this is true let us consider the following sequence of events

1. P_1 sends a message in broadcast. Before sending the message, the vector V_1 is updated
$$V_1 = [0, 0, 0] \rightarrow [1, 0, 0]$$
The message is sent in broadcast to P_2 and P_2 using in both cases $T = V_1 = [1, 0, 0]$.
2. The message is received by P_2 that updates its vector.
$$V_1 = [0, 0, 0] \rightarrow [1, 0, 0]$$
3. P_2 sends a message. Before sending a message it updates its vector
$$V_2 = [1, 0, 0] \rightarrow [1, 1, 0]$$
and then it sends the message in broadcast using the timestamp $T = V_2 = [1, 1, 0]$.
4. P_3 receives the message sent by P_2 (but P_3 hasn't received the one sent by P_1 yet). This message is put in hold because $T[1] > V_3[1]$, that is because P_2 says that it has received a message from P_1 before sending its message but P_3 hasn't received such message (otherwise $V_3[1]$ would have been 1).

Chapter 12

Mutual exclusion

Mutual exclusion allows us to prevent interference between multiple processes that want to access the same shared resource.

Mutual exclusion isn't only applied on the same machine (e.g. when multiple threads want to read or write a variable) but also in distributed systems (e.g. when multiple devices want to use a print, i.e. the shared resource).

Assumptions The protocols we are going to analyse assume **reliable channels and processes**.

Requirements A protocol that handles mutual exclusion should ensure

- The safety property. At most one process can access a resource at a time.
- The liveness property. All requests that request or release a resource eventually succeed. In other words deadlocks or starvation aren't allowed.

Optionally, we can ensure that if a request happens-before another request, then the former is executed before the latter.

12.1 Central server

The easiest solution to handle mutual exclusion is to let a server decide which process can access a resource. For instance if the resource is a printer, the printer can implement a server that receives the printing requests from the processes and handles them one at a time.

Disadvantages The main disadvantage of this solution is that the server is the bottleneck of the system because it has to handle all the requests. In other words the problem is that the system is centralised.

12.2 Mutual exclusion using scalar clocks

Mutual exclusion can be achieved using scalar clocks.

Algorithm To request access to a shared resource

1. A process P_i broadcasts a resource request message m to tell the other processes that it wants to use the resource. The message m is timestamped with the local timestamp T of the process P_i.
2. If a process P_j receives a request m and it's not interested in the resource requested by P_i then the message m is acknowledged only to the sender.
3. If a process P_j receives a request m_i and it is interested in the same resource (i.e. P_j has sent a request message m_j for the same resource) or is already using the requested resource then the timestamps are confronted. In particular
   - If the timestamp of m_j is lower than the timestamp of m_i, than the request m_i is kept by P_j and P_j doesn't acknowledge.
   - Otherwise P_j acknowledges to the sender P_i.
4. The process P_i (i.e. the process that requested the resource) can access the resource when it receives all the acknowledgements.
5. P_i sends an acknowledgement to all the processes queued (i.e. those processes that sent a request that hasn't been acknowledged by P_i because it was occupying the resource) when it stops using the resource.

Assumptions Such process works under the following assumptions

- The communication channel is reliable.
- The processes are reliable.

This solution works because two process could access the same resource at the same time only if they emitted the request with the same timestamp, but because we use Lamport clock this is impossible.

Happens-before order This protocol also preserves the happens-before order between processes, in fact if A sends a request for a resource to B (suppose a system with just 2 processes) and because B sees the request of B, it sends a request for the resource to A, then A has access to the resource before B. In other words A sends the request before, so it has access to the resource before. This happens because B increments its Lamport clock when it receives the request and when B sends its request, it has a bigger timestamp than A.

12.3 Token ring

The token ring protocol assigns an identifier (i.e. a number) to each process and organises them in a ring. The processes aren't necessarily physically connected in a ring configuration, in fact the ring connection is just a logical connection. Each process in the ring is connected to the process with the next identifier.

Algorithm	Messages per entry	Delay before entry	Problems
Centralised	2	2	centralised
Lamport	$2(n-1)$	$2(n-1)$	Crash of any process
Token ring	1 to ∞	0 to $n-1$	Lost token, process crash

Table 12.1: Characteristics of the the mutual exception implementations.

Token The process with lower identifier generates a token and sends it to the next process in the ring. When a process P receives a token

- If the process P needs the resource associated with such token, P can use the resource.
- If the process P doesn't need the resource associated with the token, P passes the token to the next process.

This protocol works if there is a single token (i.e. no process generates a new token for the same resource or the network is unreliable and the token gets lost).

Chapter 13

Leader election

Many times in a system we have to identify a node that has to do some special operations (e.g. coordinate the other nodes). Such node could be pre-defined but if it crashes then the system has to elect a new node to execute such special operations. The protocols that allow to choose a new special node are called **leader election** protocols.

Assumptions The leader election protocols we are going to analyse work under the following assumptions

- The **nodes** have to be **distinguishable** (e.g. we can assign an id to each node without repeating the same id for two nodes).
- The **nodes know each others** (i.e. the ids of the other nodes).

The nodes don't have to be reliable, in fact even if new nodes join or some nodes fail, the leader can be elected without problems.

Goal The goal of all the leader election protocols is to agree on the leader node. The leader node is the non-crashed node that at the end of the election has the highest ID.

13.1 Bully election algorithm

Assumptions The bully election algorithm works under the leader election assumptions and add the following ones

- The links between the nodes are reliable.
- It is possible to determine which node has crashed. To agree on the fact that a node has crashed we have to make assumptions on the latency of the network. In fact we have to understand if a node is not sending acks because the network is slow or unreliable or because the node has crashed. In other words we have to be in a synchronous system.

Algorithm The bully election algorithm works this way

1. When a node discovers that the leader has crashed, it starts an election by sending a special election message to all nodes that have a bigger id. This message means that the node proposes itself as new leader.

2. When a node receives an election message it immediately replies with a not-acknowledgement because the sender has a lower id. The node starts another election.

3. This process goes on until a node doesn't receives any not-acknowledgement. If it doesn't receives a KO message, than it is the node with the greater id (because the KO is sent only by nodes with greater id and the past leader, that had a greater id, has crashed and cannot send messages).

4. When a node discovers to be the one with the new highest id, it broadcast coordination-message that tells to all the other nodes that it is the new leader.

This protocol is called bully protocol because nodes with greater ids block (like a bully) the request of nodes with smaller id that want to be leaders.

Parallel election If more nodes discover that the leader crashed, multiple elections can be held in parallel.

Restarted nodes When a node restarts, it starts an election so that if the new node has the highest id, it can be the leader.

13.2 Ring based algorithm

To use this algorithm the nodes have to be in a physical or logical ring configuration.

Algorithm The ring based algorithm works this way

1. When a node discovers that the leader has crashed, it starts a new election by creating an election message that contains its id. The message is forwarded to the next node of the ring.

2. Every node of the ring adds its id to the election message and forwards it (skipping the node that crashed).

3. When the message comes back to the node that started the election, it checks the highest id in the message and elects the node with such id as new leader. The new leader is elected sending a message (that says who the new leader is) to every node.

Multiple elections Multiple elections can happen at the same time (only one leader is elected).

Restarted nodes When a node restarts, it starts an election so that if the new node has the highest id, it can be the leader.

Chapter 14

Capturing global state

In many cases (e.g. backup, logging, debugging) it's useful to capture the global state of the system, that is the local state of each process and the messages that are travelling on the network.

On a centralised system this task is quite easy because we can use the clock of the machine to stop all the threads and processes and capture the state of the memory. On a distributed system we have no centralised clock, thus it's harder to synchronise the processes.

Getting the global view is hard also because every process has its local view of the system and obeys to a different clock. For this reason it's hard to get a global picture of the system at a specific time. The only way to really get the whole picture of a system we should stop the execution on every process until everyone has captured its local state. This solution would block the system for a very long thus can't be applied in real world applications.

14.1 Cut and snapshot

Distributed snapshot A distributed snapshot is a picture of the application that can be collected in a distributed way.

Cut A cut is a picture taken at different point in global time at the different components of the application. Putting together all the points we get the cut.

In particular a cut C of a system S with N processes is the union of the history h_i of all processes up to a certain event k_i (in general different for every process)

$$C = \bigcup_{i=1}^{N} h_i^{k_i}$$

where h_i is the set of all events that happened in process p_i until the event k_i

$$h_i =< e_i^{k_0}, ..., e_i^{k_i} >$$

14.1.1 Consistency

If we accept the fact that it's very hard to get a vertical cut (i.e. a picture of every process at the same global time), we should understand if a cut is consistent or not (i.e. if it correctly represents some state of the system in which the application could have been).

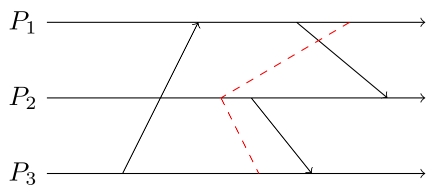


Figure 14.1: A consistent cut (in red)

Inconsistent cut To define what a consistent cut is, we have to define an inconsistent cut. An inconsistent cut is a cut in which

- The snapshot of a process P_2 is taken after the arrival of a message m sent by P_1.
- The snapshot of the process P_1 is taken before sending m.

In other words the cut is inconsistent if the global snapshot obtained has a message that has been received by a process but it has never been sent.

On the other hand a state in which a message has been sent but hasn't been received yet is consistent.

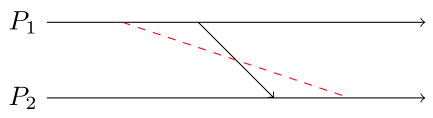


Figure 14.2: An inconsistent cut (in red)

Consistent cut Using the formal definition of cut seen before, we can say that a cut C is consistent if for any event $e \in C$, it also includes all the events that happened before e

$$\forall e, f : e \in C \land f \to e \Rightarrow f \in C$$

More informally a cut is consistent if it's not inconsistent, that is if we cannot find a message m that it has been received (event e) but not been sent (event f).

Consistent cuts are the closest representation of a snapshot.

14.1.2 Chandy-Lamport algorithm

The Chandy-Lamport algorithm is used to obtain a distributed snapshot.

Assumption The Chandy-Lamport algorithm assumes that

- The links are FIFO and reliable.
- Nodes are reliable.
- The graph of processes is strongly connected (from each process we can reach any other process).

Protocol Any process in the system can decide to start a snapshot. When a process decides to start a snapshot, it atomically (i.e. in zero time, suspending its execution)

1. Records its internal state to save storage .
2. Sends out a special message (called **marker** or **token**) to all outgoing channels.
3. Starts recording what arrives from each incoming channel. During the recording phase the process operates normally, that is if a message arrives it is handled normally.
4. Stops recording a channel when a token arrives from a channel that is being recorded.
5. Ends the snapshot phase when a token has been received from every incoming channel.

When a process receives a token, it atomically

- Saves its own state.
- Forwards the token to all outgoing channels.
- Starts recording whatever comes from the incoming channels (a part from the channel from which it received the token). While recording incoming messages the process operates normally.
- Stops recording a channel when a token arrives from a channel that is being recorded.
- Ends the snapshot phase when a token is received from every incoming channel.

At the end of this process each node has a snapshot made of

- The local state saved (when it received the token the first time) to storage.
- The messages recorded from each channel before receiving the marker. In particular if the node hasn't started the snapshot phase and has received the first marker from P_1, than the queue with the messages coming from P_1 should be empty.

At the end of this process, every node has obtained a snapshot made of

- The local snapshot saved to storage when the first token arrived (or when the node decided to start the snapshot algorithm).
- The list of messages arrived from each incoming channel until the receipt of the token. If the node hasn't started the snapshot algorithm and has received the first token from process P_1, then the list of messages from P_1 is empty.

For each state we can collect (using whatever algorithm, this part is not part of the algorithm because it's not difficult to implement) a snapshot to obtain the consistent cut. The cut is consistent because every node has collected the incoming messages (i.e. the messages that, when taking the snapshot where still on the network).

Consistency of the cut The distributed snapshot algorithm selects a consistent cut (under the assumption of the algorithm), in fact let e_i and e_j be two different events happening on different processes p_i and p_2 such that e_i happens before e_j

$$e_i \rightarrow e_j$$

Suppose that e_j is part of the cut and e_i isn't. This means e_j occurred before p_j saved its state, while e_i occurred after p_i saved its state. Let $m_1 \rightarrow \cdots \rightarrow m_h$ be the sequence of messages that give rise to the relation $e_i \rightarrow e_j$ (i.e. the complete sequence of events is $e_i \rightarrow m_1 \rightarrow \cdots \rightarrow m_h \rightarrow e_j$). If e_i occurred after p_i saved its state then p_i sent a marker ahead of $m_1 \rightarrow \cdots \rightarrow m_h$. By FIFO ordering over channels and by the marker propagating rules it results that p_j received a marker ahead of $m_1 \rightarrow \cdots \rightarrow m_h$. By the marker processing rule it results that p_j saved its state before receiving $m_1 \rightarrow \cdots \rightarrow m_h$, i.e., before e_j , which contradicts our initial assumption thus the cut has to be consistent.

Multiple snapshots A system can start multiple snapshot and identify every snapshot with an identifier to distinguish the cuts.

14.2 Termination detection

Termination detection aims to determine if the computation of a system has completed or deadlocked. In particular a single node of the system has a limited view of the whole system so it may want to know if, after not receiving some messages for a while, the computation is ended or the system is in a deadlock state.

Diffusing computations In a diffusing computation all processes are idle a part from one process that has to initiate the computation. The computation of all the other processes when receiving a message. After terminating the computation the processes return to the idle state and wait for another message to start another computation.

In other words processes do not start the computation by thyself but they start the computation because they received a message.

A diffusing computation in which it easy to understand if a computation has ended.

Global snapshot to detect termination Global snapshots can be used to detect termination, in fact if we associate a specific condition to a termination state we can check if the global snapshot satisfies the terminating condition. Taking a snapshot is very expensive, therefore this method is used very rarely.

14.2.1 Dijkstra-Scholten termination detection algorithm

The Dijkstra-Scholten algorithm can detect termination in diffusing computations.

Spanning tree The DS algorithm creates a spanning tree of activated processes. In particular

- The root of the tree is the initial process.
- When a process sends a message to another process to activate it, the receipt of the message is added to the spanning tree. If the receipt was already activated, it sends a message to the sender to notify that its computation was already under going.

- When an active node terminates its execution it sends a message to its parent to notify the end of the computation and to ask to be removed from the spanning tree.
- A process keeps track of all processes that it has activated an remains active until all of that processes have ended their computation.
- When the init node ended its execution, the whole computation has ended.

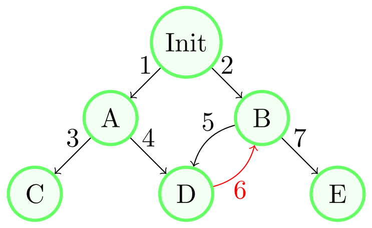


Figure 14.3: Dijkstra-Scholten spanning tree.

For instance in figure 14.3 the initiate process activates the processes A and B. After that the process A sends a message to C and D to start their execution. When process B tries to activate D, it sends back a message to say that it's already in execution. Finally B activates node E. If we remove the links 5 and 6 (because the message 5 hasn't actually started an execution) we obtain a tree.

This algorithm doesn't require to stop any process or to collect local snapshots, it simply collects information about who is the parent and who are the sons in the tree.

Chapter 15

Distributed transactions

A distributed transaction is a transaction that happens on multiple databases.

Transaction To work with transactions we will use the following primitives

- `BEGIN_TRANSACTION` to start the transaction.
- `END_TRANSACTION` to end and commit the transaction.
- `ABORT_TRANSACTION` to abort a transaction. When we abort a transaction we want to bring back the state of the database to the situation before the start of the transaction.
- `WRITE` to write some data.
- `READ` to read some data.

Transaction types Transactions can be

- **Flat**. A flat transaction is the transaction that happens on a single database.
- **Nested**. In nested transactions the databases are organised in an hierarchy (a tree). The transaction starts in the root of the tree and for each child node a sub-transaction is executed. If a transaction aborts in the leaves the abort is propagated to the parent node until it reaches the root that aborts the whole transaction. Sub-transactions operate on a private copy of the data that can be deleted in case of abort or made available to the next sub-transaction if in case of commit.
- **Distributed**. In distributed transactions the database is divided in multiple nodes that aren't organised in a fixed structure.

Among these three structures we will consider the fully distributed database.

15.1 ACID properties

ACID properties Like in a centralised database we want transactions to satisfy the ACID properties.

- **Atomicity**. Atomicity ensures that either the transaction is executed all together or it's not executed at all.
- **Consistency**. Consistency ensures that concurrent transaction can access the database without violating any constraint.
- **Isolation**. Isolation ensures that concurrent transactions do not interfere with each other. Isolation is sometimes referred as **serialisability**.
- **Durability**. Durability ensures that the changes made are permanent.

15.1.1 Atomicity

Private workspaces

To achieve atomicity (i.e. either the transaction is executed all together or it is not executed at all) we can use private workspaces. Each transaction

- Has a private workspace that is a shadow copy of the original database. The transaction modifies its private workspace.
- Deletes the private workspace if it needs to abort.
- Copies the private workspace in the main workspace if it commits.

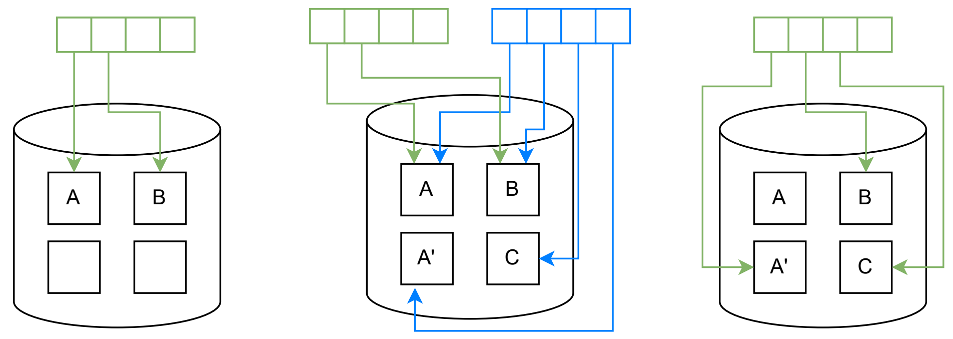
This behaviour can be optimised using indices, in particular

- Each transaction has a private index made of pointers to the blocks in the database. The index is initially a copy of the main index and represents the private workspace on which the transaction works. Notice that in this case we have to copy a new index, which is cheaper than copying a part of the database.
- A transaction modifies its private index.
- If the transaction aborts, the private index can be deleted.
- If the transaction commits, the private index is copied in the main index.

Examples To better understand how this mechanism works let us consider a couple of examples.

- Say we want to add a new tuple. The transaction adds the tuple to the database and adds a pointer only in the private index. The new pointer is copied in the main index only if the transaction commits.
- Say we want to modify a tuple. In this case the transaction adds the modified tuple to the database and adds a private pointer to the modified tuple. Upon committing, the reference of the main index's pointer is moved from the old tuple to the new tuple (i.e. the one indexed by the private index).

The examples are shown in Figure 15.1. The green and blue arrays represents the main and private indices, respectively.



(a) Before execution. (b) During execution. (c) After execution.

Figure 15.1: Atomicity with private workspaces.

Pessimistic approach This approach is pessimistic because it assumes that the transaction may abort so it doesn't delete all the initial state of the local database. In pessimistic approaches if things go bad it's easy to go back to the initial space.

Writeahead log

In this case when a transaction asks to modify a file, the operation is logger and then executed normally (notice that we log the operation before executing it). If the transaction aborts we have to execute (undo) the operations in reverse order.

Optimistic approach This approach is optimistic because we assume that in principle every transaction commits and only in the rare cases in which it doesn't we have to rollback all the operations. The rollback is very costly.

15.1.2 Consistency and Isolation (serialisability)

Structure To control concurrency (i.e. consistency and isolation) we have to analyse three different layers that handle a different part of the transaction, in particular

- The **transaction manager** receives the commands of the transaction (`BEGIN_TRANSACTION`, `READ`, `WRITE`, `END_TRANSACTION`, `ABORT_TRANSACTION`). The transaction manager is typically unique for each transaction or for the entire database. The transaction manager also ensures atomicity.
- The **scheduler** receives the commands from the transaction manager and decides which operation can be executed to ensure consistency (and isolation). The scheduler uses locks or timestamps to ensure such properties. The scheduler can be distributed on every node or on a single node.
- The **data manager** simply executes reads and writes in the order decided by the scheduler without knowing anything about transactions. Every node has a data manager.

