Distributed Systems - Lecture Notes

Davide Calabrò

September 9, 2020

Contents

1 Modelling **4**
- 1.1 Run-time architectures . . . 4
  - 1.1.1 Client-Server . . . 4
  - 1.1.2 Service Oriented . . . 4
  - 1.1.3 Peer-to-peer . . . 6
  - 1.1.4 Object-Oriented . . . 7
  - 1.1.5 Data-Centered . . . 7
  - 1.1.6 Event-Based . . . 8
  - 1.1.7 Mobile code . . . 9
- 1.2 The interaction model . . . 9
  - 1.2.1 Distributed algorithms . . . 9

2 Communication **10**
- 2.1 Layered protocols . . . 10
  - 2.1.1 OSI model . . . 10
  - 2.1.2 Middleware as a protocol layer . . . 11
- 2.2 Remote procedure call . . . 11
  - 2.2.1 Marshalling and serialization . . . 11
  - 2.2.2 Sun Microsystems' RPC . . . 12
  - 2.2.3 Distributed Computed Environment (DCE) . . . 12
  - 2.2.4 Binding client to server . . . 13
  - 2.2.5 Lightweight RPC . . . 13
  - 2.2.6 Async RPC . . . 14
  - 2.2.7 Batched vs. queued RPC . . . 14
- 2.3 Remote method invocation . . . 14
  - 2.3.1 Implementation . . . 15
- 2.4 Message oriented communication . . . 15
  - 2.4.1 Communication types . . . 16
  - 2.4.2 From network protocols to communication services . . . 16
  - 2.4.3 Message Queuing . . . 17
  - 2.4.4 Publish-subscribe . . . 18
- 2.5 Stream-oriented communication . . . 19

3 Naming **20**
- 3.1 Introduction . . . 20
  - 3.1.1 What are names? . . . 20
  - 3.1.2 Identifiers . . . 20
- 3.2 Naming schema . . . 20
  - 3.2.1 Flat naming . . . 20
  - 3.2.2 Structured naming . . . 22
  - 3.2.3 Attribute-based naming . . . 23
- 3.3 Removing unreferenced entities . . . 24
  - 3.3.1 Reference counting . . . 24
  - 3.3.2 Weighted reference counting . . . 24
  - 3.3.3 Reference listing . . . 24
  - 3.3.4 Identifying unreachable entities . . . 25

4 Consistency and replication **27**
4.1 Replication . . . 27
4.1.1 Introduction . . . 27
4.1.2 Consistency . . . 27
4.1.3 Some concept and definition . . . 27
4.2 Consistency protocols implement consistency model . . . 27
4.3 Data-centric consistency models . . . 28
4.3.1 Strict consistency . . . 28
4.3.2 Sequential consistency . . . 29
4.3.3 Leaderless protocols . . . 30
4.3.4 Linearizability . . . 31
4.3.5 Causal consistency . . . 31
4.3.6 FIFO consistency . . . 32
4.4 Synchronized models . . . 32
4.4.1 Weak consistency . . . 32
4.4.2 Release consistency . . . 32
4.4.3 Entry consistency . . . 33
4.5 Other consistency models . . . 33
4.5.1 Eventual consistency . . . 33
4.6 Summary of consistency models . . . 34
4.7 Client-centric consistency . . . 34
4.7.1 Monotonic reads . . . 35
4.7.2 Monotonic writes . . . 35
4.7.3 Read your writes . . . 36
4.8 Design strategies . . . 37
4.8.1 Replica placement . . . 37
4.8.2 Update propagation . . . 38

5 Synchronization **39**
5.1 Introduction . . . 39
5.2 Physical clocks synchronization . . . 39
5.2.1 GPS . . . 39
5.2.2 Cristian's algorithm . . . 39
5.2.3 Berkeley algorithm . . . 40
5.2.4 Network Time Protocol (NTP) . . . 40
5.3 Logical time . . . 40
5.3.1 Scalar clocks (Lamport clock) . . . 41
5.3.2 Vector clocks . . . 41
5.4 Mutual exclusion . . . 42
5.4.1 Mutual exclusion with scalar clocks . . . 43
5.4.2 Token ring solution . . . 43
5.5 Leader election . . . 43
5.5.1 Bully election algorithm . . . 44
5.5.2 Ring-based algorithm . . . 44
5.6 Collecting global state . . . 44
5.6.1 Distributed snapshot . . . 45
5.6.2 Chandy-Lamport algrithm . . . 45

6 Fault tolerance **47**
6.0.1 Introduction . . . 47
6.1 Protection against process failures . . . 48
6.1.1 Process resilience . . . 48
6.1.2 Agreement in a process group . . . 48
6.2 Reliable group communication . . . 49
6.2.1 Non-faulty processes with faulty channels . . . 49
6.2.2 Faulty processes with non-faulty channels . . . 50
6.2.3 Failure detection . . . 50
6.3 Distributed commits . . . 52
6.3.1 Two-phase commit protocol . . . 52
6.3.2 Three-phase commit . . . 52
6.4 Recovery techniques . . . 53
6.4.1 Checkpointing . . . 53
6.4.2 Logging . . . 54

7 Big data **55**
7.1 Data science . . . 55
7.2 Big data . . . 55
7.3 MapReduce framework . . . 55
7.3.1 Phases . . . 55
7.3.2 Platform . . . 56
7.3.3 Strengths, limitations and typical applications . . . 56

8 Bibliography **58**
8.1 References . . . 58

1 Modelling

1.1 Run-time architectures

1.1.1 Client-Server

Nowadays, the client-server architecture one of the most common architecture.
The main characteristic is that the components have different roles, in other words:

- Servers provide a set of services through a defined API, and most important, those API are **passive**, which means that the server is waiting for client calls.
- Users access those services through clients.
- The communication is message based.

The typical architecture is the **two tiers architecture**, which is basically composed by a server and a client and, based on the client component workload, we can distinguish between *thick-client* and *thin-client*.
Moreover, in this architecture the server can also operate like a client, for example calling another server to get some information *(e.g. calling a DB server in order to get some kind of data)*. In this case we speak about **three-tiered client-server architecture**. Obviously this concept can be extended to n tiers, so we can have **multi-tiered client-server applications**. Those applications can be classified looking at the way such services are assigned to the different tiers.

1.1.2 Service Oriented

In this architecture, the service became the most important thing.
We basically 4 concepts:

- *Services*: represent loosely coupled units of functionality
- *Service providers*: entities which exported the services
- *Brokers*: entities which hold the description of available services to be searched by interested consumers and reduce the dependency between client and server
- *Service consumers*: entities which bind and invoke the services they need and exposed by the Brokers

Strictly related with the *Service Oriented* architecture, we have the *orchestration* procedure. The *Orchestration* is the process of invoking a set of services in an ad-hoc workflow to satisfy a specific goal.
In the practice, there are many ways to implement this architecture. The most commons are:

- OGSI (Open Grid Services Infrastructure)
- JXTA
- Jini
- Web Services

Web Services

Def A software system designed to support interoperability machine-to-machine interaction over a network.

Related on the Service Oriented Architecture, we can identify the Web Services as one of the most important application.

Note that: it exposes an interface that is described by *WSDL*, which stands for *Web Service Description Language*, and it contains the set of operations exported by the web service. Moreover, web service **operations** are invoked through *SOAP*

SOAP A protocol, based on *XML*, which defines the way messages (operation calls) are actually exchanged. It's usually based on *HTTP* but other transport protocols can be used.

UDDI It stands for *Universal Description Discovery And Integration* describes the rules that allows web services to be exported and searched through a *registry*

REST

Def *REST* stands for *REpresentational State Transfer*

This style is very important for two main reasons:

- It's a nice way to describe the web
- It's a set of principles that define how Web standards are supposed to be used

The REST style is an optimal solution in order to have:

- Scalability of component interactions
- Generality of interfaces
- Independent deployment of components
- Intermediary components to reduce latency, enforce security and encapsulate legacy systems

The main constraints are:

- Interactions are client-server
- Interactions are stateless, otherwise:
  - If it's stateful there is a big dependency between client and server
  - A client can't call another server, because it needs that specific server, because it has the interactions story

  Moreover, with the stateless constraint:

  - The system becomes very scalable
  - I can introduce a caching service, in order to improve performance

- The data within a response to a request must be implicitly or explicitly labeled as *cacheable* or *non-cacheable*
- Each component cannot "see" beyond the immediate layer with which they are interacting, so we can say that the **REST is layered**.
- Components expose a uniform interface
- Clients must support *code-on-demand*, but note that this is an *optional constraint*

REST: Uniform interface constraints The uniform interface exposed by components must satisfy four constraints:

- **Identification of resources:** each resource must have an Identification (usually an *URI*) and everything that have an ID is a valid resource (including a service)
- **Manipulation of resources through representations**
  - *REST* components communicate by transferring a representation of a resource in a format matching one of an evolving set of standards data types, selected dynamically based on the capabilities or desires of the recipient and the nature of the resource
  - Whether the representation is in the same format as the raw resource, or is derived from the resource, remains hidden behind the interface
  - A representation consists of data and metadata describing the data
- **Self-descriptive messages**
  - Control data defines the purpose of a message between components, such as the action being requested or the meaning of a response
  - It is also used to parameterize requests and override the default behavior of some connecting elements
- **Hypermedia as the engine of application site** Clients move from a state to another each time process a new representation, usually linked to other representation through hypermedia links

1.1.3 Peer-to-peer

In a *peer-to-peer* applications all components play the same role, so there is no distinction between clients and servers.
But, why the *peer-to-peer* model is born? The main reasons are:

- Client-server does not scale well, due to centralization of service provision and management
- The server is also a single point of failure

- P2P leverages off the increased availability of broadband connectivity and processing power at the end-host to overcome such limitations

Moreover, *P2P* promotes the sharing of resources and services through direct exchange between peers.
Resources can be:

- Processing cycles
- Collaborative work (*i.e. Skype*)
- Storage space
- Network bandwidth (*i.e. ad hoc networking, internet*)
- Data

1.1.4 Object-Oriented

The distributed components encapsulate a data structure providing an API to access and modify it

- Each component is responsible for ensuring the integrity of the data structure it encapsulate
- The internal organization of such data structure id hidden to the other components (who may access it only through the API mentioned above)

Moreover, components interact through *RPC* and we can consider it as a *P2P* model, but it's often used to implement client-server application.
This architecture has many advantages:

- Information hiding hides complexity in accessing/managing the shared data
- Encapsulation plus information hiding reduce the management complexity, in fact we can provide some load-balancing techniques in order to share the server load.
- Objects are easy to reuse among different applications
- Legacy components can be wrapped within objects and easily integrated in new applications

1.1.5 Data-Centered

The main principle is that components communicate through a common (usually passive) repository. On the repository, data can be added to the repository or taken from it.
Access to the repository is usually synchronized and the communication with the repository is usually through RPC.

Linda *Linda* is one of the possible example of Data-Centered model.
The main characteristics are:

- Communication is persistent, implicit, content-based, generative
- High degree of decoupling
- Data is contained i ordered sequences of typed fields (*tuples*)
- Tuples are stored in a persistent and global shared space (*tuple space*)

Architectural issues

Scalability The main problem of the data-centered architectures is they're cannot be easily scaled on a wide-area, mainly because it's tricky to store and replicate tuples efficiently in such a wide area. Another problem is the routing. For the same reasons routing queries efficiently in very large system is not a simple problem to approach.

Proactive The model is proactive in the sense that processes explicitly request a tuple query, and so there's is no way to be notified where an info is available. In order to have a reactive and asynchronous behavior must implemented with an extra process and a blocking operation.

Bottleneck In general the repository is a bottleneck for the performance
In the end, in the commercial implementations, only client access to a server holding the tuple space, and, in order to add reactivity to the system, in general we add reactive primitives, such as *notify*.

1.1.6 Event-Based

The main idea is that components collaborate by exchanging information about *events*. In particular we distinguish among:

- *publishers*: components which publish notifications about the events they observe
- *subscribers*: components which subscribe to events they are interested to be notified about

In particular, the communication is:

- Purely message based
- Asynchronous
- Multicast
- Implicit
- Anonymous

1.1.7 Mobile code

It's based on the ability of relocating the components of a distributed application at run-time. In same case you can also move the entire state and not only the code. We can have multiple paradigms:

- Client-Server
- Remote evaluation
- Code on demand
- Mobile agent

The mobility can be divided in two types:

- *Strong mobility* is the ability of a system to allow migration of both code and execution state
- *Weak mobility* is the ability of a system to allow code movement across different computational environments (*e.g. JavaScript*)

This solution provides a great flexibility for add, upgrade and enrich services at run-time, but, at the same time, securing mobile code applications is tricky.

CREST *CREST* stands for *Computational REST* and it joins together the concepts of *REST* with mobile code, so, instead of *representations*, interacting parties exchanges *computations*.

1.2 The interaction model

1.2.1 Distributed algorithms

A traditional algorithm is defined as a sequence of steps, in which the process execution speed is the only variable that influence performance. In a distributed system we have also the concept of the distributed algorithm, which is difference from the traditional one, in the sense that the performance is also influenced by the transmission of messages between them. In general the behavior of a distributed algorithm is influenced by:

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

We can distinguish between:

- Synchronous DS: there are lower and upper bounds
- Asynchronous DS: there are no bounds

Note that: Any solution that is valid for an asynchronous DS is also valid for a synchronous one.

2 Communication

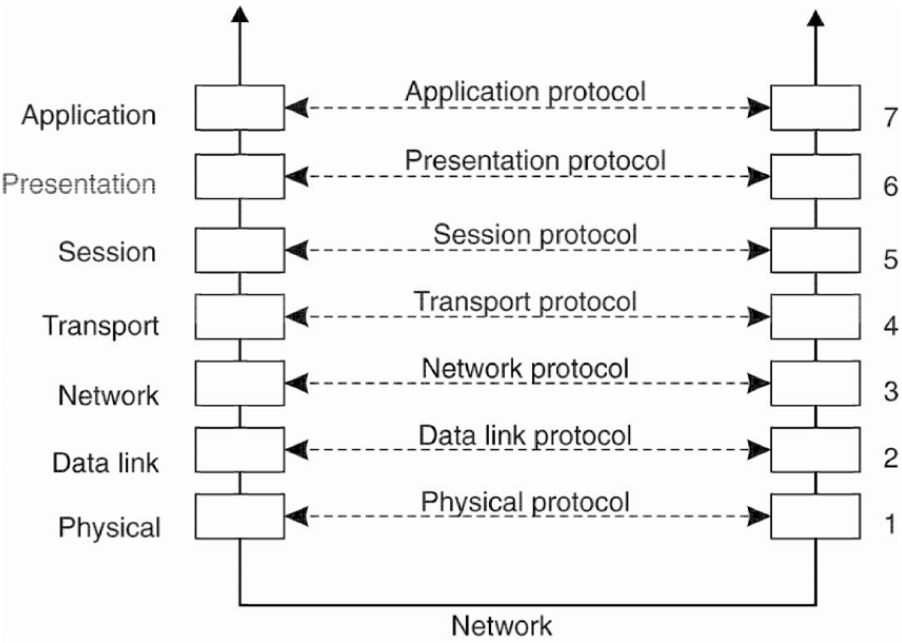
2.1 Layered protocols

The *layered protocols* are protocols in which there is a clear division between different layers. Every layer has a specific function and we can say that each layer speaks with the corresponding layer on the other side.

2.1.1 OSI model

An example of *layered protocol* is the *OSI model*. As we can see from *Figure 1*, we have several levels, each of which plays a specific role.

Figure 1: OSI model



In the *OSI model* we distinguish between three different kinds of layers.

- **Low layers**
  - *Physical layer*: it describes the bit transmission
  - *Data link layer*: it describes the organization of the series of bits into frames
  - *Network layer*: it describes how packets have to be routed
- **Transport layer**
  - It describes how data is transmitted
  - It offers a service independent from the lower layers

  - It provides the actual communication facilities for most distributed systems

  The main standards are **TCP** *(connection-oriented, reliable, stream-oriented communication)* and **UDP** *(unreliable datagram communication)*

- **Higher level layers**
  They are *session, presentation and application* layers and typically they are merged together.

Another important concept is the *encapsulation*. As the image above can suggest, every layer is encapsulated by the lower one.

2.1.2 Middleware as a protocol layer

We can consider *middleware* as a protocol layer in the sense that they include common services and protocols that can be used by different applications. In practice, it can:

- Implement *marshaling* and *unmarshaling* data procedures
- Implement naming protocols in order to allow sharing of resources
- Implement security protocol for secure communication
- Implement scaling mechanism, such as for replication and caching

---

2.2 Remote procedure call

In a **local procedure call** the parameters are passed through stack. The caller writes the parameters in the stack and then call the procedure. The called simply read the parameters from the stack since the memory is shared.
There are different ways to pass parameters to a procedure:

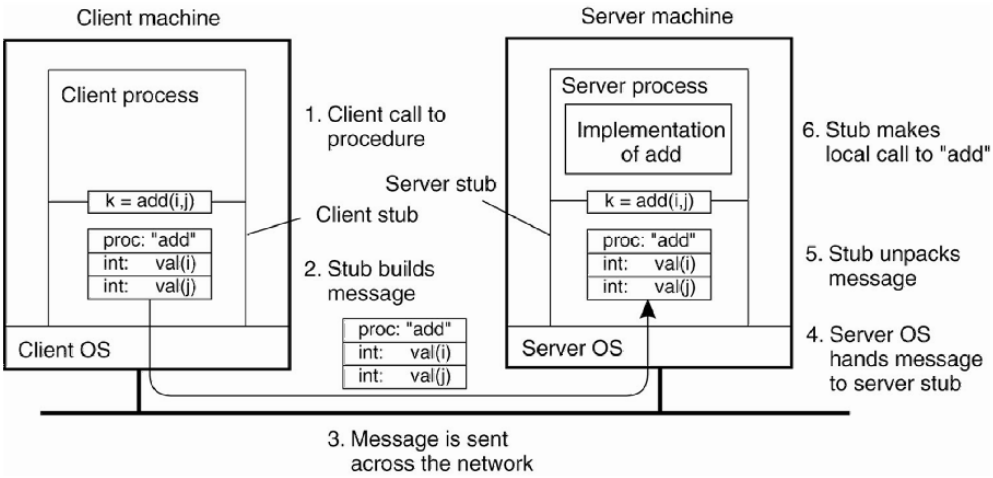
- *By value*: C-like when passing basic data types
- *By reference*: Java-like when passing objects
- *By copy/restore*: it's similar but slightly different than previous one. Value is passed in and has no effect on the value of the variable passed in until the end of the function, at which point the final value of the function variable is stored in the passed in variable. The basic difference between *call by reference* and *copy/restore* then is that changes made to the function variable will not show up in the passed in variable until after the end of the function while call by reference changes will be seen immediately.