Locking

To ensure consistency and isolation we can use Two-Phase Locking (2PL). Notice that Two-phase locking can still lead to deadlocks. The transaction manager asks for lock to the schedulers and asks the data manager to execute the operations. Each data manager operates on the local data.

Locks Two-phase locking is (mainly) based on two types of locks

- **Read locks**, used to read a resource. A transaction can acquire a read lock on a resource only if the resource is free or other transactions have acquired a read lock. This means that multiple transactions can read a resource concurrently.
- **Write locks**, used to write a resource. A transaction can acquire a read lock only if the resource is free or the transaction already has a read lock on the same resource (in this case no other transaction can have a read lock on the resource requested). In other words a resource is written exclusively by a transaction and no other transaction can read concurrently.

Once a transaction releases a lock, it can't acquire any more locks. In other words a transaction has to acquire all locks before releasing one.

Types of locking Two-phase locking (2PL) can be divided in

- **Centralised 2PL** in which a single process provides the lock for every piece of data in the system. The transaction manager has to interact with the central lock manager to obtain and release locks.
- **Primary 2PL** in which
  - Each piece of data has a primary copy on a node.
  - Each node has a scheduler (i.e. a lock manager).

  The transaction manager interacts directly with the nodes, in particular with the data manager.
- **Distributed 2PL** in which there exists a main piece of data on a node and other secondary copies (to improve performance). To access a piece of data, either primary or secondary, the transaction manager has to contact the scheduler (i.e. the lock manager) of the node which has the main copy. The scheduler on node N grants locks on N and interacts with N's data manager.

Timestamps

Consistency and isolation can be achieved also using timestamps. In particular we ensure that an operation with an smaller timestamp is not executed after an operation with a bigger timestamp.

Write operation When the scheduler receives a $write(T, x)$ operation at time ts from a transaction T

- If $ts > ts_{rd}(x)$ and $ts > ts_{wr}(x)$ it performs a tentative $write(x_i)$ with timestamp $ts_{wr}(x_i)$.
- Else abort T since the write request arrived too late.

Read operation When the scheduler receives a $read(T, x)$ operation at time ts

- If the timestamp is bigger than the timestamp of the last write (i.e. $ts > ts_{wr}(x)$)
  - Let x_{sel} be the latest version of x with the write timestamp lower than ts.
  - If x_{sel} is committed perform read on x_{sel} and set $ts_{rd}(x) = max(ts, ts_{rd}(x))$.
  - Else wait until the transaction that wrote version x_{sel} commits or abort then reapply the rule.
- Else abort T since the read request arrived too late.

Optimistic approach Timestamp can also be used with an optimistic approach. In particular in this case all transactions are executed and reordered. At the end of the transaction we have to check if something went wrong and in this case rollback and abort the transaction.

Deadlocks

If we use locking to ensure consistency and isolation we could incur in deadlocks. Deadlocks can be

- **Detected**, in these cases we execute operations normally and if we discover a deadlock we kill one of the transactions that cause the deadlock. Luckily, some studies show that 90% of the deadlocks (at least in databases) involve only two transactions, thus killing one is enough to solve the problem.
- **Prevented**, in these case we avoid deadlocks.
- **Avoided**, in which we guarantee that deadlocks never appear by design.

Centralised deadlock detection Deadlock prevention can be hard in distributed system because we can't suspend the system to analyse the sequence of operation to understand if a deadlock occurred.

To prevent deadlocks every node should send its local picture of the system to a node (the coordinator) that has to build the complete picture of the system and analyse it to look for deadlocks. The problem is that it's impossible to perfectly coordinate the snapshots of nodes, thus the picture obtained by the coordinator may not be updated (i.e. may not refer to a specific moment in time with respect to global time).

Chandy-Misra-Haas algorithm The Chandy-Misra-Haas algorithm is used to detect a deadlock in a distributed way. When a process thinks that a deadlock may have occurred (it's waiting too long), it sends a probe message to the process that holds the resource that the process is waiting (i.e. if process A waits for r_1 and process B holds r_1, than A sends a message to B). If there is a deadlock, than the process that receives the message does the same and when a process sends a probe message to A (i.e. there is a cycle), then A understands that a deadlock occurred.

Distributed deadlock prevention In distributed systems it is also possible to prevent deadlocks using timestamps. In particular the **wait-die algorithm** says that

- If an older transaction T_1 requires a lock for a resource held by a younger transaction T_2 than T_1 has to wait.

- If an younger transaction T_2 requires a lock for a resource held by an older transaction T_1 than T_2 is killed and asked to release all its resources.

This way it's impossible to form a loop (i.e. a deadlock) because the timestamps of the transactions are in increasing order. In other words it's impossible that a resource T_1 waits for T_2 and at the same time T_2 waits for T_1 because the timestamp always have to be in increasing order. The problem with this solution is that the transactions have to be killed and when started again they have a bigger timestamp (are younger) so it's more probable to be killed.

Another approach is the **wound-wait algorithm** that works as follows

- If an older transaction T_1 needs a resource r_1 held by a younger transaction T_2 than T_2 is killed to release r_1.
- If a younger transaction T_2 needs a resource r_1 held by an older transaction T_1 than T_2 waits T_1 to release r_1.

In this case the younger resource is killed only once (whilst in the previous solution it had to be killed many times).

Part VI

Fault tolerance

Chapter 16

Introduction

In a distributed system we want to ensure that the system keeps working even if some nodes crash. In particular we want to ensure

- **Availability**. A system has to be ready for use every time a user needs it.
- **Reliability**. A system has to run continuously for a long time.
- **Safety**. A system never does something bad.
- **Maintainability**. A system has to be easily modified by a programmer. A maintainable system is easier to fix.

It's important to highlight the difference between the first two properties. Consider a system that goes down (and then immediately up) for 1 millisecond every hour. Such system is

- Very available because it's highly uncommon that a user needs the system in that millisecond.
- Very unreliable because it crashes every hour.

16.1 Faults and failures

Another important thing to underline is the difference between faults and failures. In particular

- A system fails (generate a failure) when it is not able to provide its services. A failure is typically the result of an error.
- An error is generated by a fault. Some faults can be avoided (for instance the programmer could write code to catch some unwanted behaviours) but not in all cases (for instance if the memory fails the programmer can't avoid it). Even when a fault can't be avoided it's always possible to handle it.

Consider a program that at some point executes a division by 0. The code doesn't check if the divisor is 0 (i.e. a fault) and the division by 0 results in an error that crashes the system (i.e. a failure).

Faults Faults can be divided in

- **Transient** faults that occur once and never appear again (or at least not frequently). An example of transient fault is a corrupted packet.
- **Intermittent** faults that appear and vanish without apparent reason. An example of this type of faults is faults caused by racing conditions.
- **Permanent** faults that keep existing until we directly fix the fault.

Fault tolerance A system is fault tolerant if a fault doesn't lead to a failure (i.e. the end user doesn't notice that a fault happened because the system didn't crash).

Redundancy A fault can be corrected (or hidden) using redundancy. In particular we can use

- **Information redundancy** like the Hamming code or CRC codes.
- **Time redundancy**, that is if an operation doesn't have success, it is repeated. An example is TPC, in fact if message doesn't reach the destination (it's not acknowledged) we send it again.
- **Physical redundancy**, that is a node is connected multiple times to another node or multiple nodes offer the exact same service.

Chapter 17

Protection against process failures

17.1 Redundancy

Redundancy can be used to mask process failures. In particular redundancy is achieved by replacing a process with multiple processes so that if one process fails, the others continue their execution.

17.1.1 Topology

Redundant processes can be organised in different topologies. In particular we can distinguish between

- **Flat groups**. In flat groups, processes are connected to every other process. An example of flat group is shown in Figure 17.1a.
- **Hierarchical groups**. In hierarchical groups, processes are organised in an hierarchy with a process that coordinates the others (i.e. the process is the root of the tree). An example of hierarchical group is shown in Figure 17.1b.

Group rejoin We have to consider that processes can rejoin the group after a failure. Rejoining requires the new process to synchronise itself with the others. This operation is

- Difficult in flat groups because the new process have to contact and synchronise with every other process.
- Easy in hierarchical groups because the new process only synchronises with the coordinator.

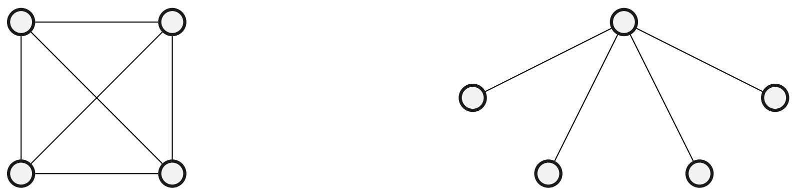

(a) A flat group. (b) A hierarchical group.

Figure 17.1: A types of redundant groups.

Point of failure The hierarchical group topology has a single point of failure. This problem can be solved only with leader election protocols.

Membership agreement Deciding if a process is part of a group is

- Easy in hierarchical topologies.
- Hard in flat topologies because all the other processes have to agree on the fact that a process is in a group.

17.1.2 Group dimension

Tolerating failures gets harder when the number of failures increases. For this reason we have to decide how big a group has to be to ensure that the system keeps working even if multiple processes fail.

Crash failures In case of crash (omission) failures, if we want to tolerate k failures, than the group has to be made of at least $k + 1$ processes.

Byzantine failures In case of Byzantine failures we need to have $2k + 1$ processes to tolerate k failures. The number of processes increases because in this case we get different results from what we expected. Consider for example two processes that exchange message; in this case a Byzantine failure is a message with a modified content. To establish the right content of the message we have to consider the majority. If k messages have a wrong content than we need to have one more correct message (i.e. $k + 1$) so that the majority of messages is correct. If we sum correct and wrong messages we get

$$k + k + 1 = 2k + 1$$

17.2 Server crash

When a client communicates with a server, it might happen that the server crashes. If the client can't even locate the server (for instance because the latter uses an obsolete protocol), then there's not much we can do. On the other hand a problem arises when client and server can communicate but messages (both requests and replies) are lost or the server crashes while executing a request. This is a problem because, when a client sends a request but it doesn't receives an answer, it can't tell if

- The request has been lost.
- The request has been received but the server has crashed while executing it. Furthermore in this case we don't even know at what point the server crashes. Consider for instance a printer (the server) that receives a print request. It might crash before or after printing, thus the printer might have printed something even if the reply never got to the client.
- The request has been received, the server executed it but the replies went lost.

This is a big problem because a client can't tell precisely if a request have been executed. In particular a client can say that either

- A request has been executed **at most once** (zero or one time).

- A request has been executed **at least once** (one or more times).

In no cases a client knows if a request has been executed exactly once.

17.2.1 Client response

Say that a client understands that a server has crashed (at some point), it can adopt four strategies

- Always reissue the request.
- Never reissue the request.
- Reissue the request only when the message hasn't been received.
- Reissue the request only when the message has been received.

If we combine this possible client's behaviours with the possible fail combinations of the server (i.e. before or after executing the request), we can notice that we can never be sure that an action always happens once.

17.2.2 Example

To better understand this problem let us consider a printer that can

- Print a document (D).
- Crash (C).
- Send a confirmation message (M).

The client doesn't know the order in which these actions happen (i.e. the printer might send the message before or after printing and it might crash anytime). This means that when the printer receives a printing request if might execute one of the following patterns

- Print → Send the confirmation message → Crash (PMC).
- Send the confirmation message → Print → Crash (MPC).
- Crash → Send the confirmation message → Print (CMP).
- Crash → Print → Send the confirmation message (CPM).
- Print → Crash → Send the confirmation message (PCM).
- Send the confirmation message → Crash → Print(MCP).

On the other hand the client, since it doesn't know what happened, has to execute always the same action, independently from the fact that a response has been received or not. In particular, as we have seen before, a client can

- Always reissue the request.
- Never reissue the request.
- Reissue the request only when the message hasn't been received.
- Reissue the request only when the message has been received.

The result of the combination of all possible actions done by server and client are shown in Table 17.1. As we can see from the table there is no action that allows the client to be sure that the request has been executed only once (i.e. there is no line with only "once" values).

	PCM	CPM	PMC	MPC	CMP	MCP
Always Resend	duplicate	once	duplicate	duplicate	once	once
Never Resend	once	zero	once	once	zero	zero
Resend on Response	once	zero	duplicate	duplicate	zero	once
Resend on Not Response	duplicate	once	once	once	once	zero

Table 17.1: The result of the combination of client and server's actions.

17.3 Client crash

In a distributed system even a client might crash. In particular a computation started by a client that is dead is called **orphan**. Orphan computations are costly thus the server has to terminate them. In particular a server can get rid of orphans using

- **Extermination** in which the clients log every message sent. After a client crashes, it reboots and asks the server to kill all orphan computations.
- **Reincarnation** in which when a client reboots, it sends a broadcast message to servers who kill all orphan computations.
- **Gentle reincarnation** that works like reincarnation but servers kill old computations only if the client can't be located.
- **Expiration** in which computation expires after some time. In this case the clients don't have to communicate with the server after reboot, in fact they only let the computation expire.

Chapter 18

Agreement in process group

Since now we analysed the system from the point of view of the client. For instance we said that a system needs to have $2k+1$ processes in a group so that the client doesn't notice Byzantine failures.

Now we will focus only on the processes inside a group. In particular we have to describe some protocols that allow the processes to agree on something.

Agreement In a group all non-faulty processes has to agree on something. In particular every process starts with some initial value and the processes have to agree starting from such condition.

During the agreement phase the following condition have to hold

- No two different processes have to agree on different values.
- If all processes starts from a value v then v must be the value on which the processes agree.
- All non faulty processes eventually decide.

Assumptions The agreement protocols assume that

- The system is synchronous. In particular if we want to allow k failures we need at least $k+1$ rounds (and $k+1$ processes in the group). In other words the system must have a time limit of more than $k+1$ rounds. A round is a time interval in which all processes can execute an iteration of the algorithm.
- Only omission failures can occur. That is if a process fail it stops its execution.
- Communication is reliable.

18.1 FloodSet algorithm

Algorithm To describe the FloodSet algorithm we will consider processes that have to agree on the value of a variable V.

The algorithm works this way

1. Each process has to initialise a set W of values in which the process adds its value of V (i.e. what it thinks the value of V is).

2. Each process sends W to the other processes.
3. Each process merges its W with the W received from the other processes.
4. Each process sends the new value of W to all the other processes.

The steps from 3 to 4 are repeated for $k+1$ rounds (where k is the maximum of failures tolerated).

Agreed value The processes agree on

- W if W contains only one element.

$$|W| = 1 \Rightarrow V = W$$

- A standard value if W has more than one element. The processes could also decide to agree on a value with some property (for instance they could choose the maximum or the minimum value).

Number of rounds If after a round (i.e. after all the processes have exchanged a message with the other processes) no process has crashed then every node has the same values in W, otherwise the processes have different values (that's why we need $k+1$ iterations). If the crash happens after a round with no crashes then nothing changes (because after the round with no crashes every process had the same values in W). In other words we need $k+1$ rounds because we need at least one rounds without failures (that's the reason for the $+1$). To further understand this statement we can impose that no process fails (i.e. $k = 0$). In this situation one round is enough to agree on a value.

18.2 Lamport algorithm

Assumptions The Lamport protocol adds the possibility to have Byzantine failures. With this change the assumptions are

- The system is synchronous. In particular if we want to allow k failures we need at least $k+1$ rounds (and $k+1$ processes in the group). In other words the system must have a time limit of more than $k+1$ rounds (where the round is some sort of time unit).
- Both omission and Byzantine failures can occur.
- Communication is reliable.
- Only one process out of 4 total processes can have Byzantine failures.

Number of processes Lamport demonstrated that with less than 3 processes it's not possible to reach an agreement. In general if there are k faulty processes we need $3k+1$ total processes to reach an agreement (more than the $2k+1$ needed from the point of view of a client). The algorithm needs $3k+1$ processes because we need $2k+1$ non-faulty processes to agree on a value if k processes have Byzantine failures.

Agreement With the new assumptions the definition of agreement changes slightly. In particular

- No two non-faulty different processes have to agree on different values.
- If all non-faulty processes starts from a value v then v must be the value on which the processes agree.
- All non faulty processes eventually decide.

Algorithm Consider that all the processes want to agree on the value of a variable V. The Lamport algorithm works this way

1. Each process i sends its value of V (i.e. V_i) to all the other processes. After this phase each process has a vector r containing the values of V of the other processes (including its own).
$$r_i = [V_1, V_2, V_3, V_4]$$
2. Each process sends its vector r to all the other processes. At the end of this phase each process has the vector of every other process (i.e. a matrix). For instance process 1 has
$$\begin{aligned} r_2 &= [V_1, V_2, V_3, V_4] \\ r_3 &= [V_1, V_2, V_3, V_4] \\ r_4 &= [V_1, V_2, V_3, V_4] \end{aligned} \tag{18.1}$$
3. A process can decide the real value of process p by looking at the most frequent value on the p-th column of the matrix. For instance process 1 can decide the real value of V_2 by looking at the values of V_2 in the three vectors. In particular if $r_2[V_2] = 2$, $r_3[V_2] = 1$ and $r_4[V_2] = 2$, then P_1 chooses $V_2 = 2$. The value sent by the faulty process is excluded because we defined the agreement only between non-faulty processes (e.g. if P_3 is faulty, then $r_2[V_3]$, $r_3[V_3]$ and $r_4[V_3]$ can be different because they are excluded).

18.3 Asynchronous systems

Lamport's algorithm allowed us to assume that both omission and Byzantine failures can happen, still we assumed that the system was synchronous. Let's try to drop this assumption and build a protocol for asynchronous systems. This task is rather difficult because in asynchronous systems we can't understand if a component isn't receiving a message because the communication channel is too slow or because the sender failed. The impossibility of agreement in asynchronous systems has been formally proven.

Chapter 19

Reliable group communication

Processes can be organised in groups in which everyone has the same job (redundancy) to tolerate failures. The processes in such groups have to communicate in a reliable way to ensure failure tolerance. Namely, we would like to achieve reliable multicast (every node has to reach all the other nodes) among the processes.

19.1 Point to point

Reliable multicast can be achieved using point to point communication between nodes. This solution is too complex and slow and doesn't behaves well when nodes are added to the group or when they crash.

19.2 Fixed groups and non faulty processes

Before considering faulty processes we'll analyse communication in groups that can't lose or get members and in which there are no faulty processes.

In particular we want to achieve a reliable multicast communication on an unreliable protocol like Ethernet or Wi-Fi.

19.2.1 TCP and scalable reliable multicast

The TCP protocol does the job, in fact every node has to send an acknowledgement when it receives the message and if the sender doesn't receive the ack, the message is sent again.

ACK implosion

TCP works but if used in group communication we could face a problem. Consider the situation in which a process P_0 sends a message to 10 other processes $P_1 ... P_{10}$. If the message is received by everyone than the process P_0 receives 10 acknowledgements (1 for each other process) thus generating a congestion on node P_0.

NACK This problem can be avoided using NACKs instead of ACKs. In particular

- If a process receives a message it does nothing.
- If a process doesn't receive a message it sends a NACK.

Miss detection For this protocol to work we have to be able to detect when a process misses a packet. This can be achieved numbering the packets so that when a process receives a sequence of packets p_1, p_3 it recognises that the packet p_2 went missing and sends a NACK.

NACK implosion

Even using NACKs we could face implosion in fact if a message is received by no one then every node sends a NACK.

Scalable reliable multicast This problem can be avoided using non reliable feedback. In this case NACKs are sent in multicast and when a process notice that it has lost a packet, it starts a timer (of random length) and

- If a NACK arrives before the timer expires, the process stops the timer and doesn't sends a NACK because some other process has already sent it.
- If no NACK arrives after the timer has expired, the process sends a NACK in multicast.

Using this procedure we assure that the receiver only gets one NACK because only the node with the shortest timeout eventually sends a NACK.

Delay This solution works but

- It introduces some delay to the communication because the NACK is sent only after the shortest timeout (i.e. not immediately after noticing the faulty message).
- It forces all the processes to process NACKs.

19.2.2 Hierarchical feedback control

In big groups scalable reliable multicast can't be applied, instead we can use hierarchies.

Structure In particular a group of nodes is divided in subgroups each with a different coordinator. Each subgroup can use scalable reliable multicast or another multicast protocol and the coordinators communicate between each other. The coordinators (i.e. the subgroups) are organised in a tree structure where the sender is in the root subgroup.