2.2.1 Marshalling and serialization

Two problems when passing parameters:

- Structured data must be ultimately flattened in a byte stream: **serialization**
- Hosts may use different data representations *(e.g., little endian vs. big endian, EBCDIC vs. ASCII)* and proper conversions are needed: **marshalling**

Figure 2: RPC in detail



In order to simplify the procedure, these operations are done by the middleware, helped by:

- **IDL** *(Interface Definition Language)*: a language and platform independent representation of the procedure's signature
  - It raises the abstraction level of the service definition
  - The language comes with "mappings" onto target languages
  - It can also be used to automatically generate the service interface code in the target language
- A data representation format to be used during communication

2.2.2 Sun Microsystems' RPC

- Also called *Open Network Computing RPC*
- Data format is specified by *XDR (eXternal Data Representation)*
- **Parameters are passed by value**
- It can use TCP or UDP
- Security is provided through *DES*

2.2.3 Distributed Computed Environment (DCE)

- It's a set of specifications and a reference implementation
- Several invocation semantics are offered
- Several services are provided on top of RPC

  - Directory service
  - Distributed time service
  - Distributed file service

- Security is provided through *Kerberos*

2.2.4 Binding client to server

One of the main problems is find out which server provides a given service and how to establish communication with it and, obviously, it's undesirable to hard-writing this info in the client code.

Sun's solution

- Introduce a daemon called *portmap* that binds calls and server/ports
- It's important to notice that *portmap* provides its services only to local clients, i.e., it solves only the problem of establish the communication
  - In fact, the client must know in advance where the service resides
- In order to bypass this limitation, the client can send multicast message to many *portmap* daemons

DCE's solution

- The DCE daemon works like *portmap*
- Client doesn't need to know in advance where the service is: the only need to know where the directory service is
  - The directory service can be distributed
- Then, the daemon solves both the problem to find a service and establish a communication with it

Dynamic activation he server processes may remain active even in absence of requests, wasting resources. A possible solution to this problem is to introduce another (local) server daemon that:

- Forks the process to serve the request
- Redirects the request if the process is already active

With this system, however, we have a disadvantage: the first request is served less efficiently.

2.2.5 Lightweight RPC

Using a conventional *RPC*, sometimes, is not the better solution. For example, if we have a TCP/UDP service on the same machine a conventional *RPC* would lead to wasted resources. For this reason it's born the a kind of simplified *RPC*, called *lightweight RPC*.
The idea is to pass messages through local facilities, i.e. communication exploits a private shared memory region.
The *invocation* procedure is the following:

1. Client copies parameters in the shared stack and performs the system call
2. Kernel does a context switch, to execute the procedure in the server
3. Results are copied on the stack and another system call
4. Context switch brings execution back to the client

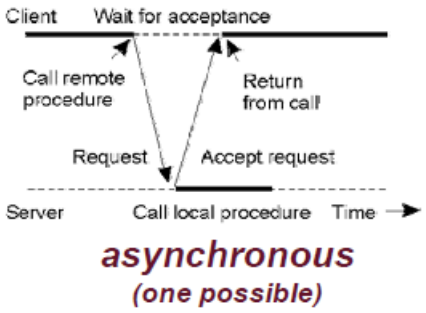
With this procedure the main advantage is that we use less threads/processes (no need to listen on a channel).

2.2.6 Async RPC

In a parallel system (or, more in general, in a non-distributed system), when a call is performed, the caller is suspended until the callee is done. *RPC* preserves this behaviour but it can, potentially, wastes client resources. For this reason we introduce the *Async RPC*, in which the caller is not suspended and it can continue in other operations.
There are many variant of *async RPC*:

- If no result is needed execution can resume after an *ack message* is received from the server

Figure 3: Asynchronous



- The callee may (async) invoke the caller back or invocation may return immediately a *promise*

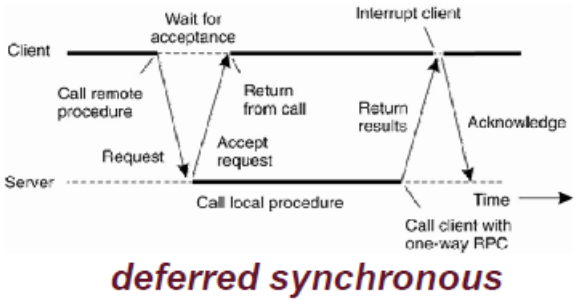
2.2.7 Batched vs. queued RPC

Sun RPC includes the ability to perform *batched RPC*. RPCs that do not require a result are buffered on the client and they are sent all together when a non-batched call is required or when a timeout expires. It's important to notice that this mechanism enables yet another form of *async RPC*.

2.3 Remote method invocation

It's an approach similar to the RPC, but it aims to obtain the advantages of OOP also in the distributed setting. The main difference is that remote object references can be passed around, so need to maintain the aliasing relationship.
Note that: sometimes, this mechanism is built on top of RPC layer.

Figure 4: Deferred synchronous



IDL The *IDL* for distributed objects are much richer because it contains also inheritance, exception handling and so on.

2.3.1 Implementation

In practice there are two main approaches:

- *Java RMI*: single language/platform
  - Easily supports passing parameters *by reference* or *by value* even in case of complex object
  - Support for code on demand (downloading code)
- *OMG CORBA*: multilanguage/multiplatform
  - It supports passing parameters *by reference* or *by value*
    - If the objects are passed by value, it's up to the programmer to guarantee the same semantics for methods on the sender and receiver sides

---

2.4 Message oriented communication

Since *RPC/RMI* fosters a sync model, this approach supports only point-to-point interaction. Moreover, synchronous communication is expensive and leads to *rigid* architectures, since there is an intrinsically tight coupling between caller and callee.

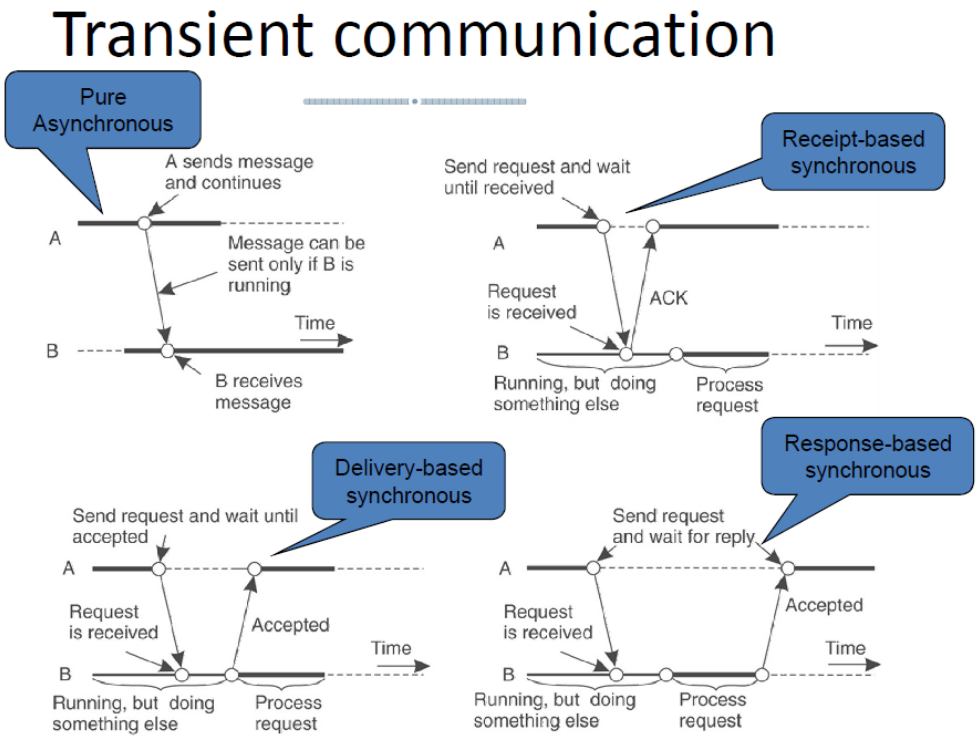
For this reason, there is another approach called **message oriented communication** that aims to solve these problems. In particular, it is:

- centered around the notion of one-way message/event
- usually asynchronous
- often supporting persistent communication
- often supporting multi-point interaction
- brings more decoupling among components

2.4.1 Communication types

- Sync vs. Async
  - *Sync*: the sender is blocked until the recipient has stored the message
  - *Async*: the sender can continues immediately after sending the message
- Transient vs. persistent
  - *Transient*: sender and receiver must both be running for the message to be delivered
  - *Persistent*: the message is stored in the communication system until it can be delivered

Figure 5: Transient communication



2.4.2 From network protocols to communication services

Since *TCP* and *UDP* are well known network protocols, the question is how to take advantages from these protocols. The answer is called **Socket**.
Sockets provide a common abstraction for inter-process communication, and allows for connection-oriented (*TCP*) or connectionless (*UDP*) communication.

Stream socket It's based on concept of connection. In fact, the server accepts connection on a port and the client connects to the server. Each connected socket is uniquely identified by 4 numbers.

Datagram socket It's based on *UDP*, so connectionless. Client and server use the same approach to send and receive datagrams, therefore both create a socket bound to a port and use it to send and receive datagrams.
There is no connection and the same socket can be used to send (or receiver) datagrams to (or from) multiple hosts.

Multicast socket It exploits the IP multicast protocol that allows to efficiently deliver *UDP* diagrams to multiple recipients.
Component interested in receiving multicast datagrams addressed to a specific group must join the group. It's important to notice that groups are open, so it's not necessary to be a member of a group in order to send datagrams to the group.
Usually, it's necessary to specify a port, in order to help operating systems to decide which process on the local machine to route packets to.

MPI The sockets are protocol independent and has low level primitives.
This is not optimal when we are handling high performance network *(e.g. clusters of computers)* where we need higher level primitives providing difference services besides pure read and write.
The **MPI** is born as answer to this need.

- Communication take place within a known group of processes
- Each process within a group is assigned a local *id*
- Messages can be sent unicast or multicast (calling the entire group)

There is no support for fault tolerance, in fact crashes are supposed to be fatal.

2.4.3 Message Queuing

In point-to-point persistent async communication we need to keep track of the messages received when we weren't running.
For this reason we must keep a queue containing the messages received, that typically guarantee only eventual insertion in the queue, so it doesn't guarantee the recipient's behaviour.
It's an intrinsically peer-to-peer architecture and each component holds an input and an output queue.

Client-server with queues We can apply this approach also to client-server architecture, where the server offers a queue in which clients put their requests. The server asynchronously fetches requests, processes them and returns results in the clients' queues.
In this way, clients need not remain connected and queues can be shared in order to simplifies load balancing.

Issues and solutions

- Since queues are identified by symbolic names, it's necessary to have a lookup service to convert queue-level addresses in network addresses.
- Queues are handled by queue managers, so they acting as relays
- Relays are often organized in an overlay network, so the messages are routed by using application-level criteria. In this way we can improve also fault tolerance.
- When integrating sub-systems, message conversion may be a problem. Message brokers can do this, providing application-level gateways supporting message conversion.

2.4.4 Publish-subscribe

The idea is that application components can publish asynchronous event notifications, and/or declare their interest in event classes by issuing a subscription.
Subscriptions are collected by an event dispatcher component, responsible for routing events to all matching subscribers and collecting subscriptions. It can be centralized or distributed (for scalability reasons). **Communication is:**

- Transiently async
- Implicit
- Multipoint

Characteristics:

- Easy to add and remove components
- Appropriate for dynamic **environments**

Expressiveness of the subscription language:

- *Subject-based:* the set of subjects is determined a priori
- *Content-based:* subscriptions contain expressions that allow clients to filter events based on their content, and the set of filters is determined by client subscriptions.

Note that: *Subject-based* and *Content-based* can be combined.

Dispatcher In a distributed architecture, the dispatcher is composed by a set of message brokers organized in an overlay network which cooperate to collect subscriptions and route messages. The topology of the network may vary: *acyclic vs cyclic*

- In an acyclic graph:
  - The messages are forwarded from broker to broker and delivered to clients only if subscribed
  - The subscribes are forwarded from broker to broker and they are sent only once in a link

  - The performance depends also from the forwarding algorithm used because each time a broker receives a message it must match it against the list of received filters to determine the list of recipients

- In a cyclic graph:
  - We can use a *DHT based approach*
  - *Content-based* routing, in which every source define a shortest path tree

Hierarchical forwarding It exploits the tree properties. In particular:

- Both messages and subscriptions are forwarded by brokers towards the root of the tree
- Messages flow *downwards* only if a matching subscription had been received along that route

Complex event processing *CEP* systems adds the ability to deploy rules that describe how composite events can be generated from primitive (or composite) ones.

2.5 Stream-oriented communication

The main concept is to send a sequence of data units in an orderly and fast way.
Transmission modes:

- *Async*: the data items in a stream are transmitted on after the other without any further timing constrains
- *Sync*: there is a max end-to-end delay for each unit in the data stream
- *Isochronous*: there is a max and min end-to-end delay

It needs some non-functional requirements *QoS*:

- Required bit rate
- Maximum delay to setup the session
- Maximum end-to-end delay
- Maximum variance in delay (*jitter*)

We can also enforcing *QoS* at the application layer:

- Buffering: control *max jitter* by sacrificing session setup time
- Forward error connection
- Interleaving data: to mitigate the impact of lost packets

3 Naming

3.1 Introduction

3.1.1 What are names?

Names are used to refer entities and these entities are usually accessed through an *access point*, that is a special entity characterized by an address. **Note that:**

- An address is just a special case of name
- An entity can be accessed by several access points at the same time
- An entity can change its access points during its lifetime

So, it's better to use *location-independent* names.
A name can be:

- *Global vs.local*
- *Human-friendly vs. machine-friendly*

3.1.2 Identifiers

Resolving a name directly into an address does not work with mobility. For this reason we introduce identifiers, which are names they never change (during the entity lifetime). Moreover, each entity has exactly one identifier and this one is never assigned to another entity.
It's important to notice that using identifiers enables to split the problem of mapping a name to an entity and the problem of location the entity.

Name resolution: it's the process of obtaining the address of a valid access point of an entity having its name

---

3.2 Naming schema

There are three main naming schemas:

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

3.2.1 Flat naming

Names are flat, so they haven't any kind of structure. In practice, they're simple strings.
There are several name resolution processes:

- *Simple solutions*: they're based on broadcast or multicast approach or through pointers forwarding

- *Home-based approaches*
- *DHT*
- *Hierarchical approaches*

Simple solutions They're designed for small-scale environments. There are three main ways to perform a simple resolution:

- **Broadcast**: send *find* message on broadcast channel and only the interested host replies. Obviously, since each host have to process the *find* message there is a traffic and computational overhead
- **Multicast**: same as broadcast, but send to multicast address to reduce the search scope
- **Forwarding pointers**: the idea is leave reference to the next location at the previous location. The main problem is that chains can became very long and broken links are fatal and, in general, there is a network latency increase. In order to reduce chain length, it can be adopted an adjusting procedure when searching an host, simply generate shortcuts to an host in order to reduce hops

Home-based approach The main idea is that one home node knows the location of the mobile unit we're looking for. In fact, this approach is used mainly in Mobile IP and 2.5G cellphone networks.

In order to work right, the home is assumed to be stable and it can be replicated for robustness and the original IP of the host is effectively used as an identifier.

There are also some problems. In particular, latency can increase of there are too many steps towards the home. Moreover, the home address has to be supported as long as the entity lives and since this address is fixed, there may be an unnecessary burden when the entity permanently moves to another location. In the end, there also a poor geographical scalability problem, since entity can be next to client.

DHT The idea is to have an hash table distributed over several nodes, organized and structured in an overlay network.

An example of DHT is *Chord*:

The main idea is to jump as far as possible in order to be closer to the possible solution. In order to do that, every node run very simple operations, because the hash tables are small.

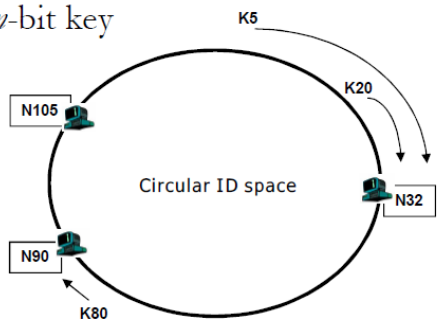
Hierarchical approach The idea is to have a tree, possibly balanced, and make a search that first look in local registry to see if the entity is around and then, if nothing has been found, contact the home.

The network is divided into domains, and the root domain spans the entire network, while leaf nodes are typically a LAN or a cell. Thus, all the nodes in the up of the tree have an huge router table. For this reason, this solution can scale but not so good.