Queues If a coordinator doesn't receive a message (using the methods seen for scalable reliable multicast) it can ask the parent coordinator to transmit a message again. Each coordinator has a buffer (a queue) of sent messages to ensure that they are delivered to each node. A message can be removed from the queue of a coordinator if

- All the receivers in the subgroup have acknowledged the message.
- All the child coordinators have acknowledged the message.

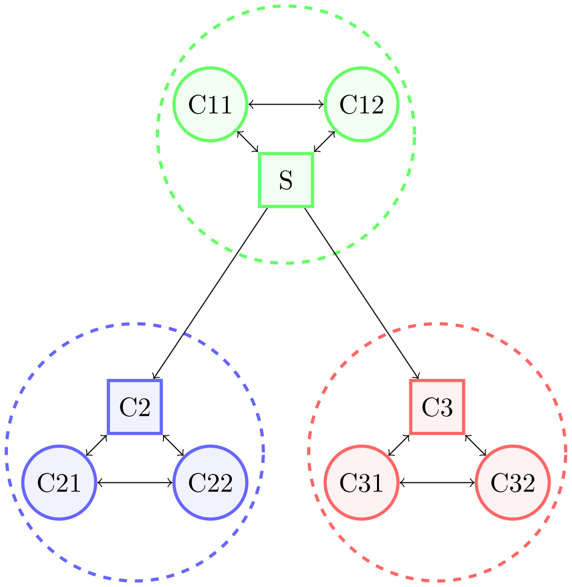


Figure 19.1: The structure used in hierarchical feedback control.

Hierarchy problem The problem with this protocol is that the hierarchy (the tree) has to be constructed and maintained, in fact the sender has always to be in the root sub-group.

19.3 Faulty processes

If in a group some processes can fail, then we have to define better what multicast is and we have to find some protocols to implement it.

The atomic multicast problem If some processes in a group might fail, messages should be

- either delivered to anyone in the same order (i.e. all processes receive the same messages in the same order)
- or to no one.

This means that if an application passes a packet to a middleware, it can be sent immediately to every other process or to no process. This is known as the **atomic multicast problem**.

Closed synchrony Ideally, to solve the atomic multicast problem, we would like

- Any two processes that receive the same multicast messages or observe the same group membership changes to see the corresponding events in the same order.
- A multicast to a process group to be delivered to its full membership (i.e. to all members of the group). The send and delivery events should be considered to occur as a single, instantaneous event.

This behaviour is called **closed synchrony** and can't be completely achieved because we can't know if a failed process has received and processed a message before failing.

19.3.1 Virtual synchrony

Since close synchrony can't be completely archived, we have to relax some assumptions and goals and solve the atomic multicast problem under less strict assumptions. Virtual synchrony is a form of reliable multicast that solves the atomic multicast problem.

Group view Before defining assumptions and goals of this new form of reliable multicast let us introduce some vocabulary. A group view is the set of processes to which a message should be delivered. A view change happens when a process joins or leaves (because of a crash) a group. Messages have to be delivered before or after view changes. When a group view changes, a group view messages is sent in multicast.

Assumptions

Virtual synchronisation assumes

- It's **possible to detect failures**.
- It's **possible to detect new nodes joining the group**.
- It's <u>not</u> possible to know **if a process has received a message or not**.

Goal

When a process crashes, it is purged from the group. In this context

- Messages sent by a correct process are handled to all correct processes.
- Messages sent by a failed process are handles to all correct processes (before the sender fails) or to no process.
- Only relevant messages are received in a specific order.
- The minimal ordering requirement imposes that group view changes have to be delivered in consistent order with respect to other multicasts and with respect to each other.

These goals and the assumption stated earlier lead to a form of reliable multicast said to be **virtually synchronous**.

Message ordering As we have just seen, virtual synchrony allows messages to be ordered in various way (the third goal). In particular we can achieve

- **Unordered multicast** if messages are not ordered.
- **FIFO-ordered multicast** if messages are ordered with a FIFO policy.
- **Causally-ordered multicast** if only causally-related messages are ordered.

Additionally, we can guarantee total ordering. This property ensures that, for whatever type of ordering, all processes have to receive a message in the same order. A summary of all type of multicast ensured by virtual synchrony is shown in Table 19.1.

Multicast	Basic message ordering	Total delivery order
Reliable multicast	None	No
FIFO multicast	FIFO	No
Causal multicast	Causal	No
Atomic multicast	None	Yes
FIFO atomic multicast	FIFO	Yes
Causal atomic multicast	Causal	Yes

Table 19.1: Type of ordering for the different types of multicast protocols.

Protocol

To achieve virtual synchronisation we can use FIFO reliable point-to-point channels in which messages are queued until all nodes receive the message.

Key concept The key concept in virtual synchrony is that, when a process P sends a message and then fails

- Either all processes receive the message from P before one process notice that P crashed
- Or no process receive the message from P before one process notice that P crashed

In other words the following situations aren't admitted

- One process receives the message before and another receives it after the group notices that P crashed.
- All process receive the message after the group notice that the process P crashed.

A visual representation of allowed behaviours in virtual synchrony is shown in Figure 19.2.

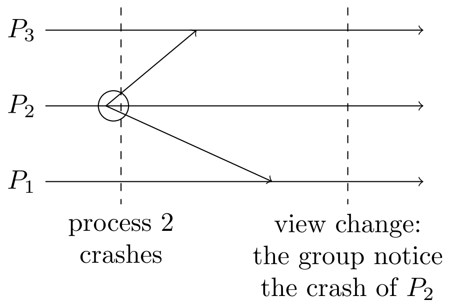
19.3.2 ISIS

The ISIS protocol implements virtual synchronisation in a group of processes.

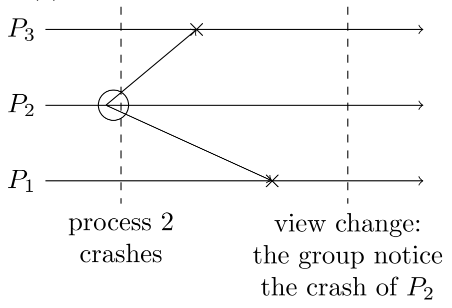
Implementation To ensure that a message from a faulty process is received by everyone or by no one (before noticing the failure), every process keeps the messages sent in a queue until it is sure that all processes received it (i.e. acknowledged).

The algorithms works this way

1. When a process notices that a process has failed it sends a view change message to all the other processes.
2. When a process receives a view change messages it
   (a) Stops sending new messages until the new view in installed.
   (b) Multicasts all pending unstable messages (i.e. the messages in the queue) to the non-faulty processes of the group.
   (c) Marks the messages in the queue as stable.
   (d) Broadcasts a flush message.



(a) The message is received by all processes.



(b) The message is received by no process.

Figure 19.2: Allowed behaviours in virtual synchrony.

3. Each process installs the new view as soon as it has received a flush message from each other process of the new view.

4. Each process can start sending messages.

Chapter 20

Distributed commit

Commit protocols in distributed databases are a form of agreement, in fact all nodes have to agree on the fact that a transaction has to commit.

In particular in distributed commit we have to ensure

- **Agreement**. All the nodes have to agree on the fact that a transaction commits.
- **Validity**. If all nodes agree on the commit, than the transaction has to commit.
- **Termination**. Termination can be
  - **Strong**. All non-faulty processes decide.
  - **Weak**. If there are no fault, all processes agree.

Implementations Distributed commit is achieved using

- **Two-phase commit** (2PC).
- **Three-phase commit** (3PC).

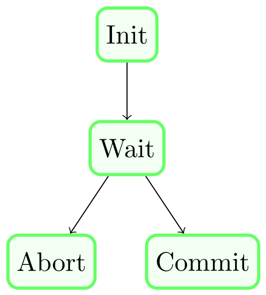
Components In both algorithms we can identify two components

- A **coordinator**, i.e. the transaction manager.
- A **replica**, i.e. the resource manager.

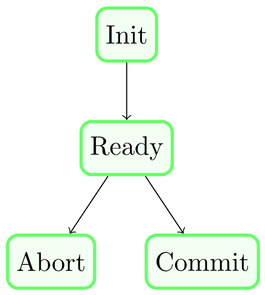
20.1 Two-phase commit

Assumptions The The two-phase commit protocol assumes that

- There is **stable storage** at each node.
- No node **crashes forever**.
- The data in storage is **not corrupted during a crash**.
- **Any two nodes can communicate** with each other.



(a) Coordinator structure.



(b) Replica structure.

Figure 20.1: Two phase commit structure.

Protocol The two-phase commit (2PC) protocol works as follows

1. The coordinator sends a query to commit message to all replicas and goes in the **wait** state. In particular the coordinator waits the responses of the replicas.
2. Each replica that receives the query to commit executes the transaction up to the point in which it has to commit the query. At this point the replica is in the **ready** state and sends a message to the coordinator to say if the query has to be committed or not. The replica doesn't commit the transaction yet.
3. The coordinator receives all the yes/no commit replies from the replicas and decides if the transaction should commit. In particular
   - If all the replicas replied with a *yes commit* message
     (a) The coordinator goes in the **commit** state.
     (b) The coordinator sends a commit message to all replicas.
     (c) Each replica goes in the goes in the **commit** state.
     (d) Each replica commits the transaction and releases the locks.
     (e) Each replica sends an acknowledgement to the coordinator.
     (f) The coordinator completes the transaction when it receives all the acks.
   - If at least one replica votes *no commit*, then
     (a) The coordinator goes in the **rollback** state.
     (b) The coordinator sends a rollback message to all replicas.
     (c) Each replica goes in the **rollback** state.
     (d) Each replica undoes the transaction and releases the locks.
     (e) Each replica sends an acknowledgement to the coordinator.
     (f) The coordinator undoes the transaction when it receives all acks.

Termination Two-phase commit only reaches weak termination in $k-1$ rounds .

20.2 Three-phase commit

Coordinator and replica fail Three-phase commit (3PC) solves the main issue of two-phase commit. Consider the situation in which both the coordinator and the first replica that receives the *commit* message crash. In this case if a new coordinator is elected it will never be able to commit again because it will never receive a message from the replica that failed.

Three-phase commit solves this problem adding a intermediate state, called **precommit** state, between the wait and commit states, in fact

- If the coordinator fails before going in the precommit state the replicas can assume that the transaction failed and can abort.
- It's not possible that a replica commits before all the others know if they have to commit or not. In other words the precommit state ensures that all replicas are aware of the action to perform (i.e. commit or abort) before actually doing it.

Protocol Three-phase commit adds a phase to 2PC. In particular

1. after receiving all *yes commit* messages the coordinator goes in the **precommit** state and asks the replicas if they are ready to commit sending a **readyToCommit** message.
2. Each replica goes in the **precommit** state.
3. Each replica replies to the coordinator saying that it's ready to commit (i.e. it's in the precommit state).
4. When the coordinator receives all the replies it starts the commit phase like in 2PC.

Termination Three-phase commit reaches strong termination. In this case the number of rounds needed is higher unless there are no failures, in this case the protocol only needs 3 rounds.

20.2.1 Solving two-phase commit criticisms

Three-phase commit has been created to solve 2PC main issues. In particular 3PC handles participant and coordinator failures.

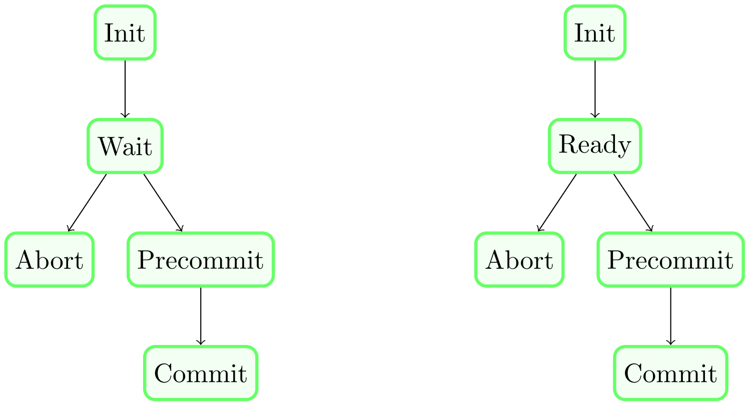
Participant failure

Let us consider a situation in which a participant of the distributed transaction fails.

2PC Using 2PC the coordinator would have been blocked in the **wait** state in fact

- If the participant fails after the coordinator has allowed everyone to commit, then the coordinator has already gone in the **commit** state.
- If the participant fails before the coordinator has started the transaction (i.e. when it's in the **init** state), then the transaction never even starts.

The coordinator, if stuck in the **wait**, can only wait some time after that the transaction aborts because it assumes that a node failed.



(a) Coordinator structure (b) Replica structure.

Figure 20.2: Three phase commit structure.

3PC Three-Phase Commit allows the coordinator to be also stuck in the **precommit** state. In this case the coordinator already knows that the transaction on the failed node should commit so it (the coordinator) can commit and ask non-failed processes to commit. When the failed node restarts, the coordinator can ask it to commit the transaction.

Coordinator failure

Assume now that the coordinator of the distributed transaction fails.

2PC Upon a coordinator fail, the participants can get stuck in the **ready** state. In this situation, a participant can start a timer and decide to abort when the timer expires. This solution is based on the idea that we can assume the coordinator failed if it doesn't responds for some time.

3PC In Three-Phase Commit, when a participant realises that the coordinator failed it can contact the other participants and decide what to do depending on the answers. In particular a participant

- **Aborts** if at least **one** participants is in the **abort state**.
- **Aborts** if the **majority** of the participants are in the **ready state**.
- **Aborts** if at least **one** of the participants is in the **init state**.
- **Commits** if at least **one** of the participants is in the **commit state**.
- **Commits** if the **majority** of the participants are in the **precommit state**.

Chapter 21

Recovery techniques

Recovery techniques allow us to bring back a process to a valid state after a failure.

Modes In particular we can identify two different ways to recover a process

- **Forward recovery** in which the process is brought to a new state from which execution can restart. Some examples of forward recovery are error correction codes.
- **Backward recovery** in which the last previous valid state is recovered. An example of backward recovery is the use of acknowledgement.

Backward recovery In backward recovery we have to save all the states until they are not useful anymore. For instance we have to keep all packets in a buffer until an acknowledgement arrives.

Two implementations of backward recovery are

- **Checkpointing** in which the entire state of the system is saved. When we have to restore the last state we reload the last checkpoint. This technique is quite expensive.
- **Logging** in which all operations are saved (logged). To obtain the last state we start from the initial state and we apply the operations.

Usually this two techniques are combined, in fact we can periodically take a snapshot of the system (checkpoint) and we use logging between snapshots. When we have to reload a valid state we start from the last valid snapshot and we apply all the operation logged after that snapshot.

21.1 Checkpoints

To implement checkpoints we use distributed snapshots.

Problems The problem with checkpoints in a distributed system is that every process works on a local clock and the processes can't be synchronised so if a process asks to start a checkpoint procedure, every process takes its local snapshot at a different time. In other words it's not possible to synchronise all processes to take a snapshot in the same global instant.

Coordinated snapshots A process, i.e. the coordinator, periodically starts a snapshot procedure sending a message to the other processes.

This solution needs coordination, but it would be better for processes to create snapshots independently (without coordination) and then collect all the snapshots to generate a checkpoint.

Failures The independent solution is easier for single processes but requires some extra processing to rebuild a correct snapshot. In particular, checkpoints might be inconsistent if a process fails, i.e. they might generate an inconsistent state. In this cases the processes have to

1. Find the inconsistency.
2. Find a new checkpoint that is consistent. This set is called **recovery line** and is the first consistent set of snapshots.

To find the recovery line all processes have to keep all the previous snapshots because if there is no valid checkpoint then the system might have to start back from its initial state (domino effect).

21.1.1 Algorithm

Creating snapshots

A process P_j

- Creates a snapshot of its local state and tags it.

$$c_{j,0}, ..., c_{j,n}$$

- Tags the intervals between the snapshots.

When process P_j sends a message it also includes the tag of the last snapshot so that the receiver knows that its local snapshot depends from the sender's snapshot sent with the message. For instance if P_j sends a message between checkpoint $c_{j,x-1}$ and $c_{j,x}$, then the messaged is tagged with checkpoint $c_{j,x}$.

Collecting snapshot

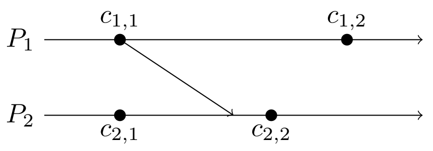
When we have to create a checkpoint all the snapshots (taken in a distributed way) are collected from a process that creates a representation of the dependencies between the various snapshots. The dependencies are used to create a graph that represents such dependencies. We can use two graphs

- The **rollback dependency graph**.
- The **checkpoint dependency graph**.

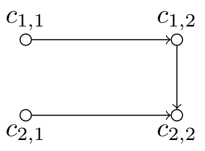
Rollback dependency graph

Graph In the rollback dependency graph

- Every send or receive message event correspond to a node.
- Consecutive events on a single process are connected (from the older to the newer).
- The checkpoint after a send event is connected with the checkpoint after the receive event.

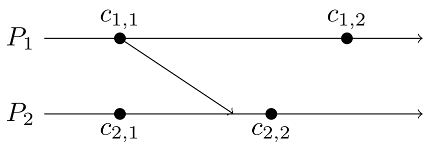


(a) Dependencies exchanged between processes.

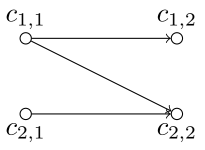


(b) The dependency graph.

Figure 21.1: A rollback dependency graph.



(a) Dependencies exchanged between processes.



(b) The dependency graph.

Figure 21.2: A checkpoint dependency graph.

Recovery line To find the recovery line we have to

1. Consider the last snapshot for every process as the recovery line.
2. Mark the checkpoint before the failure.
3. Follow the arrow that starts from the marked nodes and mark the destination node.
4. When there are no more nodes to expand the algorithm stops. The recovery line is the set of the rightmost snapshot not marked for every process.

For instance in figure 21.1b if the node $c_{1,2}$ is marked we have to follow the arrow and mark node $c_{2,2}$. No arrows exit from $c_{2,2}$ so the algorithm stops and the recovery line is $\{c_{1,1}, c_{2,1}\}$.

Checkpoint dependency graph

In the checkpoint dependency graph

- Every send or receive message event correspond to a node.
- Consecutive events on a single process are connected (from the older to the newer).
- The checkpoint before a send event is connected with the checkpoint after the receive event.

Recovery line To find the recovery line we have to

1. Delete all the snapshots immediately before a failure.
2. Consider the rightmost nodes for every process as the recovery line.
3. Discard a node if it exists a dependency (also indirect, i.e. made of more arrows) with another node the recovery line. A dependency exists when starting from a node is possible to reach another nodes following some arrows. For instance in figure 21.2b there exist a dependency between $c_{1,1}$ and $c_{2,2}$. Practically, consider two nodes n_1 and n_2 in the recovery line. We can eliminate n_2 if it's possible to reach n_2 starting from n_1.

4. When there are no more dependencies the algorithm stops. The recovery line is the set of the rightmost snapshot remained for every process.

21.1.2 Coordinated checkpointing

In coordinated checkpointing no useless checkpoints are ever taken. In particular

1. The coordinator sends a checkpoint request.
2. Receivers take the checkpoint and queue other outgoing messages delivered by the application.
3. When done the receiver sends an acknowledgement to the coordinator.
4. When the coordinator receives all the acknowledgements sends a checkpoint done message to the other processes.

21.2 Logging

Logging protocols

- Save the actions done by the processes.
- When a fail occurs, reply the logs (i.e. the operations) from a save point (i.e. from a checkpoint or from the initial state).

It's important to notice that a distributed system can use logging only if it (the system) is **piecewise deterministic**. A system is piecewise deterministic if execution proceeds deterministically between the receipt of two messages. Otherwise, if execution was stochastic, we could repeat the same sequence of operations.

21.2.1 Messages

A message is

- **Stable** if it's logged and save to memory.
- **Unstable** if it's not logged.

For every unstable message m we can define

- A set $DEP(m)$ of processes that depend (directly or not) from the message m. A **process depend on m if it's the receipt of the message** m or is the receipt of a message m' that depends on m.
- A set $COPY(m)$ of processes that have the message m in memory but not in storage (i.e. it's not logged).

Orphan processes A process Q is an orphan process if it's a survivor (i.e. it's not faulty) and exists a message m so that Q is in the set $DEP(m)$ and all processes in $COPY(m)$ have crashed.