If it happens often that the search remains in the called node, it will be efficient, otherwise we can have performance problems. We can cache the information so the next time we have

Figure 6: Chord

- Nodes and keys are organized in a *logical ring*
  - Each node is assigned a unique m-bit identifier
    - Usually the hash of the IP address
  - Every item is assigned a unique m-bit key
    - Usually the hash of the item
  - The item with key k is managed (e.g., stored) by the node with the smallest $id \geq k$ (the *successor*)



the same request, we can reply in less time than before, though caching addresses directly is generally inefficient.

In the hierarchical approach the updates start with insert request from new location. Location records are created top down or bottom up, and this one allows immediate queries.

3.2.2 Structured naming

In a structured naming system names are organized in namespaces, so we have to introduce the concept of *namespaces* which are labelled graphs composed of leaf nodes, representing a name entity, and directory nodes. In this case resources are referred through *path names*, which can be *relatives* or *absolutes*.

Name spaces for a large scale, possibly worldwide, distributed system are often distributed among different name servers, usually organized hierarchically (DNS).

The name space is partitioned into layers:

- *Global level*
- *Administration level*
- *Managerial level*

Each node of the name space is assigned to a name server, so each name server has a portion of name space.

There are also different needs in terms of performance between the various levels. For example, I can accept that a name at the global level is resolved in 2 seconds because it's stable stable and I can cache it, but the same thing does not hold fit the managerial layer, that has frequently edits and adds, so I cannot adopt cache policies.

Name resolutions approaches

- *Iterative*: the client manages every single request to every single node
- *Recursive*: the name servers manages the request to sub-servers. In this way, every layer must wait the under layer, so there are suspended threads, but we can apply some cache policies and so share the cache of different user requests

DNS Clients can request the resolution mode, but servers are not obliged to comply. For the name resolution approach, a mixture of the two is used, though global name servers typically support only iterative resolution.

In order to improve performance, caching and replication are massively used, with up-to-date procedure and TTL attribute associated to information which are useful to keep data consistent. However, transient inconsistencies are allowed.

It's important to notice that mobile entities breaks the DNS assumptions, i.e. that the global/administration levels are quite stables and the managerial levels are not. That's why phone numbers are not managed in this way.

What if a host is allowed to move?

- If it stays in the original domain, so for example from *ftp.example.com* to *example.com*, there is no problem. The only thing to do is update the database of the domain server.
- If a host moves to an entirely different domain, there are two main ways:
  - Provide an IP address of the new location: lookups are ok, but further updates no longer *localized*
  - Provide the name of the new location: it will be necessary to perform two lookups and further updates no longer affected

3.2.3 Attribute-based naming

The idea is to refer to entities not with their name but with a set of attributes. In practice, each entity has a set of associated attribute.

They are usually implemented by using *DBMS technology* and Attribute based naming systems are usually called *directory services*.

LDAP *LDAP* stands for Lightweight Directory Access Protocol

A typical approach to implement distributed directory services is to combine structured with attribute-based naming.

An LDAP directory consist of a number of records, each is made as a collection of pairs and each attribute has a type. The collection of all records in a *LDAP* is called *Directory Information Base - DIB*.

This approach leads to build a *directory information tree*, where we use typically the mandatory attributes. Each server is known as *Directory Service Agent - DSA* and each client is known as *Directory User Agent - DUA*.

In general, several DSA have to be accessed to resolve a query.

Figure 7: Attribute naming example

Attribute	Value
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Comp. Sc.
CommonName	Main server
Host_Name	star
Host_Address	192.31.231.42

Attribute	Value
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Comp. Sc.
CommonName	Main server
Host_Name	zephyr
Host_Address	137.37.20.10

3.3 Removing unreferenced entities

It's important to remove unreferenced entities, mostly in some distributed object platforms, like *RMI*. The common used system is an automatic garbage collector.

3.3.1 Reference counting

The idea of reference counting is to keep track of how many other objects have been given references. Obviously, reliability must be ensured, so exactly-once message delivery, but, since passing a reference required three messages, there will be potential performance issue in large-scale systems.

3.3.2 Weighted reference counting

In order to avoid the communication problems, we can use only counter decrements. The main idea is to have a counter that get lower and lower as you pass a reference to an host. In practice, the counter tells us how many reference are distributed in the network. If the total and partial weights are equal the object can be removed, since it doesn't have any other reference.

This method has a limitation: we cannot generate more reference than ones permitted by the total counters. In order to bypass this limitation, we can introduce an additional hop to access the target object.

3.3.3 Reference listing

The problem of the mechanisms presented above is that they don't solve the unreachability problems. The idea to solve this problem is, instead of keeping track of the number of references, keep track of the identities of the proxies.
With this method we have several advantages:

- Insertion and deletion of references must still be acknowledged, so we can send many messages with the same effect
  - Related to the point before, non-reliable communication can be used

Figure 8: An example of reference counting

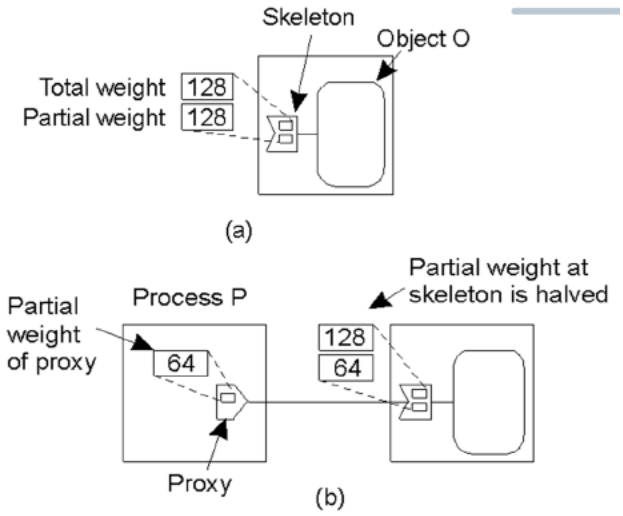
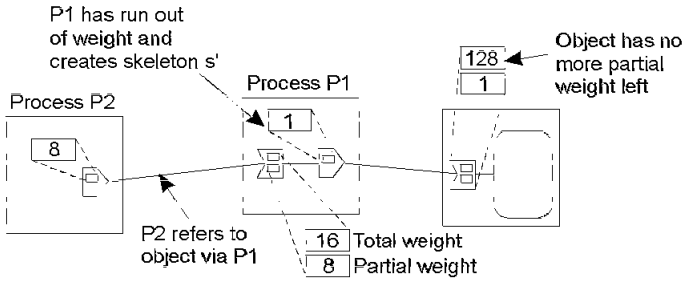


Figure 9: Reference counting in case of counter equal to 1



- It's easy to maintain the list consistent w.r.t. network failures, applying some policies (*e.g. ping*)

Note that: this method is used by *Java RMI*

3.3.4 Identifying unreachable entities

Mark-and-sweep on uniprocessor systems:

1. Marks accessible entities by following references
2. Initially all nodes white
   - A node is coloured grey when reachable from a root but some of its references still need to be evaluated
   - A node is coloured black after it turned grey and all its outgoing references have been marked grey

3. Second phase exhaustively examines memory to locate entities not marked and removes them

    - Garbage collects all white nodes

There is also a distributed version of this algorithm, in which garbage collectors run on each site.

4 Consistency and replication

4.1 Replication

4.1.1 Introduction

- Improve performance. How it does?
- Share workload in order to increase *throughput* of served requests and reduce the *latency* for individual requests
- Replicate data close to the users to reduce the *latency* for individual requests (like CDN approach). This is true not only in DS but also in parallel systems
- Fault tolerance. How it does?
- Increase availability, because data may be available only intermittently in a mobile settings
- Achieve fault tolerance: related to availability data may become unavailable due to failure

4.1.2 Consistency

The main problem in **replication** is the **consistency**. In particular changing a replica demands changes to all the other, but, if we update multiple replicas concurrently we will have *write-write* and *read-write* conflicts. It's important to notice that replication may actually degrade performance because of the overhead needed to make all this system works may have a bad influence.

Ideally, a read should show the result of the last write, but what does *last* mean? It's impossible to determine without a global clock.

4.1.3 Some concept and definition

Consistency models: a consistency model is a contract between the processes and the data store.
Guarantees on content: set the maximum "difference" on the versions stored at different replicas. *For example, if I want some macro statistics I can accept some difference, but not if I'm managing my money.*
Guarantees on staleness: set the maximum time between a change and its propagation to all replicas, in other words, I don't want data older than a certain period. *For example, for daily backups, it's possible to lose data, but at least I have a checkpoint, that in this case correspond to yesterday backup.*
Guarantees on the order of updates: constrain the possible behaviors in the case of conflicts.