Orphans are problematic in logging because there is no way to reply to m because all the copies of m have been lost because they where only in memory of processes that have crashed.

All orphan processes have to be terminated. In other words if all processes in $COPY(m)$ crashed, then all the processes in $DEP(m)$ have to be killed.

We kill a process to bring it back to a valid state.

21.2.2 Pessimistic and optimistic logging

Unstable messages can be handled in different ways. In particular we can distinguish between pessimistic and optimistic approaches.

Pessimistic logging In pessimistic logging

- Every unstable message is delivered to at most one processes. This way the receiver is always in $COPY(m)$ (where m unstable) unless explicitly discarded by the receiver.
- The receiver can't answer the message until it's not saved to storage so no orphans are created.

This approach is pessimistic because it assumes that an error will happen so it doesn't take the risk of creating orphans.

Optimistic logging In optimistic logging

- All processes send and receive messages without logging.
- Messages are logged asynchronously (i.e. periodically).

This approach may create orphans and in such case the orphan processes have to be killed and rolled back to a previous valid state.

Part VII

Replication and consistency

Chapter 22

Introduction

Replication of data on different components of the network is widely used in distributed systems. Among all the advantages of data replication we can highlight

- **Fault tolerance**, in fact if data is lost on a node the other nodes still have a valid copy of such data that can be accessed.
- **Availability**, in fact a device can save a copy of some data locally so that it can always be available even in case of network failures or excessive load. The availability a is computed as

  $$a = 1 - p^n$$

  where p is the probability of a failure and n is the number of replicas. Data replication is widely used in mobiles to increase availability.
- **Speed**, in fact a client can access faster to some (geographically) closer. The Web uses replication widely, in fact caches, proxies and geo-replicated datastores are all examples of replicas. In particular replicated data increases throughput and decreases latency. Geographic distance between nodes is very important in distributed systems in fact the round trip time of a message exchanged between very distant nodes might be very significant (i.e. even a human can notice the difference).
- **Load balancing**, in fact if data is replicated on multiple nodes, the clients can send requests to different servers thus decreasing the load on a single node of the network.

To sum all this up, data replication allows to improve the reading the data that might not be available otherwise.

22.1 Problems

The main problems related to data replication are

- **Consistency**.
- **Overhead**.

22.1.1 Consistency

Data replication works perfectly when data is read-only. On the other hand when data can be modified we have to be sure that every copy of such data is updated. In particular we can identify two situations in which things might go wrong (i.e. conflicts)

- **Write-write conflicts**. Write-write conflicts happen when two nodes are trying to write the same data.
- **Read-write conflicts**. Read-write conflicts happen when a node is trying to read some data that is being modified.

22.1.2 Overhead

Overhead also has to be taken in consideration, in fact in some cases it might result that copying and updating the data on every node is more expensive than keeping a single copy.

Levels of consistency To avoid overheads we have to consider data replication as a trade-off and identify different types of consistency that allow to speed up data access with some relaxed constraints. In other words in some cases we have to ensure less consistency to avoid overheads.

22.2 Distributed data stores

To analyse data replication models and protocols we will use the distributed data store model. This model gives the illusion of a single database on which items can be read and written.

Register Each item of the datastore is saved in a register, i.e. a key-value pair

```
register = <key, value>
```

Operations The basic operations that can be done on a distributed datastore are

- Read a value a from a register with key k.

$$R(k)a$$

- Write a value a to the register with key k.

$$W(k)a$$

Caches Every process that uses the data store is connected to a local cache. The complete structure of the system is transparent to the process. The system is actually made of all the caches.

Goal The goal of the distributed datastore is to propagate all the edits to every cache in the system. In particular we want a process to see the last value written, when reading a value from its cache. In general it's impossible to guarantee that the value read is exactly the last value but we can develop some protocols that can guarantee the same behaviour with less guarantees.

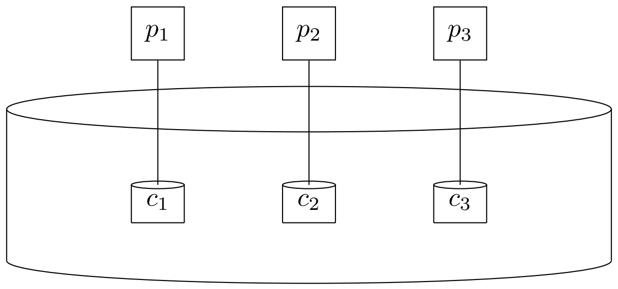


Figure 22.1: Structure of a distributed datastore.

22.2.1 Definitions and models

We have already said that it's impossible to guarantee a strictly correct behaviour so we have to relax some guarantees. We can also say that

- Strict guarantee are costly because we need more coordination between the processes and we need to update the copies often.
- Weak guarantees are cheaper.

Types of guarantee

We can divide guarantees in three main categories

- **Guarantees on content**. Guarantees on content ensure that a local copy doesn't have to be different to the main copy for some number of values. In other words this guarantee fixes a maximum difference between two copies of data (for instance the number of words in a text document).
- **Guarantees on staleness**. Guarantees on staleness ensure that the data is updated in a fixed interval of time. In this case the guarantee is on the update time, not on the data. Such guarantees are used in sensors systems where we ask the system an update at regular intervals.
- **Guarantees on the order of the updates**. Guarantees on the order of the updates ensure synchronisation across replicas to guarantee the write-write or the read-write order. In particular we can distinguish between two models that guarantee the order of updates
  - The **data-centric** model in which the operations are executed on the replicas and the focus is on what is written in the datastore.
  - The **client-centric** model in which clients can move between various replicas.

The choice between these models depends on the assumption that we do.

22.2.2 Types of protocols

We can use different types of protocols depending on the assumption that we do. In particular we can identify

- Single-leader, multi-leader or leaderless protocol.

- Synchronous or asynchronous protocols.
- Protocols for sticky clients (i.e. that use just one replica) or mobile clients (that move and use multiple replicas).

High availability A client that can continue to work on a datastore even if it's disconnected to the network is high available. In other words high availability ensures that the client doesn't require blocking communication (and synchronisation) to keep working.

If a protocol ensures high availability then the model used is highly available. High availability is costly.

Backup

Backup protocols are used only to ensure fault tolerance. In particular

- All operations are executed on the main file.
- Periodically the main file is copied (replicated) by the main to build a backup copy. All copies are handled by the main.
- If the main copy fails it is replaced with the last backup copy.

Backup protocols can be

- **Synchronous**. Data is safe only after all the copies have acknowledged to the main a successful backup. In this case execution is interrupted until all copies have acknowledged.
- **Asynchronous**. Processes only wait for save storage on the main without acknowledgements from the copies. In this case fault tolerance is not provided.

Single-leader protocols

In single-leader protocols we can distinguish

- A node called **leader** that has the main data.
- All the other nodes called **followers** that have a copy of the main data.

In single-leader protocols the leader coordinates writes, reads and copies. In particular

- Processes can write or read on the main node (the leader). The leader has to order the write and read requests of the clients. The leader also has to send the copy of the main data to the followers.
- Processes can also read from the followers. In this case a copy of the main is read.

Advantages This type of protocol

- Is easy to implement.
- Works well in networks with a not so big delay.
- Works well in systems in which multiple reads are required.
- Avoids write-write conflict, in fact the leader can order the writes.

Disadvantages Having a single leader means that there is a single point of failure in the system. Even if write-write conflict are avoided, we can still have read-write conflict because processes can also read on followers that might not have the last copy of the main data.

Synchronicity Single-leader protocols can be

- **Synchronous**. In this case the write operation is over when all the copies have acknowledged.
- **Asynchronous**. In this case the write operation is over when the result is stored on the leader (i.e. no acknowledgement is required).
- **Semi-synchronous**. In this case the leader needs at least k acknowledgements to end a write operation. This solution is a trade-off between synchronous and asynchronous solutions.

Multi-leader protocols

In multi-leader protocols there are multiple leaders that work in parallel. In particular

- Writes can be executed on one of the leaders.
- Reads can be executed on every node (both leaders and followers).

Conflicts Having multiple leaders we can't ensure that writes are executed in a determined order, in fact different writes are executed in parallel. This behaviour can generate write-write conflict that in some cases can be solved merging the data (for instance git can merge two copies of a file in many cases). Merging isn't always possible, in fact in some cases a human is required to solve conflict (like in git when the same line is modified by two developers).

Usage This solution is very efficient when the process use more frequently some data that is not used by other processes. For instance in social networks a person usually has friends that are geographically close. Consider two leaders: one in Europe and one in the US. A person from Italy would like to have fast access with updated data from people from Europe while he doesn't case about people of the US which data is handled by the US server and is not always updated in the EU server.

Leaderless protocols

In leaderless protocols

- Clients are connected to multiple replicas
- Writes can be done to several nodes. In particular a write or read request is sent to multiple nodes and all the nodes have to select the majority to elect the correct result of the operation.

Chapter 23

Data centric consistency models

In this chapter we will analyse the main data-centric consistency model and how the various types of protocols (single, multi-leader and leaderless) can be applied.

23.1 Sequential consistency

A schedule is sequential consistent if it's 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.

Order for the processes When creating a global sequence the operations of the single processes can be interleaved but can't be inverted. In particular we have to find a sequence of operation (interleaving the operations of the single processes) that every process agrees on.

For instance the sequence 23.2 is inconsistent because we can't find a sequence so that P_3 reads $x = b$ and than reads $x = a$ while P_4 reads $x = a$ before reading $x = b$.

Time Sequential consistency doesn't rely on time.

Number of sequences There exists multiple sequences that satisfy sequential consistency, in fact operations can be interleaved in multiple ways to create correct sequences (i.e. sequences everyone agrees on).

Examples In example 23.1 we can define a global sequence that respects the order of all the operations of the single processes. For instance the schedule

$$W(x)b, R(x)b, R(x)b, R(x)b, W(x)a, R(x)a, R(x)a$$

P1:	W(x)a				
P2:		W(x)b			
P3:			R(x)b		R(x)a
P4:				R(x)b	R(x)a

Table 23.1: A consistent sequence.

P1:	W(x)a				
P2:		W(x)b			
P3:			R(x)b		R(x)a
P4:				R(x)a	R(x)b

Table 23.2: An inconsistent sequence.

is a global sequential consistent sequence. Notice that the sequence doesn't respect the time order in which the various requests arrived.

23.1.1 Single-leader implementation

Sequential consistency can be implemented using single-leader protocols only under some specific assumptions. In particular

- The communication channel has to be FIFO.
- We must have synchronous communication.

Implementation As for all single-leader protocols, all the processes have to send the operations to the coordinator that orders them to create a sequence everyone agrees on.

23.1.2 Leaderless implementation

Leaderless implementations are based on a quorum system. This means that the majority of nodes have to agree on a specific sequence to have an agreement. In particular

1. A process sends its operations to multiple nodes.
2. An operation is executed only if a number of servers agrees on a version.

Reading and writing quorum Reading requires a quorum that allows to read the last version of the main data. In particular if we call NR the number of nodes to which a process has to send a read operation and NW the number of nodes to which a write operation has to be sent, then we have that

- The sum of NR and NW as to be bigger than the total number of nodes N

$$NR + NW > N$$

  to avoid read-write conflicts. In fact if this sum is bigger than the number of nodes then at least one process has seen the last write, so its data is the most current.

- The number NW has to be bigger than the half of the number of nodes N

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

  to avoid write-write conflicts.

A commonly used solution (that satisfy both properties) for frequent reads and sporadic writes is ROWA (Read One Write All).

P1:	W(x)a		
P2:		R(x)b	R(x)a

Table 23.3: A non linearisable sequence.

23.2 Linearisability

In linearisability each operation should appear to take effect instantaneously at some moment between its start and its completion.

In other words this model is similar to the sequential consistency model but each operation is seen in a certain point in global time. This means that the linear model is stronger then the sequential model, in fact processes have to agree on a single instant in which an operation happens.

In other words operation can't be interleaved freely but only ensuring that an operation is executed in a specific instant between when it has started and ended. For instance in example 23.3 we can't find a linearisable schedule because operation $R(x)b$ can't be moved before $W(x)a$ because it has to be executed between the its start and end.

Linearisable schedule If the operations on single processes are linearisable then the global schedule is also linearisable.

23.2.1 Single-leader implementation

Linearisability can be achieved using a single-leader protocol in which the leader orders the writes according to their timestamps.

The clients can read from

- The leader (always).
- The followers if they can be updated synchronously and atomically.

23.3 Causal consistency

In causal consistency only writes that are potentially causally related must be seen by all processes in the same order, otherwise concurrent writes may be seen in any order at different machines.

Strength This model is weaker than the sequential consistency model, in fact ordering is ensured only for operations that are in a causal relationship.

Causal relation A write operation is causally related to another operation if a process sees the effects of the first operation before executing the write. For instance in example 23.4 the operation $W(x)b$ could be causally related to $R(x)a$ because $P2$ might have written $x = b$ after reading $x = a$.

P1:	W(x)a		
P2:		R(x)a	W(x)b

Table 23.4: A causally related sequence.

More formally we can define causal relation as follows

- A write operation W by a process P is causally related to (i.e. is causally ordered after) all the previous operations on the same process (even considering different variables). This means that the operations on the same process always have to be seen in the same order.
- A read operation R on a variable x by a process P is causally related to (i.e. causally ordered after) all previous writes by P on the same variable.

Properties Causal order

- Is **transitive**.
- Is **not a total order**.

23.3.1 Vector clocks implementation

Implementation To ensure causal consistency we can use vector clocks. In particular

- The vector clock of a process P contains the timestamp of the latest operation that P has seen from all the other processes.
- Before showing a value to the application layer we have to check all causally related values. In other words when we receive a write operation from the network, it contains the vector clock of the sender. Before showing executing the operation we have to check if we have the last update, i.e. we have to check if all the cells in our vector are bigger than the cells of the vector of the sender. In other words if the vector of P is greater then the sender's vector, then P's state is the same as the sender's.
- If the vector of P isn't greater, then the incoming operation has to be putted in a pending list because we have to wait other operations so that our state can be the same as the sender's.

In a nutshell a write operation W by S is executed on R only if R has received all previous writes (causally related to W) of S.

High available Implementations of protocols that ensure causal consistency are highly available, this means that a client can read or write a local replica while being disconnected from the other node.

In particular we can ensure high availability because when we are disconnected from the other nodes, everything we do doesn't have a causal relation with the operations of the other nodes so our operations can be done concurrently. In fact what we do isn't causally related to the other nodes' operations because if we don't know such operations, it's not possible that we did some operation as a consequence of the others did.

For instance consider an SMS chat between two people Alice and Bob. Consider that A writes *How are you?* to B and that B can't receive the message because it's disconnected from the network. After some seconds B writes *Hi, do you want to come over tonight?* to A (and of course the message is not delivered). In this example there is no way that B answered to A because he never received Alice's message. In other words Bob's message isn't caused by Alice's question so the two messages can be sent concurrently.

Assumptions This protocols works and ensures high availability only if the client is sticky, i.e. it accesses the data from the same replica.

Thus we can say that causal consistency is **stick high available**.

Practical examples

Since I usually have an hard time remembering how to solve causal consistency exercises, here is a small guide. For each process separately

- Consider all dependencies between write operations and write down the dependencies.
- Consider all dependencies between write operations and previous read operations.
- Check if the other processes read the values in the same order.

23.4 FIFO consistency

FIFO consistency ensures that

- 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.

In other words a process P sees the write operations of process P_1 in the order P_1 issued them. On the other hand if we consider processes P_1, P_2 and P_3, their writes can be seen in any order by P (i.e. we can have W_1, W_2, W_3 or W_3, W_1, W_2, we only care about the ordering of writes of the single process). Namely there is no specified order between write operations of different processes.

Strength FIFO consistency is strictly weaker than causal consistency, in fact it guarantees less. In other words if a schedule is causally consistent than it is also FIFO consistent but not vice versa.

High availability Because causal consistency was high available than also FIFO consistency is high available.

Implementation FIFO consistency can be implemented using scalar clocks so that the write operations coming from a process contain the clock and can be ordered. In particular, a replica performs an update u from P with sequence number S only after receiving all the updates from P with sequence number lower than S.

23.5 Eventual consistency

In data-centric consistency models we have analysed systems in which simultaneous writes and reads can happen. This doesn't happen in every system, in fact in some cases writes are not simultaneous, conflict can be easily resolved or clients only have to read data.

Eventual consistency In such systems we don't guarantee a certain order but we ensure that all replica will eventually (that's why is called eventual consistency) receive all messages.

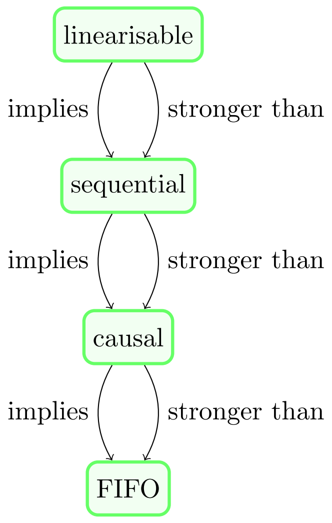


Figure 23.1: A summary of data-centric consistency models' strengths.

23.5.1 Strong eventual consistency

In some cases if we ensure eventual consistency we might end up with divergent states on different processes. We can avoid this problem by adding convergence to eventual consistency. A model that guarantees both properties is called strong eventual consistency.

This means that every process receives all updates and when no more writes are send, every process will eventually end up (converge) in the same state. The common final state is reached independently from the order in which updates are received.

Conflict-free replicated data types To obtain such result we typically use commutative operations so that they can be applied in any order. Commutative operations are obtained using Conflict-free Replicated Data Types (CRDTs). CRDTs are data types whose operations are commutative.

Chapter 24

Client-centric consistency models

In client-centric consistency models we shift the focus from the data and the order in which updates have to be executed to the clients.

Mobile clients Clients are relevant when they are mobile (i.e. not sticky), that means when clients can access data from different replicas.

We can distinguish four properties that define what a client can perceive in terms of write-write, read-read, write-reads, read-writes

- **Monotonic reads**.
- **Monotonic writes**.
- **Read your writes**.
- **Writes follow reads**.

24.1 Implementation

24.1.1 Client state

Client-centric consistency models can be implemented saving the state of the client. In particular clients

- Mark all operations with an unique id.
- Save the ids of the commands that they have executed.
- Use the ids when they move to different replicas. In particular if the replica doesn't have the update with the last id of the client, the operations on the new replica have to be putted in hold.

Write and read sets A client has two sets

- A read set that saves the id of the last read operation on each variable.
- A write set that saves the id of the last write operation on each variable.

24.1.2 Monotonic reads

Monotonic reads allow the client to see reads in order. In other words if a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more recent value.

To put in another way once the process reads a value from a replica, in the future it will never read an older value.

Implementation In monotonic reads before reading on a replica $L2$,the client checks that all the writes in the read-set have been performed on $L2$.

This ensures that what we have read before is already in $L2$.

24.1.3 Monotonic writes

Monotonic writes guarantee that a write operation by a process on a data item x is completed before any successive write operation on x by the same process.

In other words a client always have to overwrite the previous data.

Implementation Monotonic writes works like monotonic reads but the write-set is replaced with the read-set because we want to check that all previous writes are already in the new replica.

24.1.4 Read your writes

Read your writes guarantees that the effect of a write operation by a process on a data item x will always be seen by a successive read operation on x by the same process.

In other words when a client writes something and then it reads it, then the client has to read at least what it has written.

Implementation Read your writes behaves like monotonic writes.

24.1.5 Writes follow reads

Writes follow reads guarantees that a write operation by a process on a data item x following a previous read operation on x by the same process is guaranteed to take place on the same or more recent value of x that was read.

Implementation In write follow reads

1. The state of the server is brought up-to-date using the read-set.
2. The write is added to the write-set.

Chapter 25

Design strategies

25.1 Replica placement

When designing distributed systems with replicas we have to decide where to put such replicas. In particular we can have

- **Permanent replicas** that are mainly used in distributed databases. In this case an administrator decides on which machines are located the nodes.
- **Server-initiated replicas** that are mainly used in cloud services. In this case the user buys a server and the server decides which resources it needs to provide the service required by the user. In other words the server initiate some replicas and dynamically enlarges or shrinks depending on the load.
- **Client-initiated replicas** like Web caches. In this case the user decides to keep some data locally for a given time.

25.2 Update propagation

Another thing to keep in mind is what data we want to propagate and how we want to do it.