4.2 Consistency protocols implement consistency model

There are different strategies for different assumptions/configurations:

Passive vs active

- **Passive**: there is only one machine that works on data. *e.g. the hard disk backup is a passive copy.*
- **Active**: there are multiple machines which work on data

Single leader vs multiple leader vs leaderless

- **Single leader**: only one write, there are no conflicts
- **Multiple leader**: there are many agents which work on the data
  - Writes are carried out at different replicas concurrently
  - There is no single entity that decides the order of writes
  - We can have *write-write* conflicts in which two clients update the same value almost concurrently. *How to solve conflicts depends on the specific consistency model*
- **Leaderless**: there is no leader, and the conflicts are managed in several ways, *for example based on an election procedure*
  - The client directly contacts several replicas to perform *writes/reads*
  - *Quorum-based protocols*

Synchronous vs. asynchronous

- **Synchronous**: the write operation completes after the leader has received a reply from from all the followers
  - We have also a variant called **semi-synchronous** where the write operation completes when the leader has received a reply from at least k replicas
  - *Sync* and *semi-sync* methods are safe, because even if $k-1$ replicas fail, we still have a copy of the data
- **Asynchronous**: the write operation completes when the new value is stored on the leader and followers are updated asynchronously

---

4.3 Data-centric consistency models

4.3.1 Strict consistency

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

Notes

- *Strict consistency* is what I want to have if I'm a developer
- Consistency I have if I think at my PC memory
- In DS *"most recent"* is ambiguous because we don't have a single clock. Anyway, this is not enough. Even if we find a solution for having a single clock, we must know we have control on our system, not on the request from users across the world
- All writes are instantaneously visible, global order is maintained
- In practice, possible on within a uniprocessor machine

4.3.2 Sequential consistency

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

Notes

- Operations within a process may not be re-ordered
- All processes see the same interleaving
- Does not rely on time

Example Here we have an example in which we have the implementation of a single leader protocol

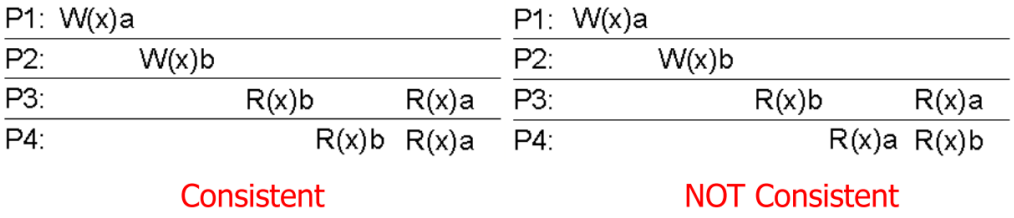


Figure 10: An example of sequential consistency

Note that: for some reasons, we don't care which, the sequential order here is b -> a, so the *image1* is consistent, because either *P3* and *P4* read the value x in the same order. This is not true in the *image2* and this is contradictory.

Why use sequential consistency? This is the best practician protocol to apply also to parallel systems *(e.g. DBMS or shared memory on multi-prcocessor computers).*

Java example Java doesn't use sequential consistency because it's a distributed system, in the sense of it doesn't share memory, like a parallel system. This is true because of the cost of implementing a sequential consistency. In fact it's necessary that all processes agree on a shared order, and this is expensive.

In practice programming languages implement sequential consistency only when is needed, that is when the variable is declared as *volatile* or there is a *synchronized block*.

Implementation

- All replicas need to agree on a given order of operations, so we need a **single leader** or a **distributed agreement**.
- The data sent is **semi-synchronous**
- Sequential consistency limits availability
  - We need to contact the leader, *that might be further away from the client*
  - The leader must propagate the update to the replicas, *in a synchronous way if we want to be fault tolerance*
  - This causes us some problems, such as *high latency*
  - Clients are blocked in the case of network partitions, because it can only proceed if they can contact the leader, and this can only proceed if it can contact followers

Assumption Sticky clients, in the sense of they access always to the same replica. This is not so useful, because they already agree on the order.

4.3.3 Leaderless protocols

Idea An update occurs only if a quorum of the servers agrees on the version number to be assigned. Moreover, reading requires a quorum to ensure latest version is being read. We define:

- N: number of total nodes
- NR: number of nodes reading
- NW: number of nodes writing

We need that those two condition are satisfied:

- $NR + NWN$
- $NWN/2$

Note that, in these conditions we are sure that, even in the worst case, there is always an overlapping on R and W.

There is also an extreme approach that anyway satisfy the above conditions, and it's called ***ROWA (read one, write all)***, and this is the opposite of the single leader approach.

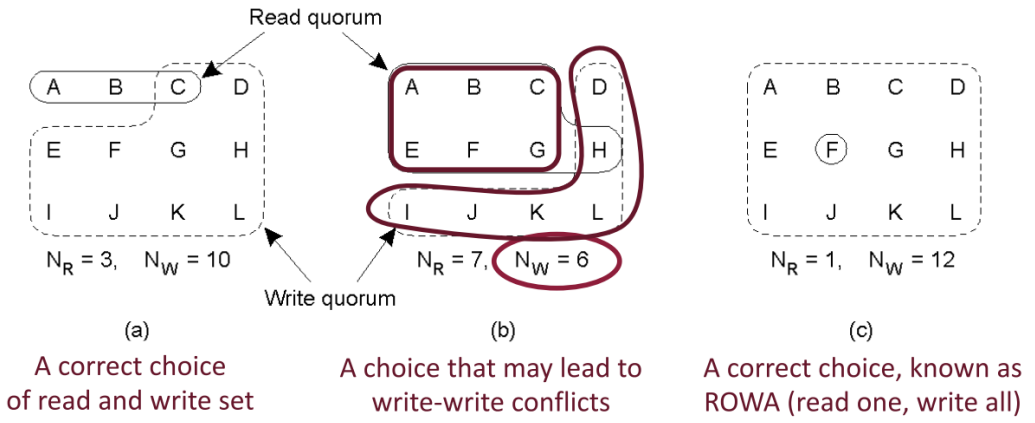


Figure 11: The ROWA procedure

4.3.4 Linearizability

Def. The system is sequentially consistent; moreover, if *tsOP1(x)* < *tsOP2(y)* then the operation *OP1(x)* precedes *OP2(y)* in the operation sequence.
Idea There is a sequence and we impose it's based on timestamp.
Assumptions We must have globally available clocks but only with finite precision.

4.3.5 Causal consistency

Def. Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in any order at different machines.
Idea I understand the meaning of a group of message because of their order.
Note that, it's the same principle on that I can understand the meaning of a message because I read another message before *(e.g. in a chat)*.

It's important to notice that I don't want to have a specific sequence for everything, but, if there are any messages with some kind of relationship I have to obtain some kind of agreement on the sequence, otherwise I can loss the relationship and not full understand the semantic.

Some definitions

Causal order: a write operation W by a process P is causally ordered after every previous operation O by the same process, even if W and O are performed on different variables.

Concurrent: operations that are not causally ordered are said to be concurrent

Implementation

Writes are timestamped with Lamport's vector clock and this implementation enabled a high degree of availability. This is the same principle I apply when a reply on WhatsApp and I'm offline. I can replay to messages which I see at the moment.

When I'm offline and I write the rest of the world will not be informed, so the writes that occur in the rest of the world will be concurrent.

4.3.6 FIFO consistency

Def. 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.

Idea I have to check only the writes of the same process which must be read in the same order

---

4.4 Synchronized models

Writes become visible only when processes explicitly request so through the variable, providing appropriate constructs. In practice, it's up to the programmer to force consistency when needed.

4.4.1 Weak consistency

Procedure

1. Access to synchronization variables is sequentially consistent
2. No operation on a synchronization variable is allowed until all previous writes have completed everywhere
3. No read or write to data are allowed until all previous operations to synchronization variables have completed

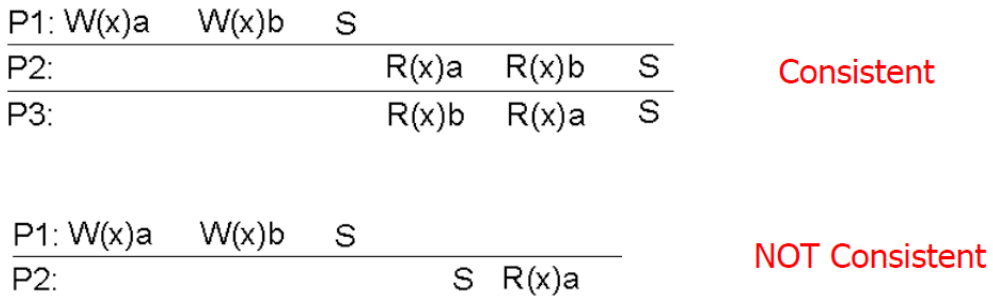


Figure 12: An example of weak consistency implementation

4.4.2 Release consistency

The previous consistency model has a problem: the data store cannot distinguish between a synchronization request for disseminating writes or for reading consistent data. A solution is to introduce different synchronization operations.

- *Acquire*
- *Release*

Procedure

1. Before a read or write is performed, all previous acquires done by the process must have completed successfully
2. Before a release is allowed, all previous reads and writes done by the process must have been completed
3. Accesses to synchronization variables are FIFO consistent

Two ways to achieve updates:

- *Eager release*: on release all updates are pushed to other replicas
- *Lazy release*: on acquire, acquiring process must get latest version from the data from other processes

4.4.3 Entry consistency

Idea Explicitly associates each shared data item with a synchronization variable.
Two ways of access to a synchronized variable

- *Non-exclusive*: multiple processes can hold read locks simultaneously
- *Exclusive*: only one process holds the lock to the variable

Procedure

1. An acquire access of a synchronization variable is not allowed to perform w.r.t. a process until all updates to the guarded shared data have been performed w.r.t. that process
2. Before an exclusive mode access to a synchronization variable by a process is allowed to perform w.r.t. that process, no other process may hold the synchronization variable, not even in non-exclusive mode
3. After an exclusive mode access to a synchronization variable has been performed, any other process' next non-exclusive mode access to that synchronization variable may not be performed until it has performed w.r.t. to that variable's owner

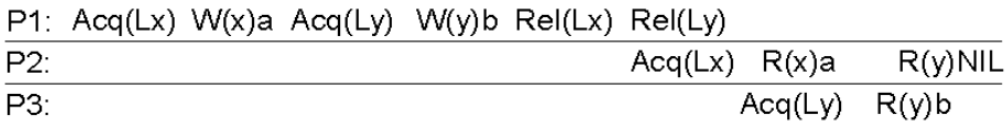


Figure 13: An example of entry consistency implementation

4.5 Other consistency models

4.5.1 Eventual consistency

Idea Eventually, you will receive updates

There are situations in which there are no simultaneous updates, or can easily resolved (*e.g. highest ID wins*) and mostly reads. In practice, this system is often sufficient: updates

are guaranteed to eventually propagate to all replicas.

An important concept related to this consistency model is the **conflict-free replicated data type**. In practice this kind of data type guarantees convergence even if updates are received in different orders. In this case we guarantee the convergence of the result.

For example, think about an *integer counter*. If I have a sequence of elementary operations, such as $+/-i$, where i is any integer, we do not care about the order in which the operations arrive, as the final result, or the counter value, will always be the same.

$$(+6)\ (-5)\ (+1) \ = \ (+2)$$
$$(-5)\ (+1)\ (+6) \ = \ (+2)$$

4.6 Summary of consistency models

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order

Consistency	Description
Weak	Shared data can be counted on to be consistent only after a synchronization is done
Release	Shared data are made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered

Figure 14: Summary of consistency models previously seen

4.7 Client-centric consistency

The previous consistency model referred to a situation in which clients are connected always to a specific replica, but *what happens if a client dynamically changes the replica it connects to*?

4.7.1 Monotonic reads

Def. 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.
Idea Once a process reads a value from a replica, it will never see an older value from a read at a different replica.

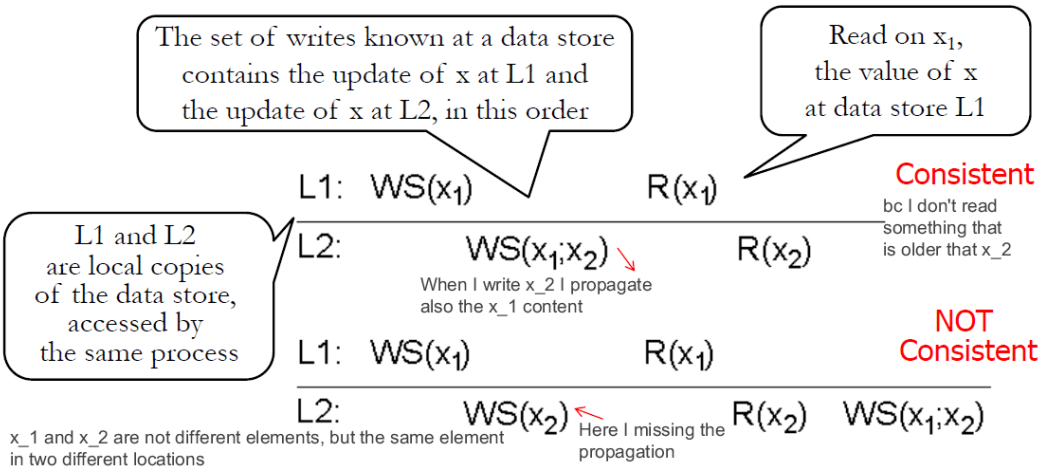


Figure 15: An example of *monotonic reads* model implementation

4.7.2 Monotonic writes

Def. A write operation by a process on a data item x is completed before any successive write operation on x by the same process.
It's similar to *FIFO consistency*, although this time for a single process.

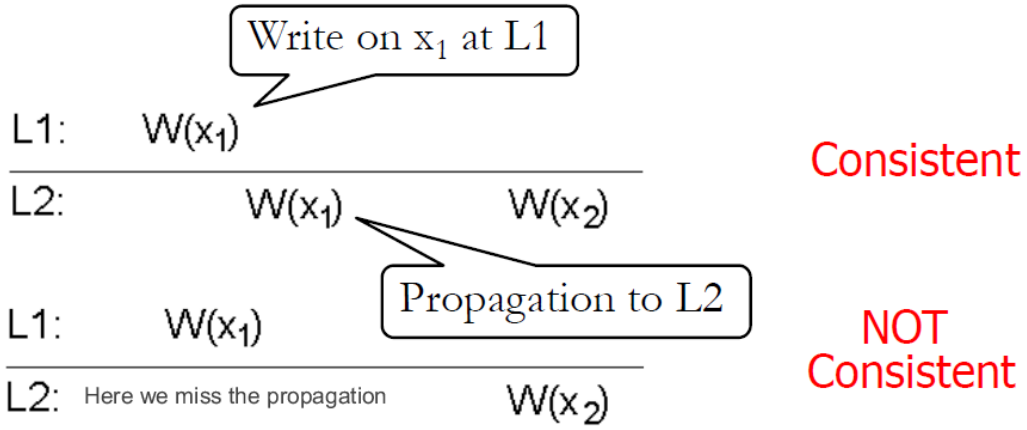


Figure 16: An example of *monotonic writes* model implementation

4.7.3 Read your writes

Def. 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.

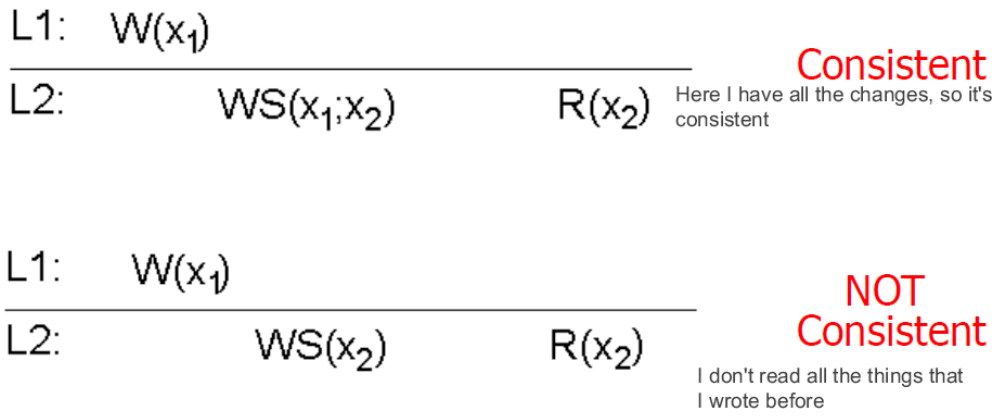


Figure 17: An example of *read your write* model implementation

Writes follow reads **Def.** 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

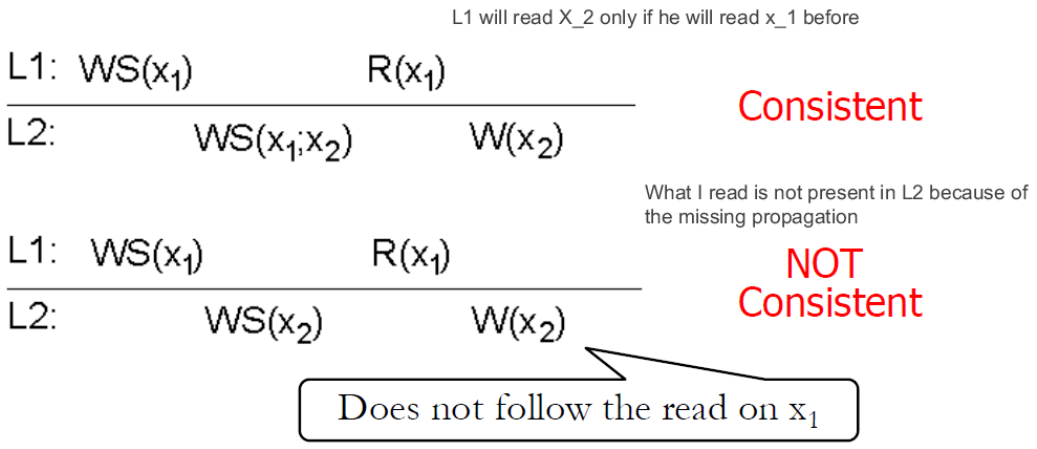


Figure 18: An example of *writes follow reads* model implementation

Implementation

- Each operation gets a unique identifier

- Two sets are defined for each client
  - Read-set: the write identifiers relevant for the read operations performed by the client
  - Write-set: the identifiers of the write performed by the client
- It can be encoded as vector clocks