What to update In particular, regarding what to propagate we can

- Perform the update and propagate only a **notification**. In this case we notify the replica that a variable has a new value so that the replica knows that the local cache is invalid. This technique is used in conjunction with invalidation protocols to avoid unnecessarily propagating subsequent writes (the data is actually retrieved only when the user asks for a read). This solution has small communication overheads and works best if the number of reads much smaller than the number of writes.
- **Transfer the modified data** to all copies. This method works best if the number of reads is much bigger of the number of writes.
- **Propagate information** to enable the update operation to occur at the other copies (also called active replication because the replicas have to perform some operations). This method

needs a very small communication overhead, but may require unnecessary processing power if the update operation is complex, so it's useful to pass a big amount of data unless the computation power to execute the operations is too high.

How to update Regarding how to update

- **Push-based approach** in which the main data is pushed to the cache.
- **Pull-based approach** in which the cache pulls the update.

Leases can be used to switch between the two modes.

25.3 Propagation strategies

- Leader-based protocols. Propagation can be synchronous, asynchronous, or semi-synchronous.
- Leaderless protocols.
  - Read repair. When a client makes a read from several replicas in parallel, it can detect stale responses from some replica. The client (or a coordinator on its behalf) updates the state replica.
  - Anti-entropy process. Background process that constantly checks for differences among replicas and copies missing data. Interaction between replicas can be push or pull, and different strategies are possible.

25.4 Replication and consistency in databases

Distributed databases usually replicate some data, thus consistency is a key aspect. Real-world database usually replicate data only partially. This means that the data is partitioned and each partition is replicated across different nodes but not all nodes have all the partition. In other words each node gets some (or sometimes all) partitions of the data. For instance say data D is partitioned in D_1, D_2 and D_3. D_1 can be replicated on node N_1 and N_2, D_2 on all nodes and D_3 on node N_1 and N_3. As we can see

- Node N_1 has D_1, D_2 and D_3 (all the partitions).
- Node N_2 has only D_1 and D_2.
- Node N_3 has only D_2 and D_3.

Part VIII

Peer to peer

Chapter 26

Introduction

Peer to peer (P2P) is an architectural paradigm based on

- The absence of a centralised server.
- A network of nodes called **peers**.
- The resources of the nodes at the edge of the network (i.e. the clients in a client-server architecture).

Peers A peer-to-peer network is based on peers, i.e. nodes that doesn't have a fixed role but can act both as server or client. Peers are

- **Independent** so there is no central admin.
- **Dynamic**, in fact they can join and leave the network dynamically and unexpectedly.
- **High variable resources**.

Resources The P2P paradigm take advantage of some resources at the edge of the network (i.e. of those nodes that are clients in a client-server architecture). In particular such resources are

- **Bandwidth** (packets forwarding). For instance in sensors networks every node can forward packets coming from other sensors.
- **Computation power**, for instance in block-chain.
- **Storage** or **memory**.

Initially peer-to-peer architectures where used to share storage, in particular many file sharing applications like Napster and BitTorrent used this architectural paradigm to store files on the clients and allow clients to share them directly without contacting a central server.

Edge computing We can use the resources at the edge of the network because end devices

- Have became more powerful.
- Have access to broadband connection.

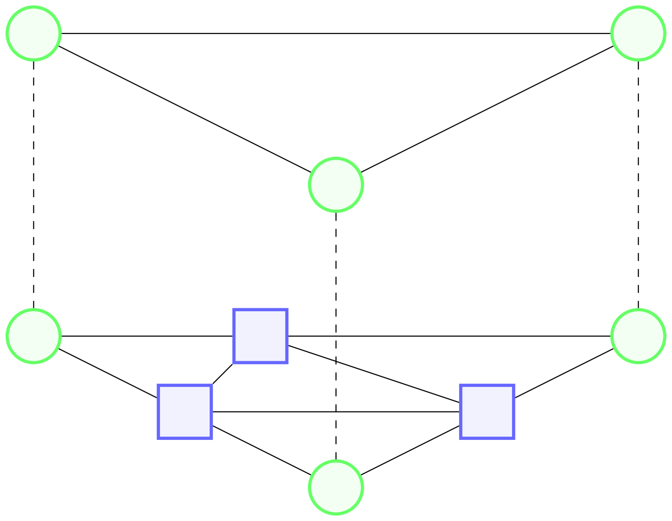


Figure 26.1: An overlay network (green) build on top of a physical network(blue).

26.1 Differences with client-server architecture

Scaling The client-server architecture is hard to scale in fact when new clients are added we have to add more resources (i.e. more servers) to handle the requests of the clients. Peer-to-peer architecture solves this problem, in fact when new nodes are added to the network, such nodes can be both clients and server thus we don't have to add new resources.

Peer-to-peer networks can be very big, in fact they can be internet-wide.

Single point of failure Client-server architecture presents a single point of failure and require a single admin to handle the resources. We don't have the same problem in P2P architectures, in fact every node has a different admin and if a peer fails the others can still offer the same service.

Unused resources Client-server architecture leaves some resources unused, in fact clients usually have an high processing power and can access broadband connectivity but are only used to send requests to clients without doing big computations or storing any data. The same doesn't apply for peer-to-peer networks, in fact end devices can be peers and offer their computational power and storage.

26.2 Terminology

Overlay network A peer-to-peer network is defined as an overlay network, that is a virtual network that connects peers without considering the real physical underlying network. In other words an overlay network establish a virtual link between any two peers.

Geographic proximity An overlay network may be very different from the physical network, in fact virtual networks many not consider geographical proximity. In other words adjacent nodes in a virtual network might be geographically far (i.e. many hops away) one from the other.

26.3 Goal

Peer-to-peer networks have two main goals

- **Finding a resource**. For instance in applications that use P2P to share files we need to find the peers that have the file that we are looking for.
- **Download the resource**, in case of file sharing applications.

Finding a resource Finding a resource can be divided in two different problems

- **Search a resource**, i.e. obtaining all resources that satisfy a string (like when we search something on Google). In other words when a peer searches a resource it wants all the resources that satisfy a property. Searching a resource can be harder than locating it.
- **Lookup a resource**, i.e. obtain the address of the peer that has the resource with a specified identifier.

Downloading a resource Regarding resource downloading we can

- Directly download the resource from the source peer.
- Get the location from where the data can be retrieved.

Chapter 27

Resource search

27.1 Centralised search - Napster

Centralised search solves the search problem. To describe this architecture we'll use a file sharing application.

Structure In centralised-search files are distributed in the peers of the network and use a centralised server to search the files in the clients.

Given this description we notice that

- Searching is not done is a peer-to-peer way because it's achieved using a central server.
- Peer-to-peer is used to store and distribute files. In other words the resources shared in peer-to-peer are storage (to store files) and bandwidth (to share files).

27.1.1 Actions

Join When a node wants to join the P2P network they have to contact the central server that adds them to the network.

Publish When a node wants to publish a file (i.e. make it available to other peers) it has to tell the central server the identifier of the file and some metadata that helps the central server to search a file.

Search The central server has a table (built using the publications of the peers) that associate the id of a resource to the address of the peer that has such resource. When a peer searches for a resource, the server finds the id of such resource and returns the address corresponding to such resource in the table.

Fetch The data is fetched directly from a peer, the central server doesn't intervene in the process.

27.1.2 Advantages and disadvantages

Advantages The advantages of this solution are

- It's very simple.
- Has a constant search scope, in fact every peer only has to contact the server to search a resource.

Disadvantages On the other hand

- The server maintenance has a linear cost with the number of states N, in fact the central server has to handle all the nodes.
- The central server represents a single point of failure, thus if the server fail there's no way to search a file.
- The central server represents a single point of control.

27.2 Query flooding - Gnutella

Query flooding takes the opposite approach with respect to centralised search with respect to search strategies. In particular all peers query and forward the query to the adjacent nodes in the overlay network.

27.2.1 Actions

Join When a peer wants to join the network, it has to connect to a random node of the network. If the new peer doesn't know any node of network it can use a big public list of anchor nodes.

When a peer contacts an anchor node (both previously known or obtained from the public list), the anchor node sends a `PING` message to the other peers of the network to communicate that a new node has joined the network. Some nodes can answer with a `PONG` message to say that the new node can connect. The new peer connects to all the other peers that have answered with a `PONG` message. This way if a peer to which the new node is connected fails, the new node is still connected to the network.

Publish A peer doesn't have to publish the identifier, in fact the files are kept locally.

Search When a node needs to search a file, it simply queries its neighbours. When a peer R receives a query from a peer Q, it searches in its local storage and

- If it has the searched file it answers immediately the query.
- Otherwise it forwards the request to the other neighbours and waits for a response from one of them. When the response arrives, the peer answers to Q.

Fetch Files are downloaded directly from the peer that has the resource.

27.2.2 Hop limit

Query flooding generates a lot of requests, in fact every peer may have to forward a query to all its neighbours. This behaviour may slow down the network. To reduce network congestion we can introduce a limit to the number of hops a query can do. In particular

1. A query is initialised with a maximum number of hops D.
2. Every time a peer receives a request, it decrements the maximum number of hops in the request and
   - If the new number of hops is 0, it doesn't forward the query.
   - Otherwise the query is forwarded to all the adjacent nodes.

27.2.3 Advantages and disadvantages

Advantages Query flooding has many advantages, in fact the protocol

- Is **fully non centralised**, in fact the search effort is distributed on all nodes. Flooding query is very robust because there isn't a single point of failure.
- **Distributes the search cost** and reduces the search cost on a single node, in fact every single node has to search a file in a small database (i.e. its local database).

Disadvantages Query flooding also has some weak spots, in particular

- It generates a **lot of traffic** (which is expensive), in fact every peer has to forward a query to all its neighbours.
- It has a **linear** (with the number of nodes N) **search scope**, in fact in general we could need to search all nodes before finding a file.

$$\mathcal{O}(N)$$

- It has a search time that scales up with the maximum number of hops D (and because of the hop limit it's not guaranteed that the file is found).

$$\mathcal{O}(2D)$$

27.3 Hierarchical topology - Kazaa

An intermediate approach between centralised search and query flooding is to organise the peers in a hierarchy.

Structure In particular peers are divided in

- **Normal nodes** that are normal peers with no special behaviour.
- **Super nodes** that are special peers that use query flooding between each other. Super nodes differ from normal nodes only for the fact that they communicate using flooding, but they are still end-clients.

Normal nodes communicate with a super node like if it was a centralised server but the super node uses flooding to communicate with other super nodes.

27.3.1 Actions

Join When a peer wants to join the network it has to ask a super node. After joining a normal node might be promoted to super node if it is in a good condition (e.g. has a good bandwidth).

Publish When a normal node wants to publish a file, it simply has to contact a super node and communicate that it has a file. In particular the normal node only sends the couple `<file_id , normal_node_address>`, in fact the super node only stores the addresses of the normal nodes and the ids of the resources they hold. This happens because the resources are fetched directly from the peers.

In other words a super node has

- The ids of its files.
- Its files.
- The ids of the files stored in some other normal nodes (but not the actual files).

A super node might also propagate the ids to other super nodes.

Search When a normal node wants to search a file, it asks a super node. The super node uses query flooding to search the file on the other super nodes.

If a super node is searching a file, it simply uses query flooding.

Fetch Files are downloaded directly from the peer that has the file (either a super node or a normal node). In this case a file might also be fetched from multiple peers.

27.3.2 Advantages and disadvantages

Advantages The main advantage of this search protocol is that it's a trade-off between centralised search and query flooding. In particular it exploits the non-centralisation of query flooding without generating too many messages (because the flooding is used only by some nodes).

Disadvantages This solution have disadvantages, too. In particular

- There's no guarantee on search scope and time.
- It's not possible to build a structured network between super nodes because they are normal peers that can fail unexpectedly.

Chapter 28

Download protocols

28.1 Collaborative systems - BitTorrent

BitTorrent is a protocol meant only for better downloading, in fact it doesn't solve the search problem. In particular we have to use an external protocol to search the peers that have the resource. We say that the search problem is solved out-of-band.

28.1.1 Terminology

Torrent A Torrent (`.dot`) is a meta-data file that describes the file to share. In particular the file contains

- The name of the file.
- The size of the file.
- The checksum of all blocks (sha-1 hash of each chunk).
- The address of the tracker.
- The address of the peers that have such file.

Seed A seed is a peer that has the complete file and still offers it for upload (even if it ended downloading it).

Leech A leech is a peer that has an incomplete download.

Swarm The swarm is the set of all seeds and leeches.

Tracker A central server that keeps track of leeches and seeds. When a peer wants to download a file, it should ask the tracker the list of peers and start downloading from them. The address of the tracker is in the Torrent file that has to be retrieved out-of-band (e.g. on a web page).

28.1.2 Chunk download

In precedent protocols a file was fetched directly from one peer. On the other hand in collaborative systems the file to download is divided in chunks and a peer can download different chunks from different peers.

Contribution This behaviour allows a peer to contribute immediately to the system, in fact as soon as a node receives a chuck of a file, it can immediately make it available for download so that other peers can fetch such chunk. Furthermore when a node is downloading, it can still use a part of the bandwidth to upload the chunks it has already downloaded.

File chunks At a certain time it's possible that no peer has all the chunks of a file but sill all peers can eventually obtain the full file. This is possible because peers can exchange chunks even if they don't have the full file. This can happen only if there is at least one distributed copy (i.e. all the chunks are somewhere in the network).

Chunk order Chunks don't have to be downloaded in order, in fact every peer can reorder the chunks when the download is complete.

Chunks priority When a peer uploads a file, it prioritises the chunks that are rare. This way such chunks become less rare and more peers can obtain it. This solution makes the protocol more robust.

28.1.3 Exploiting good connections

Collaborative systems try to exploit good connections to remove free riders (i.e. peers that only download and can't upload any chunk), in fact the protocol prioritises the upload to nodes that offer a good download service. In other words if a node A sees that it can download from B with a good speed, then A prioritises B for upload.

To put it in another way the more A downloads from B, the higher chances are that A uploads to B.

Slots Each peer has a number of slots. Each slot is assigned to a peer. A peer uploads chunks to the peers in its slot.

Slots are rotated and offered to peers that offer a good downloading service. This allows peers to exploit all connection at their best, improve download speed and improve the topology of the overlay network.

28.1.4 Advantages and disadvantages

Advantages The main advantages of this protocol are

- The protocol works well in practice.
- The protocol gives incentive to share resources and eliminates free riders.

Disadvantages Even if it's very popular, we can still find some disadvantages in this protocol, in fact

- It still relies on some central server (the tracker) that represents a single point of failure.
- Pareto efficiency is a weak condition. Pareto efficiency is a game theory's concept and in this protocol it's approximated by the fact that if two clients get poor download rates for uploads, they can start uploading to each other to improve download speed.

Chapter 29

Distributed data storage

29.1 Secure storage - Freenet

Secure storage allows to store files securely and anonymously in a peer-to-peer network. In this case the resources offered by peers are storage and bandwidth.

Anonymity The main goal of secure storage is to store and exchange files keeping anonymous the publisher and all the readers of the file. This way it's possible to avoid censorship.

29.1.1 Node

Storage Every node on the network has to participate to the system with some storage space. The files stored on peer's storage are encrypted so that a peer can't discover what's saved in its storage.

File removal If a file is not used for a long time, it gets removed from the local storage. In this sense secure storage protocols behave like caches.

File propagation On the other hand if the file is popular, it gets copied and propagated to other nodes so that it can be downloaded at a faster rate from many peers.

29.1.2 URI

Entities in the network Secure storage protocols move both queries and files.

Resource identifier Some secure storage protocols offer Uniform Resource Identifiers to locate files over the network.

29.1.3 Routing table

Each peer has a routing table that associate a resource (the id of the resource) to a peer of the network (in particular to a neighbour peer). If an id is associated with a peer P, it doesn't mean that P has the resource, it only means that we have to contact P to get the resource (it might have the file or it might forward the request using its routing table). This means that a peer only has a local view of the network (it only speaks to its neighbours; even if the file is far away in the network it only know which neighbour to contact).

Populating an empty routing table Initially the routing table of a node is empty. When a node wants a file it sends a request to one of its neighbours (chosen at random) and waits for the response. If the peer (say P) eventually returns the file F than a new entry $< id_F, P >$ is added to the table.

Populating an non empty routing table If the routing table has some entries, when a user searches for a new file (i.e. the id isn't in the table) it searches it in the peer that has the closest id to the id of the file to search. For instance if

- Peer A has (i.e. we contact it to retrieve the file) files with id 1 and 4
- Peer B has files with id 5, 6 and 0.

Then a request for file with $id = 2$ is routed to A because it has the closest id ($id = 1$ is the closest to $id = 2$).

This procedure is applied because a file is published to the node that has the file with id closer to the id of the file to publish.

Id The id of each file is generated using the secure hash of the content of the file itself.

29.1.4 Search

The search procedure is made of the following steps

1. A peer sends a file request to one the correct peer using its routing table (or at random if the table is empty).
2. The requested is forwarded by the peers using their routing table until one peer finds the file in its local storage.
3. The resource travels back on the same path used to forward the request.
4. At each node of the path the file has a probability to be saved on the local storage of the node. This way the file is brought closer to the peer that requested it so that if the file is requested again it can be retrieved faster (because it's closer).

29.1.5 Actions

Join When a peer wants to join the network it has to contact a some other nodes (like in query flooding).

Publish When a peer publishes a new file, it is routed (i.e. forwarded from peer to peer using the routing table of each peer) to the node that has the id closest to the id of the file published.

Search When a peer wants to search a file it queries the node using the routing table. A peer search a file using its id. The next hop is

- The peer with the specified id if the file's id is in the table.
- The node with the closest id if the specified id is not in the table.
- A random node if the table is empty.

29.1.6 Advantages and disadvantages

Advantages The main advantages of secure storage protocols are

- The routing mechanism is very intelligent.
- The search scope is very small.
- Such protocols ensure anonymity.

Disadvantages Secure storage protocols also come with some disadvantages, in particular

- It's expensive to guarantee anonymity.
- The protocol is still probabilistic, in fact we don't know for sure how much a request needs to reach the node that has the resource. This happens because resources can be copied in a node in a probabilistic way.

Chapter 30

Computational peer-to-peer

30.1 Blockchain - Bitcoin

A blockchain is a consensus mechanism that doesn't require a centralised authority to establish consensus. Blockchain doesn't even require a central server to connect new nodes.

Formal definition Formally a blockchain is a Byzantine fault-tolerant permissionless consensus protocol.

Resources In blockchains the peers share computational power.

Structure

Distributed log A blockchain is a distributed ledger (i.e. a log) in which every peer agrees on the order in which operations are added to the ledger.

Every peer has a copy of the ledger and when a new log is added, the new ledger is propagated to the peers so that they can update their local copy.

Adding a transaction

When a peer wants to add a log to the ledger a complex mathematical problem has to be solved (not necessarily by the peer that wants to add the log) before adding the log. The problem consists of finding the original string given an secure hash digest. In particular the original string is the log that contains the transaction.

Solving the problem is also referred as mining. When a peer solves the problem it gets rewarded with a bitcoin.

Properties Solving the problem is very difficult and requires a lot of computational power but it's easy to verify if the solution is correct. In particular given an hash digest it's hard to find the original string (i.e. the solution) but, if we know the string it's easy to calculate the hash, in fact we can calculate the hash of the solution and if it corresponds with the original hash we are sure that the result is correct.

Updating the ledger Before mining transactions are saved in a queue of pending transactions. When a peer solves the problem, it adds the log to its ledger (solving the problem means finding the actual transaction) and sends its ledger to other peers. The other nodes confront their log with the new ledger and if the added logs are valid (can be verified easily) then the old ledger is replaced with the new one.

In particular higher ledgers (i.e. with more logs) always replace lower ones.

Illegal update

Mining has to be hard because otherwise a peer could send the same money to two different people and still get accepted by all the peers. This happens only if a peer can mine two transaction very fast.

This situation never shows because

- Mining take time.
- Longer ledgers are stronger.

In fact if two transactions are found at the same time only the higher ledger survives.

30.1.1 Bitcoin

The Bitcoin software (the currency takes its name from the software) is based on a blockchain. In particular the transactions (i.e. person A sends 2 bitcoins to person B) are stored in the ledger (in principle one log per transaction, in reality more transaction in a single log to improve performance) and every peer has to agree on the validity of the transaction and on the order in which the transactions are added.

Balance Since all transactions added to the blockchain are verified (i.e. everyone agrees on the fact that they are legit), to obtain the balance of one wallet (i.e. the account of one person) we simply have to sum all the operations done by the owner of the wallet. For instance if in the ledger we find that A receive 2 bitcoins and receive one bitcoin then we know that the balance of A is 1.

Chapter 31

Distributed Hash Tables

Searching can be simplified if we can give a structure to the peers in the overlay network (i.e. the virtual network, not the physical one). In particular we can use Hash Tables, i.e. a data structures that associate a key to a value. In a peer-to-peer structure we use a distributed hash table, i.e. an hash table whose keys are distributed between multiple nodes (peers).

In a DHT we associate ids to keys and nodes and then we have to find a way to map keys (their id) to nodes (their id).

Operations The operations that can be done on an hash table are

- `put(key, value)` to add the key `key` and map it to the value `value`.
- `get(key)` to get the value of the key `key`.

31.1 Actions

Join To join the network, peers have to contact a "bootstrap" node and integrate into the distributed data structure. The new peer also has to generate its node id.

Publish When a peer wants to publish a file it routes the publication for file id toward a close node id along the data structure.

Search When a peers wants to search a file, it has to route a query for file id toward a close node id. The data structure guarantees that query will meet the publication.

Fetch When a peer wants to download a file

- If the publication contains actual file then the peer can fetch it from where the query stops.
- If the publication contains a reference then the peer can fetch the file from the location indicated in the reference.

31.2 Chord

Structure In a chord structure peers are organised in a ring.

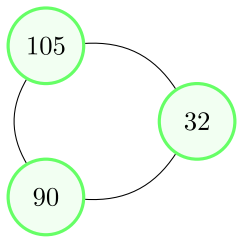


Figure 31.1: Chord Distributed Hash Table.

Identifier In a chord topology

- Every node has an id (e.g. $N123$). The id usually is the hash of the IP address of the node.
- Every resource has a key (i.e. an identifier) that usually is the hash of the content of the resource. The identifier is used when a peer looks up a resource. A resource usually contains the address of the node from which it's possible to download a file. This is done so that when nodes join or leave only references have to move, not entire files (which would be more expensive).

Keys and node ids have the same length m, thus the keys and ids belong to the same keyspace.

Relation between nodes and resources Each node is responsible for a resource, in other words we can define an association between a node id and a resource key. In particular a file with key k is managed by the node with the smallest id i that is bigger than k

$$\text{resource } k \text{ managed by node } i \iff \min i : i \geq k$$

For instance in figure 31.1

- Node 90 manages resources with key between 33 and 90.
- Node 105 manages resources with key between 91 and 105.
- Node 32 manages resources with key between 106 and 32.

31.2.1 Single pointer implementation

The chord structure can be implemented using a single pointer for each peer. In particular each peer has a pointer to the next peer in the ring. The routing table of each node is therefor made of just one entry (i.e. the next hop).

This structure is very simple but we need $n/2$ moves on average to reach a node, in fact

- In the worst case scenario we the destination is the node just before the sender, thus we need $n-1$ hops to reach the destination.
- In the best case scenario we the destination is in the node just after the sender, thus we need only 1 hop to reach the destination.

Lookup When a peer want to look up a file it sends a request containing the key of the file to its successor.

When a peer receives a request

- If the key is managed by the peer it replies directly to the sender.
- Otherwise it forwards the request to the next peer in the chord.

31.2.2 Full routing table

At the opposite side of the single pointer implementation we have the full routing table implementation. In this case every peer has a pointer to every other peer. This implementation allows to reach the node that manages a resource with just one hop but the routing tables become very big.

31.2.3 Finger table

The finger table implementation is a trade-off between the simplicity of single pointer's routing table and the speed of full table implementation.

Principles of working A finger table allows to halve the search space for each iteration. In particular every peer has m entries (m being the length in bit of the keys and the ids) and every entry i in the finger table of node n is the first node whose id is higher or equal than $n + 2^i$ In other words, the i-th finger points $\frac{1}{2m-i}$ way around the ring.

Routing table

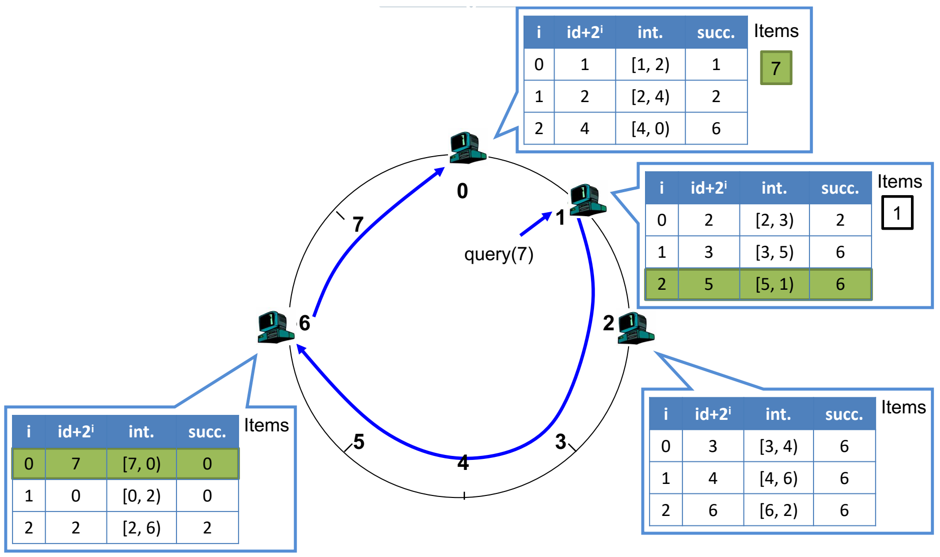
To build the routing table of node n we keep an identifier for every bit of the identifiers For instance if the network can have at most 8 peers, than the length of the key must be $m = 3$ bits. In this case the routing table of each peer has 3 entries. Practically to build the table of n we have to do the following operations (all sum operations have to be considered as modulo 2^m)

1. Consider $i = 0, ..., m - 1$.
2. For every i compute $id + 2^i$ where id is the identifier of the node n. The number 2^i indicates how far away the reference points. For instance if $id = 1$ and $i = 2$ then the table entry related to $i = 2$ points $2^2 = 4$ positions after node n, i.e. points to the node with identifier $id + 4$.
3. If the node with identifier $id + 2^i$ doesn't exits we consider the next available node following the ring. For instance if node with id 4 isn't available we consider identifiers 5, 6 and 7 until we find a node.

Entry An entry of the routing table contains

- The number i.
- The successor.
- The keys managed by the successor. In particular the successor i handles all keys between $id + 2^i$ (included) and $id + 2^{i+1}$ (excluded).

$$[id + 2^i, id + 2^{i+1})$$



i	id+2^i	int.	succ.
0	1	[1, 2)	1
1	2	[2, 4)	2
2	4	[4, 0)	6

i	id+2^i	int.	succ.
0	2	[2, 3)	2
1	3	[3, 5)	6
2	5	[5, 1)	6

i	id+2^i	int.	succ.
0	3	[3, 4)	6
1	4	[4, 6)	6
2	6	[6, 2)	6

i	id+2^i	int.	succ.
0	7	[7, 0)	0
1	0	[0, 2)	0
2	2	[2, 6)	2

Figure 31.2: A DHT with finger tables.

Routing Consider the node with identifier id. For every entry $i = 0, ..., m-1$ (with m number of bits of keys and ids) we have to contact node i to look for resources with key between

$$id + 2^i$$

and

$$id + 2^{i+1} \mod 2^m$$

This means that to route a request for key k we have to consider the entry for which the following condition holds

$$id + 2^i \leq k < id + 2^{i+1} \mod 2^m$$

Complexity This solution allows to reach a peer in a logarithmic number of moves, in fact at every hop the search space is halved. It's also important to highlight that the routing table has $m = \log |N|$ number of entries (with $|N|$ maximum number of nodes in the network).

Joining

Disadvantage Even if this solution is a good trade-off between fast routing and small routing tables, we still have to handle the join of a node, in fact every time a node joins the network, all the routing tables have to be updated.

Joining process Each node keeps also track of its predecessor to allow counter-clockwise routing useful to manage join operations. When a peer n wants to join the network, it has to

1. Initialise the predecessor and fingers of node n.
2. Update the fingers and predecessors of existing nodes to reflect the addition of n.

We assume n knows another node n' already into the system. If this assumption is correct, n can use n' to initialise its predecessor and its predecessors' fingers.

To update the fingers we may observe that:

- Node n will become the i-th finger of node p if and only if
- p precedes n by at least 2^i, the i-th finger of p succeeds n.
- The first node p that can meet these two conditions is the immediate predecessor of node $n - 2^i$.

Complexity For each line (there are $\log|N|$ lines) in a routing table we have to do 1 lookup. Every lookup costs $\log|N|$ thus the complexity of joining is

$$\log^2|N|$$

Advantages and disadvantages

Advantages The main advantages of the finger table implementation are that

- It's always possible to lookup a key.
- Both the lookup and the dimension of the routing table are logarithmic.

Disadvantages The finger table implementation has some disadvantages, too. Such disadvantages don't allow this implementation to be widely use and are

- The network is more fragile than unstructured networks due to unstable peers.
- It's hard to implement search instead of lookup.
- This implementation does not consider physical topology.

Part IX

Big Data Platforms

Chapter 32

Introduction

Big data platforms are used to analyse huge amounts of data. In other words, such platforms focus on extracting knowledge from data to make decisions and solve problems. This solution is opposed to the traditional way of approaching a problem in which we extract knowledge from models.

This new way of approaching a problem can be very useful in problems in which computational power isn't as relevant as the capability of handling huge amounts of data.

Focus Big data platforms try to solve those problems that only occur when we deal with huge volumes of data.

Usages Big data is used in many fields, for instance in

- **Recommended algorithms**.
- **Translation**.
- **Genomics data**.
- **Medicine**.

32.1 Data collection and storage

In big data the focus is on data, in particular, we want to collect as much data as possible and store it so that it's always possible to obtain new knowledge from it. This is in conflict with the usual way of working in which data is filtered and discarded after usage.

Characteristics The main things to keep in mind when using big data are

- **Volume**, in fact, big data need to manage and handle a huge amount of data.
- **Velocity**, in fact data arrives in the system and gets updated continuously and swiftly and decisions (made analysing the data) have to be taken in a very small time window.
- **Variety**, in fact data can arrive in multiple different formats so we need some way to adapt and store it.
- **Veracity**, in fact, data may not be correct so has to be discarded.

Requirements Given the characteristics of big data platforms, we want to build systems that

- Quickly scale statistical analysis to large volumes of data (volume characteristic).
- Support quickly changing data (velocity characteristic).
- Are simple also considering parallelism and distribution.
- Are fault-tolerant because big data analysis is done on many machines that can fail on a distributed system.
- Offer monitoring tools to define if a certain query is too complex or if it's working correctly.
- Offer a clean abstraction for users because the experts that use the platform and define rules for a big data platform aren't programmers or computer science experts (a bit like SQL).

Chapter 33

MapReduce

MapReduce is a programming model introduced by Google in early 2000 to analyse big data.

Characteristics MapReduce has been designed to scale to large volumes of data but doesn't ensure velocity.

Problems MapReduce tries to solve those problems in which

- Data is too big to fit into memory and it's hard to distribute the data structure used to represent data in memory. For instance, it's difficult to work on a distributed graph and at the same time access such structure swiftly.
- We can't access input data easily and swiftly.
- We can't store and update the state of computation easily and swiftly.

This is the case of problems involving big data.

33.1 Model

MapReduce transposes functional programming to distributed systems. In particular, it uses the functions **map** and **reduce** to extract knowledge from large volumes of data while exploiting the parallelism offered by distributed systems.

Master-slave model MapReduce identifies

- A **master** node that has to coordinate all the slaves.
- **Slaves** that have to execute the actual computation.

Assumptions MapReduce is based on the following assumptions

- The volume of data is on the scale of **terabytes or more**. This means that the data can't fit in the disk of a machine, nonetheless in the memory.

- The distributed system is made of **normal computers** (i.e. not supercomputers) with no dedicated hardware. This assumption allows machines to be easily updated but makes them less reliable.
- The distributed system counts **hundred of thousands of nodes**.
- The master node can never fail.

33.1.1 Map and reduce

MapReduce works in a simple way,

1. The **data is divided** between all the nodes
2. Every node **computes the map function** on the data he received. The result of the map function is a couple key-value

   ```
   <key, value>
   ```

   The nodes that apply the map function are called **mappers**.
3. The key-value **pairs are redistributed** on the nodes according to the key. This means that a node receives all the couples that contain a certain key. In other words, two couples with the same key can't be on different nodes. In general, a node can handle multiple sets of keys (e.g. it can handle all the couples with key k_1 and k_2) but the reduce computation has to be handled separately (e.g. first to all the couples with k_1 to generate a result and then to all the couples with k_2 to generate another result).
4. The **reduce function is applied** to every couple key-value with the same key. The nodes that apply the reduce function are called **reducers**.

Mappers and reducers A node can be both a mapper and a reducer, in general, the coordinator initially tells all the nodes to compute the map function and once the operation is completed redistributes the data and tells them to execute the reduce operation. Every mapper executes the same map function and every reducer executes the same function. In other words the map and reduce function are the same for every node.

Immutable data MapReduce uses immutable data, in fact every time a function is applied to some data, a new set of data is returned. This means that data is never modified but only used to generate some other data.

Parallelism The map and the reduce functions can be computed independently from the other nodes, in fact,

- The map function doesn't depend on the result on other nodes.
- The reduce function works on data with the same key so can be computed without waiting for the computation of other nodes.

33.1.2 Coordinator

The coordinator of the system (the master)

- Allocates the resources to apply map and reduce. In particular it
  1. Divides the data into partitions.
  2. Sends the data to every node and asks them to apply the map function on the data received.
  3. Redistributes the data from mappers to reducers and asks them to apply the reduce function on the data received.
- Ensures fault-tolerance and takes care of synchronisation.

Stragglers If some nodes are too slow the coordinator tries to complete as many operations as possible and then reschedules to balance the load.

33.1.3 Advantages and disadvantages

Advantages The main advantages of this model are

- It's easy to use.
- It's very general.
- It's good for large scale data and systems.

Ease of use Among the advantages, it's important to underline why MapReduce is easy to use. This model is easy to use because a programmer only has to implement the functions map and reduce, in fact, the system takes care of all the stuff related to allocation, synchronisation and fault tolerance.

Optimisation We can also add that the cost of having an easy and general interface is to have a not optimised model. In other words, we sacrifice optimisation in favour of ease of use and generalisation.

Disadvantages The main disadvantages of the MapReduce model are

- It's slow.
- It's a very fixed paradigm, in fact, all the logic need for the analysis has to be written in the functions map and reduce.

33.2 Examples

It's time to look at some examples, to better understand how MapReduce works.

33.2.1 Word count

Let us consider a distributed system made of 20 nodes that, given a text, has to count the occurrences of all the words in the text. The result we want is

```
WORD        OCCURRENCES
My          2
Food        4
Meat        1
```

Map and reduce To solve this problem we can use MapReduce. In particular we can

1. Divide the text into lines. Each line is a partition.
2. Distribute the lines between all the nodes.
3. Apply the map function on every node. In particular the map function computes how many times a word appears in the line that has to be analysed. The map returns word-occurrences pairs. For instance, if the line was *Fear is the path to the dark side* we would get

   ```
   <fear, 1>
   <is, 1>
   <the, 2>
   <path, 1>
   <to, 1>
   <dark, 1>
   <side, 1>
   ```

4. Collect all the couples and redistributed them to the 20 nodes so that all couples with the same key are on the same node.
5. Apply the reduce function to all the couples on every node. In particular the reduce function sums all the values with same key. For instance if a node received the following couples

   ```
   <to, 1>
   <to, 1>
   <to, 1>
   <to, 1>
   ```

   it calculates `to = 4`.

33.2.2 Shortest path tree

MapReduce can also be applied to find the shortest path tree in a graph. For instance, consider a city map where we can find some refuelling stations represented. The map can be represented with a graph in which the nodes represent intersections and some special nodes represent the stations. We want to find, for every node, the closest refuelling station.

Map and reduce To solve this problem we can use MapReduce. In particular we can

1. Divide the graph into sub-graph. Every subgraph is centred on a station node and integrates all normal nodes that are within a fixed range (e.g. 1 km). Each sub-graph is a partition.
2. Distribute the sub-graph between all the nodes.
3. Apply the map function on every node. In particular, the map function computes the shortest path between every node in the graph and the station. The map function returns node-distance pairs.
4. Collect all the couples and redistributed them to the 20 nodes so that all couples with the same key are on the same node.
5. Apply on every node the reduce function to all the couples. In particular, the reduce function selects the minimum distance. For instance, if a node received the following couples

   ```
   <A, 2>
   <A, 1>
   <A, 3>
   ```

   it computes `A = 1`. We have to calculate the reduce function because a node A might end up in two different sub-graphs thus we want to take only the station that is closer to A, i.e. the station that has the minimum distance to A.

Chapter 34

Improvements to MapReduce

MapReduce allows us to analyse huge volumes of data exploiting a distributed system but

- Some of the assumptions on which is based are outdated.
- We can provide better abstractions than the map and reduce functions.
- We can add some more functionalities.

Improvements In particular we are going to analyse three different improvements

- **Arbitrary acyclic transformation graphs**.
- **Stream processing**.
- **Main memory usage**.

34.1 Models

To analyse how the improvements above have been implemented we will analyse two programming models

- **Scheduling of task**, implemented by **Apache Spark**.
- **Pipelining of task**, implemented by **Apache Flink**.

34.1.1 Scheduling of tasks

Multiple stages

The scheduling of tasks model extends the number of functions used from 2 (map and reduce) to an arbitrary number of operations. In particular, a set of operations that can be done without reshuffling (i.e. redistributing) the data among the nodes is called a **stage**. The stage is the unit of allocation, this means that the coordinator asks nodes to execute a stage. A stage can be made of multiple elementary operations (like the one used in functional programming).

Process We can generalise the process of analysing data as follows

1. All nodes receive a partition from the coordinator and execute a stage.
2. When every node has ended executing its stage the results are save to storage.
3. The coordinator redistributes the data.
4. Point 1 is executed applying the next stage.

We can see the MapReduce model as a particular case of task scheduling, in fact the only tasks executed are map and reduce.

Mini-batch

The scheduling of tasks model allows stream processing. In particular, it uses micro-batches to process streaming data. The procedure is the same as before but it's executed only when a node has enough data to analyse. Practically a node accumulates streaming data in a batch and starts the computation only when the batch is full.

Memory usage

The result of a stage has to be saved to storage before being redistributed but scheduling of tasks allows us to use main memory as a cache to improve performance.

34.1.2 Pipelining of tasks

The pipelining of tasks model has been designed for stream processing, in fact

- Tasks are statically allocated to nodes when the job is deployed (i.e. at the beginning of the job).
- A node immediately starts processing as soon as the data is received.

In other words computation of a stage can start as soon as some data of the previous stage is available. As the name says, pipelining of tasks works like a CPU pipeline in which after a stage is over the data is passed to the next node that can immediately compute the subsequent stage. Usually both map and reduce processes are active on every node and the node itself balances the two operations depending on the incoming data.

Communication The nodes communicate using TCP.

Stream processing Since data is executed as soon as it arrives on a node, pipelining of tasks suits well and naturally to stream processing.

34.2 Comparison

To compare the schedule of tasks and the pipelining of tasks models we are going to use the following parameters

- **Latency**. Latency measures how fast results updated.

- **Throughput**.
- **Load balancing**.
- **Elasticity**. Elasticity is the ability to grow and shrink the amount of resources used over time.
- **Fault tolerance**.

Latency

Latency is relevant in streaming scenarios because we want to analyse incoming data very fast because a new input will arrive in a small interval.

The pipeline model has better latency because there is no scheduler (the coordinator) that slows down operations (scheduling takes time). Furthermore in scheduling of tasks every node has to wait enough data to fill a batch, which increases latency.

Throughput

In general, both models perform well concerning throughput but depending on the situation and on the specific problem one might perform better than the other.

It's also worth mentioning that scheduling allows to optimise operations thus increasing throughput. Furthermore, scheduling can have a better throughput since computation is executed on blocks of data and not on smaller amounts like in pipelining (pipelining passes the data immediately to the next node).

Load balancing

In general, both models are good in load balancing but scheduling might help in some scenarios, in fact scheduling is decided dynamically based on the available resources of each node while in the pipeline approach the allocation of tasks is decided statically when the job is deployed (i.e. at the beginning of the job).

Elasticity

Only the scheduling of tasks model has good elasticity, in fact the scheduler can decide on how many nodes to distribute the stages. On the other hand, the pipelining of tasks isn't elastic because every node executes all the stages without the intervention of a scheduler. The only way to introduce some elasticity to pipeline of tasks is to restart the computation on a different set of physical nodes after taking a snapshot.

Fault tolerance

MapReduce is based on immutable data, this means that if a machine goes down, a stage can be repeated because the data of the previous state hasn't mutated and has been saved to storage.

The same thing isn't true in the pipelining of tasks model, in fact results of a stage are immediately forwarded to the next stage (i.e. they aren't saved to storage). To solve this problem the pipelining of tasks model periodically creates some checkpoints from which it's possible to resume computation after a failure.

In both cases we can assume that the input data is safely stored somewhere so that in the worst case we can restart all operations from there.

Part X

Security

Chapter 35

Introduction

Distributed systems share resource between different processes, thus it's important to secure such resources against not authorised access. In other words we want to allow only some authorised processes to access a resource. In distributed systems we can secure

- **Communication**.
- **Processes**.

Intruder

An intruder, i.e. a process that wants to mine a system's security, can be

- **Active**. An active intruder can listen to and communicate with the processes in the system.
- **Passive**. A passive intruder can only listen to the communication between the processes.

Access to a network An intruder is a process in the network of the distributed system. In particular an intruder can enter a network in an

- **Authorised way**.
- **Unauthorised way**.

Threads

The security of a distributed system can be compromised in different ways, in particular the main threads a system can face are

- **Interception**, when an unauthorised intruder reads some messages that travel the network. Some examples are sniffing and dumping.
- **Interruption**, when an intruder makes legitimate access to a server unavailable. Examples of interruption threads are disruption and denial of service attacks.
- **Modification**, when an active intruder changes something, usually the content of a message.
- **Fabrication**, when an active intruder creates and sends a new message. Some examples of fabrication threads are injection attacks and replay attacks.

Requirements

A distributed system can be considered secure if it satisfies the following requirements

- **Availability**. A system has to be ready for use at any point in time.
- **Reliability**. A system has to be ready for use for a long period of time without interruptions.
- **Safety**. Nothing bad has to happen.
- **Maintainability**. A system has to be easy to fix.
- **Confidentiality**. The information possessed by a system has to be read only by those users that are authorised.
- **Integrity**. Data doesn't have to change without permission.

Security policy To fulfil such requirements we have to define a **security policy**, i.e. a set of rules that define what is legitimate and what is not.

Security mechanism A security mechanism allows to enforce a security policy. We can divide security mechanisms in four categories

- **Encryption**. Encryption ensures confidentiality and integrity.
- **Authentication**. Authentication ensures identification of the users.
- **Authorisation**. Authorisation ensures that only authorised users access the resources.
- **Auditing**. Auditing ensures breach analysis.

Approaches to security

When securing a distributed system we can put the focus on different aspects. In particular we can identify three different approaches (that can be used together)

- **Protect the data**. This approach focuses on protecting the data regardless of the operations that are executed on the data.
- **Protect the operations**. This approach focuses on protecting the operations that can be executed on the data (but not directly on the data).
- **Protect the users**. This approach focuses on protecting the users deciding who is authorised to do what.

Layered structure

Each process in a distributed system is build using a layered architecture. This means that we have to decide between which levels to put the security mechanisms. To take such decision we have to understand which layers to trust. For instance a middleware may have to trust the operating system layer that has to trust the hardware. In particular if we don't trust the security of a low level we have to add some security mechanisms at a higher level.

Trusted Computing Base The Trusted Computing Base (TBC) is the set of services, processes' layers and components that the system trust.

Simplicity

Ideally we would like a security mechanism to be simple. This is not always possible, in fact in many cases we have to consider some corner cases that make a mechanism more complex.

Chapter 36

Cryptography

36.1 Encryption

Encryption is a way to transform a message P, called plaintext, into another message C, called ciphertext.

36.1.1 Encryption function

The encryption function should be invertible to allow decryption only for authorised users (i.e. a user can obtain P given C only if authorised).

Hidden function

Initially we tried to secure distributed systems by hiding the encryption function. This made the function hard to design (because it had to satisfy many mathematical properties) and the system not very secure because it only relied on the fact that the function was hidden. Moreover if the function is secret, the community can't verify if it's actually secure.

Parametric function

Nowadays we use parametric public encryption functions. In particular the parameter used to encrypt a plaintext is a key. In general we can use a key K_E for encryption and a key K_D for decryption.

36.1.2 Symmetric key encryption

In symmetric key encryption, the key used for encryption and decryption is the same

$$K_E = K_D = K$$

The key K is called shared key and can be used by two processes A and B to communicate safely. We represent the shared key for A and B as

$$K_{A,B}$$

Number of keys In a distributed system with N nodes we have to compute a quadratic number of keys, in fact every node has to generate a symmetric key for each other node in the system

$$\mathcal{O}(N^2)$$

36.1.3 Public (asymmetric) key encryption

In public key encryption (also called asymmetric key encryption) the keys for encryption and decryption are different

$$K_E \neq K_D$$

Let us consider a node A. The key

- K_E, also called public key K_A^+, is made publicly available.
- K_D, also called private key K_A^-, is kept secret by A. The key K_A^- can't be obtained starting from K_A^+.

Encryption and decryption When a process A wants to send a message m to a node B, it can encrypt A with B's public key K_B^+.

$$C = K_B^+(m)$$

B can decipher the message using its private key K_B^-

$$m = K_B^-(C) = K_B^-(K_B^+(m))$$

and because K_B^- is kept secret by B, only B can read the message sent by A. On the other hand everyone can send a message to B because K_B^+ is public.

Number of keys In this case every node n can generate the couple of keys $< K_n^-, K_n^+ >$, in fact every other node can communicate with a node i using i's public key. In total in a system with N nodes we have to generate a linear number of keys

$$\mathcal{O}(N)$$

36.2 Hash

An hash function h is a function that takes a message of any length as input and transforms it in a string of fixed length called digest.

The codomain of the hash function is smaller than the domain of the function, this means that the hash function is not invertible, in fact two different strings can be mapped in the same digest. In other words given a digest we can't find a unique string that, given as input of the hash function, generates such digest.

Collision A collision happens when two messages generate the same digest.

36.2.1 Properties

An hash function has to be

- **One way**, i.e. it has to be difficult to find one of the possible input messages given a digest.
- **Weakly collision resistant**, i.e. given a message m and a digest d it has to be hard to find another message m' that generates the same digest d.
- **Strongly collision resistant**, i.e. it has to be hard to find two messages m and m' that generates the same digest d.

36.3 Digital signatures

Digital signatures are used to ensure authenticity. In particular a process A can sign a message m so that every process that reads m is sure that the message has been sent by A.

36.3.1 Message signing

To sign a message we can use public key encryption. Let us consider a process A that wants to send a message m to B. The algorithm for generating the signature is the following

1. A encrypts the message m using the private key K_A^-.

   $$S = (m, K_A^-(m))$$

   S is the signature of the message m generated by A. If we sent S to B everyone could read the message so we have to encrypt S so that only B can read it.

2. A encrypts the couple m, K_A^- with B's public key K_B^+

   $$S_e = K_B^+(m, K_A^-(m))$$

   S_e is the encrypted signature that can only be read by B because only B can obtain $(m, K_A^-(m))$ using its private key K_B^-.

3. B deciphers S_e using its private key K_B^- and obtains the signature of A

   $$S = K_B^-(K_B^+(S_e)) = (m, K_A^-(m))$$

4. B deciphers $K_A^-(m)$ using A's public key K_A^+ and checks if the result equals to m

   $$K_A^+(K_A^-(m)) = m$$

This process works because

- Only B can read $m, K_A^-(m)$ because it's encrypted with B's public key.
- B can decipher $K_A^-(m)$ because everyone knows A's public key.
- If $K_A^+(K_A^-(m)) = m$ then only A can have encrypted m.

36.3.2 Document signing

Signing an entire document using public key encryption would be too expensive so we sign the hash of the document (the hash has a fixed length and is much smaller than a document).

The process of signing a document d becomes

1. A generates the H hash of the document d.
2. A encrypts the digest H using the private key K_A^-.

   $$S = (d, K_A^-(H))$$

   S is the signature of the message m generated by A and contains the document to send and the encrypted digest. In this case we don't want to encrypt S because it would be to expensive to encrypt the document d.
3. B deciphers $K_A^-(H)$ using A's public key K_A^+ and computes the hash H' of d.

   $$H = K_A^+(K_A^-(H))$$
4. B checks if the hash H' computed locally equals to the value received

   $$K_A^+(K_A^-(H)) = H = H' = h(d)$$

36.3.3 Certificate Authorities

Signatures allow us to check if a message has been sent by a certain process using private keys, but we don't know if the key K_A^+ used to verify the signature really belongs to A. In other words we don't know if the process we are talking to really is A or is someone who is impersonating A.

Trusting a process To associate an identity to a certificate we need a **certificate** distributed by a **Certificate Authority** (CA). In particular a certificate is a couple

```
<identity, public_key>
```

The certificate is signed by the CA that releases it to ensure that the certificate has been sent by the CA.

Trusting a Certificate Authority The problem now is that we have no way to know if the certificate has been sent by a CA or by some server that impersonates the CA.

To solve this problem we can use

- Hierarchies of CA in which the root CA is trusted by everyone (usually an OS has a list of trusted CAs).
- PGP's web of trust in which users authenticate other users by signing their certificates.

Chapter 37

Secure channels

A secure channel is a channel that ensure secure communication against

- **Interception**.
- **Modification**.
- **Fabrication**.

but not against interruption.

We can obtain secure channels using the following types of protocols

- **Challenge and response**.
- **Key Distribution Centre**.

37.1 Challenge and response

Challenge and response protocols allow to processes to authenticate.

37.1.1 Session

After authentication (with shared keys, as we are going to see) a new symmetric key is created. The new shared key can be used for the entire session (i.e. a sequence of messages). This means that the communication between two processes A and B involves

- A shared key $K_{A,B}$ for authentication.
- Multiple shared keys $K^i_{A,B}$ for communication. Each key is used for a different session.

Session keys have been introduced because the same key, if used for authentication and encryption, might be easier compromised.

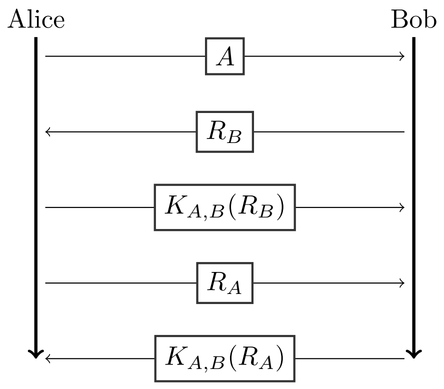


Figure 37.1: Symmetric key challenge and response.

37.1.2 Symmetric key authentication

Let us assume that two processes (say Alice and Bob) have a shared key $K_{A,B}$. A and B can establish a secure channel with the following protocol

1. Alice sends her identity A to Bob in plaintext.
2. Bob sends a challenge R_B to Alice in plaintext. The challenge is a random number.
3. Alice replies to Bob. In particular Alice encrypts R_B using the shared key $K_{A,B}$. If Bob can decipher $K_{A,B}(R_B)$ then A must have $K_{A,B}$ so it must be Alice.
4. Alice sends a challenge R_A to Bob in plaintext. The challenge is a random number.
5. Bob replies to Alice. In particular Bob encrypts R_A using the shared key $K_{A,B}$. If Alice can decipher $K_{A,B}(R_A)$ then B must have $K_{A,B}$ so it must be Bob.

37.1.3 Short symmetric key

The symmetric key challenge and response protocol can be simplified grouping some messages, in particular

1. Alice can send her identity A and the challenge R_A in the same message.
2. Bob can send the reply to Alice's challenge and it's challenge in the same message.
3. Alice can finally reply to Bob's challenge.

This protocol though, is vulnerable to the reflection attack. In particular an intruder can exploit the fact that Alice sends her challenge before authenticating Bob. Practically the intruder can authenticate itself using two challenge and response sessions, in the first session

1. Send a message
$$A, R_I$$
to Bob.

2. Bob replies with
$$R_B, K_{A,B}(R_I)$$
thinking that the first message has been sent by Alice.

In the second session

1. Send a message
$$A, R_B$$
to Bob. The challenge R_B sent to Bob is the one received in the first session.
2. Bob replies with
$$R_{B2}, K_{A,B}(R_B)$$
thinking that the first message has been sent by Alice.

Finally the intruder can terminate the first session sending the response
$$K_{A,B}(R_B)$$
received in the second session.

37.1.4 Public key authentication

The short version of the challenge and response protocol can be fixed using public key encryption. In particular the authentication process between Alice and Bob is the following

1. Alice sends her identity A and the challenge R_A. The couple is encrypted using Bob's public key.
$$K_B^+(A, R_A)$$
2. Bob replies with the challenge sent by R_A, his challenge R_B and the shared key $K_{A,B}$. Bob's answer is encrypted using Alice's public key.
$$K_A^+(R_A, R_B, K_{A,B})$$
3. Alice replies with the challenge R_B encrypted using the shared key $K_{A,B}$
$$K_{A,B}(R_B)$$

With public key encryption we can use the short version because A and R_A can only be read by Bob and not by an intruder (because they are ciphered using Bob's public key). Notice that the key $K_{A,B}$, exchanged from message 2 on, is the session key used from A and B to communicate.

37.2 Key Distribution Centre

The main problem with challenge and response protocols is that we have to create a lot of keys. Say a system has N processes, the number of keys is

- Quadratic with respect to N if we use symmetric key authentication.
- Linear with respect to N if we use public key authentication.

To mitigate this problem we can use a trusted central server called Key Distribution Centre (KDC). The protocols we will analyse assume that all nodes share a key $K_{N,KDC}$ with the KDC.

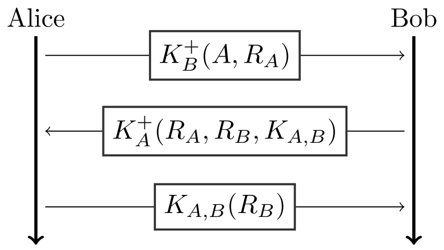


Figure 37.2: Public key challenge and response.

Ticket When Alice wants to talk to Bob, she has to request a ticket to the KDC. The ticket is the shared key between Bob and Alice encrypted using the shared key between the KDC and Bob

$$K_{KDC,B}(K_{A,B})$$

The key $K_{A,B}$ is the session key used for encryption in the session. Tickets allow Alice and Bob to authenticate because both trust the KDC, thus Bob can trust Alice if she sends a ticket to Bob, using $K_{KDC,B}$ because it came from the KDC.

37.2.1 Basic protocol

The basic protocol to obtain a ticket works as follows

1. Alice sends her identifier and Bob's identifier to the KDC.

$$A, B$$

2. The KDC replies with the ticket $K_{KDC,B}(K_{A,B})$ and the shared key $K_{A,B}$ encrypted using the key shared between A and the KDC

$$K_{A,KDC}(K_{A,B}), K_{B,KDC}(K_{A,B})$$

3. Alice sends her identifier and the ticket $K_{B,KDC}(K_{A,B})$ to Bob. Bob can obtain the shared key because he can decipher the tiket using the shared key between him and the KDC.

$$A, K_{B,KDC}(K_{A,B})$$

37.2.2 Identifier protocol

Replay attack The basic protocol can be attacked using a replay attack. In particular

1. The intruder can intercept the message A, B and send the message A, C to the KDC instead.
2. The KDC replies to the intruder sending the ticket $K_{KDC,I}(K_{A,I})$ and the symmetric key $K_{A,I}$ encrypted with the key of A and KDC.
3. The intruder forwards the message received by the server to A.

This attack works because A thinks that she has the key $K_{A,B}$ to talk to B but she actually has the key to talk to the intruder, thus when Alice sends a message to Bob, the intruder can intercept and read it.

Protocol To solve this protocol we can modify the reply sent by the KDC. In particular the KDC sends

- The identifier B of Bob.
- The shared key $K_{A,B}$.
- The token $K_{KDC,B}(K_{A,B})$.

The message is encrypted using the shared key between A and the KDC so that the intruder can't edit KDC's answer.

$$K_{A,KDC}(B, K_{A,B}, K_{KDC,B}(K_{A,B}))$$

37.2.3 Nonce protocol

Reply attack Even using an identifier an intruder can still attack the system. In particular an intruder, after stealing the shared key of B and the KDC, can save a response message $K_{KDC,B}(K_{A,B})$ from the KDC and use it to impersonate the KDC and replay to A using such message. In fact Alice after receiving KDC's reply, sends the token to Bob. The intruder can intercept the token and use the stolen $K_{B,KDC}$ to obtain the key of A and B.

Protocol To solve this problem we can add challenges to the messages. In particular when Alice sends a message she also includes a challenge. The protocol works this way

1. Alice sends
   - a challenge R_A,
   - her identifier A,
   - Bob's identifier B,

   to the KDC in plaintext.
   $$R_A, A, B$$
2. The KDC replies with
   - the challenge sent by Alice R_A
   - the shared key $K_{A,B}$,
   - Bob's identifier B,
   - the ticket (that also contains Alice's id) $K_{KDC,B}(K_{A,B})$,

   The reply is encrypted using the shared key of A and the KDC.
   $$K_{A,KDC}(R_A, B, K_{A,B}, K_{B,KDC}(A, K_{A,B}))$$
3. Alice sends to Bob
   - the ticket $K_{B,KDC}(K_{A,B})$,
   - a challenge encrypted using the shared key of Bob and Alice $K_{A,B}(R_{A2})$

.

$$K_{A,B}(R_{A2}), K_{B,KDC}(A, K_{A,B})$$

Bob can obtain the shared key because he can decipher the ticket using the shared key between him and the KDC. For the same reason, Bob can also read the challenge R_{A2}.

4. Bob replies with
   - his challenge R_B,
   - Alice's challenge decremented of 1 $R_{A2} - 1$

   The reply is encrypted using $K_{A,B}$

   $$K_{A,B}(R_{A2} - 1, R_B)$$

5. Alice finally sends a reply to Bob's challenge to end the authentication

   $$K_{A,B}(R_B - 1)$$

37.2.4 Double nonce protocol

Reply attack If the intruder can steal $K_{A,B}$ then it's possible to execute a reply attack on the single nonce (challenge) protocol. In particular the intruder can replay the first message Alice sends to Bob.

Protocol To fix this problem we can ask Bob to authenticate Alice before Alice asks the KDC a ticket. In particular the protocols works as follows

1. Alice sends a message to Bob with her identifier A.

   $$A$$

2. Bob answers with a nonce R_{B1} encrypted with the shared key of Bob and the KDC.

   $$K_{B,KDC}(R_{B1})$$

3. Alice sends
   - a challenge R_A,
   - her identifier A,
   - Bob's identifier B,
   - the challenge received from Bob $K_{B,KDC}(R_B1)$

   to the KDC in plaintext.

   $$R_A, A, B, K_{B,KDC}(R_B1)$$

4. The KDC replies with
   - the challenge sent by Alice R_A
   - the shared key $K_{A,B}$,

- Bob's identifier B,
- the ticket (that also contains Alice's id and Bob's challenge) $K_{KDC,B}(A, K_{A,B}, R_{B1})$,

The reply is encrypted using the shared key of A and the KDC.

$$K_{A,KDC}(R_A, B, K_{A,B}, K_{B,KDC}(A, K_{A,B}, R_{B1}))$$

5. Alice sends to Bob
   - the ticket $K_{B,KDC}(A, K_{A,B},)$,
   - a challenge encrypted using the shared key of Bob and Alice $K_{A,B}(R_{A2})$

   .

   $$K_{A,B}(R_{A2}), K_{B,KDC}(A, K_{A,B}, R_{B1})$$

   Bob can obtain the shared key because he can decipher the ticket using the shared key between him and the KDC. For the same reason, Bob can also read the challenge R_{A2} and the challenge R_{B1} initially sent to Alice.

6. Bob replies with
   - his challenge R_{B2},
   - Alice's challenge decremented of 1 $R_{A2} - 1$

   The reply is encrypted using $K_{A,B}$

   $$K_{A,B}(R_{A2} - 1, R_{B2})$$

7. Alice finally sends a reply to Bob's challenge to end the authentication

   $$K_{A,B}(R_B - 1)$$

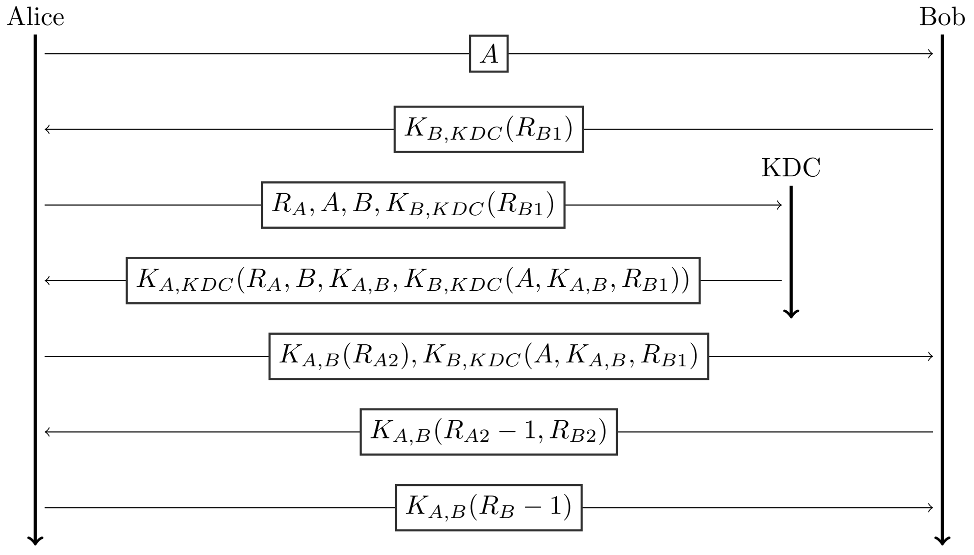

Alice
Bob
KDC
A
$K_{B,KDC}(R_{B1})$
$R_A, A, B, K_{B,KDC}(R_{B1})$
$K_{A,KDC}(R_A, B, K_{A,B}, K_{B,KDC}(A, K_{A,B}, R_{B1}))$
$K_{A,B}(R_{A2}), K_{B,KDC}(A, K_{A,B}, R_{B1})$
$K_{A,B}(R_{A2} - 1, R_{B2})$
$K_{A,B}(R_B - 1)$

Figure 37.3: Needham-Schroeder protocol for centred key distribution.

Chapter 38

Security management

In a distributed system we have to handle, distribute and manage keys and secrets. In particular we are going to discuss

- Distributing keys.
- Storing keys.
- Handling group communication.
- Authorisation management.

38.1 Key distribution

Key distribution (or agreement) protocols are used to distribute or agree on public or symmetric keys.

38.1.1 Diffie-Hellman

The Diffie-Hellman protocol allows to parties (say Alice and Bob) to agree on a symmetric key over an insecure channel (i.e. everyone can listen the channel and modify the messages). At the beginning of the protocol Alice and Bob choose two numbers a and b (that have to be kept private) and agree on two numbers (publicly) n and g. After this initial phase

1. Alice sends $g^a \mod n$ to Bob.
2. Bob can compute $(g^a \mod n)^b \equiv g^{ab} \mod n$.
3. Bob sends $g^b \mod n$ to Alice.
4. Alice can compute $(g^b \mod n)^a \equiv g^{ab} \mod n$.

Both Alice and Bob share the same key $K_{A,B} = g^{ab} \mod n$. Notice that an intruder can't obtain a and b from $g^a \mod n$ and $g^b \mod n$ because to find such values he should solve the discrete logarithm problem (which is a well known hard problem).

Man in the Middle Diffie-Hellman is very useful, still it has some problems. In particular it's vulnerable to Man in the Middle (MiM) attacks, in fact a third party (say Eve) can impersonate Bob when talking to Alice and Alice when talking to Bob. Alice and Bob won't notice that their are talking directly to Eve because DH doesn't require any authentication.

For this reason the Diffie-Hellman protocol has to be used when the parties can authenticate after the key agreement with some other mechanism. For instance, if we connect to Amazon

- We know that Amazon is legitimate because we can use its certificate to check that the site we are visiting really is Amazon.
- Amazon knows that we are who we say because we can login using a password or multi-factor-authentication.

38.2 Secure group communication

Let us consider a group of n process that want to communicate securely. We want to ensure the following conditions

- Every process can communicate with any other process in the group.
- An external process can't understand what the group processes are saying (no interception, fabrication or modification).

38.2.1 Public and symmetric key

Secure group communication could be achieved using public or symmetric key encryption. Both methods have disadvantages that becomes worst considering a big number of nodes. In particular

- Symmetric key encryption requires to compute a quadratic number of keys with respect to the number of nodes
$$\mathcal{O}(n^2)$$
because every node has to share a key with every other node. Furthermore the encryption process is linear because, when sending a broadcast message, for every other node we have to encrypt the message with the corresponding shared key.
$$\mathcal{O}(n)$$
- Public key encryption requires to compute a linear number of keys with respect to the number of nodes
$$\mathcal{O}(n)$$
because every node has to compute its private and public key. Like before, the encryption process is linear because a node has to encrypt the message with the public key of the receiver (and every receiver has a different key).
$$\mathcal{O}(n)$$
Furthermore public key encryption is very expensive.

Notice that the key generation complexity corresponds to the number of keys in the system.

	Paired shared keys	Public keys	Single shared key
Key generation complexity	$\frac{n(n+1)}{2} \Rightarrow \mathcal{O}(n^2)$	$2n \Rightarrow \mathcal{O}(n)$	$1 \Rightarrow \mathcal{O}(1)$
Encryption complexity	$n \Rightarrow \mathcal{O}(n)$	$n \Rightarrow \mathcal{O}(n)$	$1 \Rightarrow \mathcal{O}(1)$

Table 38.1: A summary of complexities for group communication protocols.

38.2.2 Single shared key

Ideally we would like to find some protocol that allows to create a single shared (symmetric) key that can be used to encrypt and decrypt every message sent between any two nodes of the group. This means that encryption and key generation complexities are constant.

Dynamic groups Another important thing to consider is that processes may join or leave groups. In dynamic groups we want to modify the single shared key to ensure that

- A new node that legitimately joined the group can only understand the messages sent after it joined. This property is called **backward secrecy**.
- A node that left the group can't understand the messages sent after it left. This property is called **forward secrecy**.

Protocol Let us consider a group of nodes that share a key k.

- When a new node n^+ **joins** the group we can send it the key k encrypted with the public key of n^+ so that the shared key k can be read only by n^+ and not by other nodes that want to gain access to the group illegitimately. This operation is very cheap in fact we don't have to calculate another key and we only have to encrypt and send a message.

$$\mathcal{O}(1)$$

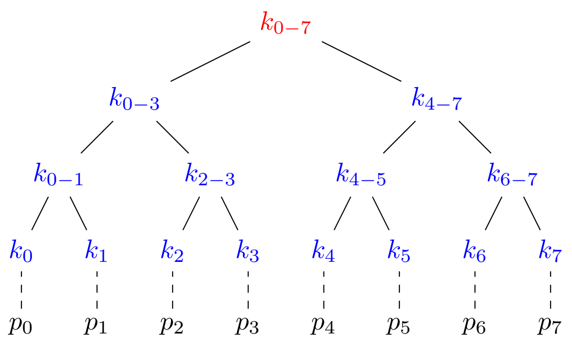
- When a node n^- **leaves** the group we have to revoke the old key k and compute a new k' (otherwise n^- could still read the messages after it left). The new key k' can be sent to all the nodes currently in the group using their public keys. This operation is much more expensive than a join operation because we have to encrypt and send N messages (with N number of nodes in the group).

$$\mathcal{O}(n)$$

The goal is to find some protocols that are a trade-off between the expensive leave operation and the efficient join operation.

Key creation The new key that has to be generated when a node leaves the group can be computed by

- A **leader node**. Using this method we have to consider the problem of distributing the key to all the other nodes.
- **All the nodes** using a generalisation of the Diffie-Hellman protocol.


k_{0-7}
k_{0-3}
k_{4-7}
k_{0-1}
k_{2-3}
k_{4-5}
k_{6-7}
k_0 k_1 k_2 k_3 k_4 k_5 k_6 k_7
p_0 p_1 p_2 p_3 p_4 p_5 p_6 p_7
Figure 38.1: Logical key hierarchy tree (KEKs in blue, DEK in red).

38.2.3 Logical key hierarchy

The logical key hierarchy protocol can be used to efficiently distribute a key generated by a leader node when a process leaves the group.

Structure In logical key hierarchy keys are organised in a hierarchical structure, i.e. a tree.

Keys In this protocol we don't have a single shared key (nor a key for every pair of nodes), in fact we have

- A Data Encryption Key (DEK) to encrypt the messages sent between any two nodes.
- A set of Key Encryption Keys (KEK) to encrypt the keys.

The keys are organised in a tree structure with the DEK in the root and the KEKs in the nodes. The leaves of the tree are the nodes in the group and every node knows only the keys on the path to the root.

Leave When a process n^- leaves, all the keys from the root (i.e. DEK) to the key of n^- are invalidated. For instance if process p_4 leaves the group, the keys k_4, k_{4-5}, k_{4-7} and k_{0-7} in tree 38.1 are invalidated.

The invalidated keys have to be generated again (k'_{4-5} is the value that replaces $k_{4,5}$) and have to be distributed to all nodes. When we distribute a key we have to encrypt it with the upper untouched key in the tree. In particular

1. k'_{4-5} can be distributed to p_5 using k_5 as encryption key.
2. k'_{4-7} can be distributed to p_5 using k'_{4-5} as encryption key.
3. k'_{4-7} can be distributed to p_6 and p_7 using k_{6-7} as encryption key.
4. k'_{0-7} can be distributed to p_5, p_6 and p_7 using k'_{4-7} as encryption key.
5. k'_{0-7} can be distributed to p_0, p_1, p_2 and p_3 using k_{0-3} as encryption key. We can see the advantage of this structure in this instruction, in fact we can distribute the DEK to all processes from 0 to 3 using a single KEK that is the closest KEK to the key to distribute. Without this structure we would have used k_0, k_1, k_2 and k_3.

Join When a process joins the group all the keys (KEK and DEK) to the root has to be send to the process.

Trade-off The logical key hierarchy allows to add and remove nodes with a logarithmic complexity (given by the tree structure). With respect to the single key protocol the join cost has slightly increased but we have also improved the join cost a lot.

38.2.4 Centralised flat table

Centralised flat tables use Data Encryption Keys and Key Encryption Keys like logical key trees but DEKs and KEKs are organised differently.

Centralised flat table Each process has an identifier of N bits (thus we can have at most 2^N nodes). The centralised flat table has

- N columns, each of which represents a bit position in the identifier.
- 2 rows, because a bit can be 0 or 1.

Each cell of the table contains a KEK key $k_{i,j}$ in which

- The first index i represents the bit number.
- The second index j represents the value of the i-th bit.

Every node has all the keys $k_{i,j}$ that represent every index of the identifier of the node.

	$bit_{id}0$	$bit_{id}1$	$bit_{id}2$	$bit_{id}3$
bit = 0	$k_{0,0}$	$k_{1,0}$	$k_{2,0}$	$k_{3,0}$
bit = 1	$k_{0,1}$	$k_{1,1}$	$k_{2,1}$	$k_{3,1}$

Table 38.2: A centralised flat table.

For instance the node n_9 has identifier 1001 thus, being 0 the position of the leftmost bit, it has the keys

- $k_{0,1}$ because the bit in position 0 has value 1.
- $k_{1,0}$ because the bit in position 1 has value 0.
- $k_{2,0}$ because the bit in position 2 has value 0.
- $k_{3,1}$ because the bit in position 3 has value 1.

Notice that a node has a unique set of keys, in fact no other node has the same binary representation of the id (and the keys a node has are based on that). For instance n_9 and n_8 share keys $k_{0,1}$, $k_{1,0}$ and $k_{2,0}$ but can't share $k_{3,1}$, because the last bit of id 8 is 0, thus n_8 has $k_{3,0}$ instead. In particular every node has at least one different key with respect to all other nodes.

Leave If node n_9 leaves than the DEK and all keys of n_9 have to be invalidated. In the key-redistribution phase we

1. Initially send the new DEK (i.e. DEK') encrypted with all keys not used by the process who left. This way we ensure that n_9 doesn't receive DEK'. All processes have at least one of the keys used for encryption because the ids differ for at least one bit from the removed node, thus every process can decrypt and obtain DEK'.

   For instance if n_9 leaves we send DEK' to all nodes using $k_{0,0}$, $k_{1,1}$, $k_{2,1}$ and $k_{3,0}$.

2. Finally send the new values of keys previously hold by n_9. In particular we want that such keys are only accessible to the nodes of the group (not to n_9) that have the same key (otherwise if another node leaves the group). To achieve such result we

   (a) Generate the new value $k'_{i,j}$.
   (b) Encrypt $k'_{i,j}$ with the old $k_{i,j}$ to allow only nodes that previously had $k_{i,j}$ to read $k'_{i,j}$.
   (c) Encrypt the result of point 2 with DEK' to allow only to group members to read the key.

   Long story short, a node receives
   $$DEK'(k_{i,j}(k'_{i,j}))$$
   We have to ensure that $k'_{i,j}$ is read only by those nodes that previously had $k_{i,j}$ because otherwise if a node leaves the group then it will understand the messages sent at point 1 and break forward security.

   In our example we send

   - $DEK'(k_{0,1}(k'_{0,1}))$
   - $DEK'(k_{1,0}(k'_{1,0}))$
   - $DEK'(k_{2,0}(k'_{2,0}))$
   - $DEK'(k_{3,1}(k'_{3,1}))$

38.3 Secure replicated servers

In distributed systems we can have replicated resources across multiple servers that can be accessed from clients, thus have to be secured. In particular we want the client to discover if a distributed resource has been corrupted by an intruder (because the server it comes from has been taken over from the intruder). The client can has to be robust against c resources, i.e. it has to be able to filer out at most c corrupted resources.

38.3.1 Simple solution

The simplest solution to this problem is to use $2c + 1$ servers so that if c responses from the servers are corrupted, still the majority of responses (i.e. $c + 1$) are not corrupted.

Verifying resources A client can verify a resource using digital signatures, in particular the server has to sign the resource and the client can verify the signature. This forces the client to know the identity of all $2c + 1$ servers.

38.3.2 n, m threshold schemes

n, m threshold schemes allow to agree on the correctness of a resource in an efficient way. In particular such schemes use only a secret (usually a signature) that is divided into m pieces. n pieces are sufficient to reconstruct the secret. If we want to ensure robustness for at most c servers then n has to be $c + 1$.

Protocol The protocol works this way

1. Every server sends a response with the resource and the signature of such response.
2. The client collects all m responses and has to find a set of n responses for which the signatures, grouped together can create a valid signature (i.e. the secret).

38.4 Access control

In distributed systems we want to create right to define for every user what resources it can access (or for every resource from who it can be accessed).

Process In general when a user (also called **subject**) wants to access a resource, has to ask a **reference monitor** to execute an operation on an **object**. The reference monitor verifies the right of the user and if it's allowed it executes the operation on the object.

Access control matrix Rights are represented in a matrix with

- Users on the rows.
- Objects on the columns.

Every cell $C_{x,y}$ of the table contains the rights of user y on resource (object) x.

	Object 1	⋯	Object n
User A	Read, Write	⋯	Read
⋯	⋯	⋯	⋯
User K	Read	⋯	Read, Write

Table 38.3: An access control table.

This representation is rather inefficient, in fact many cells are empty because many users have no rights for a resource. To solve this problem we can organise

- Column-wide, i.e. for every object we define a list of users that have some right on the object and their respective rights.
- Row-wide, i.e. for every user we define a list of objects that the user can access or modify and the specific rights of the user.

38.4.1 Access Control Lists

Access Control Lists are used to implement column-wide access control matrices. In particular, for each object, the ACL contains a list of users with the respective rights on the object.

```
ASL[O1] = {"UA" : [R, W],
           "UK" : [R]}
```

The ACLs are on the servers and when a client s wants to access a resource o, it sends a read request r and the server grants access to such resource if s is in the ACL of the requested resource o and s has the rights to execute the request r.

Groups Subjects can be organised in groups and we can assign rights to entire groups so that all subject in a group share the same rights.

Roles We can also assign roles to subjects so that all subject with a certain role share the same rights. In other words roles are like groups but are more dynamic (a user can change role over time).

38.4.2 Capabilities

Capabilities are used to implement row-wide access control matrices.

Capability A capability is a string of 128 bits, in particular

- The first $I = P + O$ bits define the identifier of an object. In particular
  - The first P bits define the server port of the server on which the object is.
  - The last O bits define the object itself.
- The following R bits define the rights on the object.
- The last U bits are used to make the capability unforgeable (i.e. the client cannot change it). These bits, on delegated capabilities, are generated as

  $$U = f(C \oplus R)$$

  where

  - f is a one way function.
  - R are the rights.
  - C is a value generated by the server upon creating the object O to which the capability refers.

  These bits are unforgeable because f is one way and some bits of information are hidden on the server, thus the owner of the capability can't forge U and pretend having different rights.

A subject can use a capability to access an object, thus a subject S should not give the capability to other untrusted subjects because it allows the other users to have the same rights of S.

Delegation

Capabilities can be delegated, this means that a subject can delegate some of its processes to other subjects. We would also like to delegate a subset of capabilities (for instance R instead of W,R).

Proxy To delegate restricted capabilities we can use proxies. A proxy created by a grantor A to give rights R to a grantee G is made of

- Access rights R.
- The public part of a secret S^+_{proxy}.
- A signature $sig(A, R, S^+_{proxy})$.
- The private part of a secret S^-_{proxy}.

The first three part of the proxy are a **certificate**.

Protocol Consider that Alice wants to delegate rights R to Bob.

1. Alice sends to Bob
   - The rights R and the public part of the proxy both signed by Alice.
     $$[R, S^+_{proxy}]_A$$
     This message can be read by everyone.
   - The secret part of the proxy encrypted with the shared key of Alice and Bob.
     $$K_{A,B}(S^-_{proxy})$$
     This message can only be read by Bob.
   $$[R, S^+_{proxy}]_A, K_{A,B}(S^-_{proxy})$$
2. Bob verifies the signed message sent by Alice and when he wants to execute some operation sends
   $$[R, S^+_{proxy}]_A$$
   to the server.
3. The server verifies the signature of the message received and answers with a nonce N encrypted with the public part of the secret.
   $$S^+(N)$$
4. Bob can decrypt the nonce N because he has received S^- from Alice and sends N to the server.
5. The server grants access to Bob because he can reply with N only if he had received the proxy from the original possessor of the rights (i.e. Alice).

Protocol for restricted rights Consider now that Alice wants to delegate rights R to Bob and Bob wants to delegate a restriction R' of R to Chuck.

1. Alice sends to Bob
   - The rights R and the public part of the proxy, both signed by Alice.
     $$[R, S^+]_A$$
     This message can be read by everyone.
   - The secret part of the proxy encrypted with the shared key of Alice and Bob.
     $$K_{A,B}(S^-)$$
     This message can only be read by Bob.

   $$[R, S^+]_A, K_{A,B}(S^-)$$

2. Bob verifies the signed message sent by Alice and sends to Chuck
   - The signed part received from Alice.
     $$[R, S^+]_A$$
   - The restricted rights R' and the public part of secret S_B^+ different from S^+. Both are signed with the private part of the secret sent by Alice.
   - The private part of the secret S_B^-.

   $$[R, S^+]_A, [R', S_B^+]_{S^-}, K_{B,C}(S_B^-)$$

3. Chuck sends both the signed right to the server
   $$[R, S^+]_A, [R', S_B^+]_{S^-}$$
   and keeps the private part of the secret S_B^- received from Bob.

4. The server verifies the signature of both the messages received, in particular he can obtain S^+ and verify the first signature using K_A^+. After retrieving S^+ the server can use it to verify the signature of the second message because it's signed with S^-. For this reason the server can also understand that the rights of the second message are a restriction of the ones in the first message.

5. The server answers with a nonce N encrypted with the public part of the secret S_B^+.
   $$S_B^+(N)$$

6. Chuck can decrypt the nonce N because he has received S_B^- from Alice and sends N to the server.

7. The server grants access to Bob because he can reply with N only if he had received the proxy from the original possessor of the rights (i.e. Alice). If Chuck had sent only Alice's rights R he wouldn't had been able to obtain N because N would have been encrypted with S^- but Chuck only has S_B^-.

Definitions

Glossary

access point An entity, characterised by an address, that allows to access another entity. 18

name An object used to reference an entity. 18

naming service A service that associate a name to an entity. 18