4.8 Design strategies

4.8.1 Replica placement

- Permanent replicas: statically configured
- Server-initiated replicas:
  - Create dynamically
  - Move data closer to clients
  - Often require topological knowledge
- Client-initiated replicas: rely on a client cache

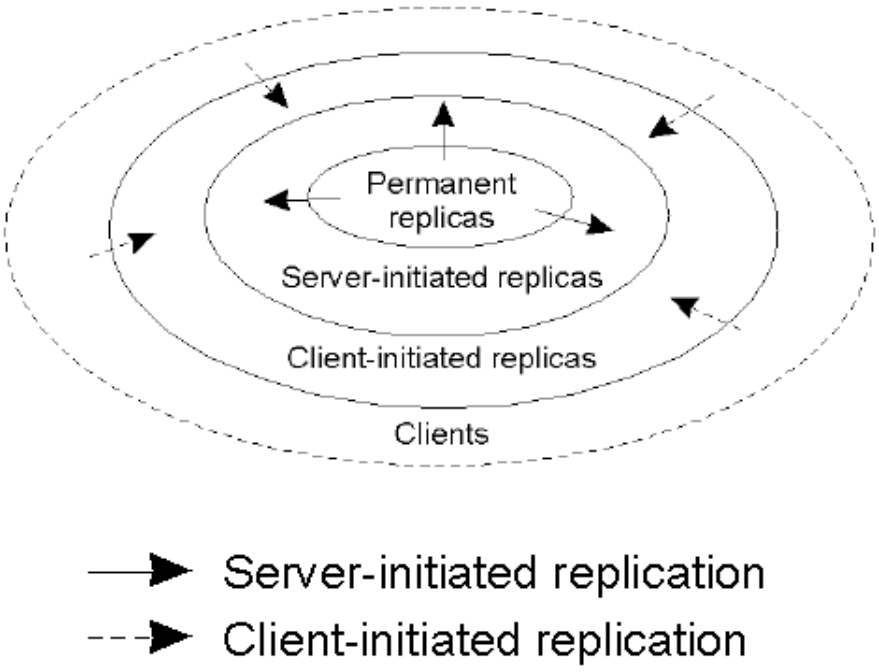


Figure 19: Replica placement placement

4.8.2 Update propagation

- Perform the update and propagate only a notification (works best if *#reads* << *#writes*)
- Transfer the modified data to all copies (works best if *#reads* >> *#writes*)
- Propagate information to enable the update operation to occur at other copies (we need to take into account side effects)

How do this?

- Push-based approach: the update is propagated to all replicas, regardless of their needs
- Pull-based approach: an update is fetched on demand when needed and this is more convenient if *#reads* << *#writes*

Propagation strategies

- Anti-entropy: server chooses another at random and exchanges updates
- Gossiping: an update propagation triggers another towards a different server, and the server reduces the probability to gossip further as the updates arrives continuously. This method is intrinsically distributed, redundant, scalable, fault-tolerant and resilient to topological changes

5 Synchronization

5.1 Introduction

In the distributed systems the natural distribution of the system complicates the synchronization, since we don't have a single local clock and/or a globally shared memory. For this reason there are some algorithms and protocols which aim to solve these problem, such as the synchronization between two or more machine clocks and the order of events in a distributed system.

5.2 Physical clocks synchronization

When we want to synchronize two or more physical clocks we have, basically, to face up with two requirements: **accuracy** and **agreement**. For *accuracy* we mean that all clocks must be synchronized against a single one, with accurate time information, while, by the term *agreement* we mean that all hosts must synchronize their clocks in the same way and with the same references between them.

When synchronizing clocks, one important thing is that temporal **monotony** must be preserved. In fact, if I discover to be ahead of time, typically I delay my timer, because it's dangerous go back with the timer because I can cause crashes.

5.2.1 GPS

One possible way to synchronize clocks is based on *GPS (Global Position System)* mechanism. In other words, the idea is to exploit the *GPS* procedure used to calculate the position of an object and, since the position is calculated by a triangulation procedure, that exploit the delays of signals to measure distance, it's possible to obtain also the perfect time.

In the practice, we need a fourth equation, and so a fourth satellite, to obtain, as well as the three coordinates, the perfect time too. Obviously the procedure is more complex since, in the previous description, we did some simplifications (*e.g. the signal propagation speed is not constant*), but, however, this method is enough accurate, with a accuracy of few tens of nanoseconds.

5.2.2 Cristian's algorithm

The Cristian's algorithm was proposed by Flaviu Cristian in 1989 as a method for clock synchronization. The implementation is pretty simple and it's used, typically, in clock synchronization in distributed computer systems within small and low-latency intranets. The idea is to send a message to the server that will respond with its current time and trying to fix it adding the approximation of the time needed to send the message back from the server to the client.
Obviously, in order to fix the time received by the server we have to assume that the go and go back times are the same, or at least similar. In this way, the average, and so the temporal correction, will be pretty accurate.

The time sets by the client will be:

$$T_c = T_s + T_{round}/2$$

where T_s is the time provided by the server, while T_{round} is the time taken by the request for go to and from the server. If the previous assumption - *go and go back times are the same* - doesn't hold, we commit an error. More precisely, we put the time in advance or in delay based on which message is faster.

Typically, the server to which request is sent is a machine that has some kind of info about the precise time, for example through a *GPS procedure*.

5.2.3 Berkeley algorithm

The Berkeley algorithm was developed at the University of California, Berkeley in 1989 as method for clock synchronization in distributed computing. Also here, the idea is pretty simple: a time server collects the time from all clients, average it, and then send to clients the required adjustment. As you may have noticed, the goal of this algorithm is not provide to the clients an absolute precise global clock, but only synchronize several machines in the net in order to share a common clock among them. We can also optimize this algorithm with the Cristian's algorithm.

5.2.4 Network Time Protocol (NTP)

Network Time Protocol (NTP) is a networking protocol for clock synchronization, used in practice over Internet, so used in nets with variable latency. The idea is to have a hierarchy organized in several layers, where the top level hosts are connected with an *UTC* source and where the time information is propagated downward in the tree.

There are three main synchronization mechanisms:

- *Multicast (over LAN)* in which servers periodically multicast their time to other machines in the network
- *Procedure-call mode* that is a procedure similar to Cristian's algorithm, in the sense that as well as sending the server clock, it estimates the offset and its accuracy.
- *Symmetric mode* that is used for higher layers which need the highest accuracy

5.3 Logical time

In many practical cases, it's sufficient to agree on a time, even if it's not accurate w.r.t. the absolute time, in fact, in these cases, we are interested about the order, so if an event is occurred before or after someone else.
In order to better understand the logical time purpose, let's introduce the **happens-before** relationship.
Let's consider two distinct events, e and e'. The two events are in **happens-before** relationship if at least one of these two conditions holds:

1. If events e and e' occur in the same process and e occurs before e', then $e \to e'$
2. If $e = send(msg)$ and $e' = recv(msg)$, then $e \to e'$

Properties

- The relationship is **transitive**
- If neither $e \to e'$ nor $e' \to e$, the two events are concurrent $e||e'$

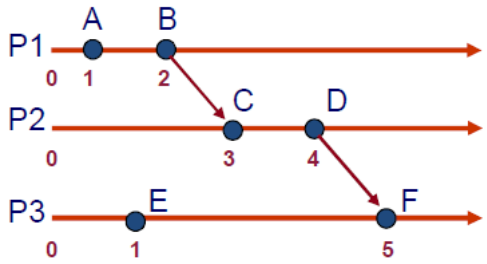
5.3.1 Scalar clocks (Lamport clock)

Leslie Lamport created a procedure in which we can capture the *happend-before* relationship numerically by using integers to represent the clock value and have no relationship with a physical clock whatsoever.

Implementation

- *Initial condition*: every process has an integer counter initialized at 0, that we will call L_i
- Every time a message is sent by process i, this is timestamped with L_i
- L_i is incremented:
  1. Before process i sends a message
  2. Upon receipt of a message, process i sets L_i to $\texttt{MAX}(ts(msg), L_i) + 1$, where $ts(msg)$ is the timestamp contained in the message received.

Figure 20: An example of Lamport clock implementation



Note that, this procedure give us only a partial order. In order to achieve a total order we can simply attach the process ID to the integer counter, so, for example, counter 1.3 tells us that the current counter value is 1 in process 3. It's important to notice that achieve a total order doesn't mean there are no parallel events.

5.3.2 Vector clocks

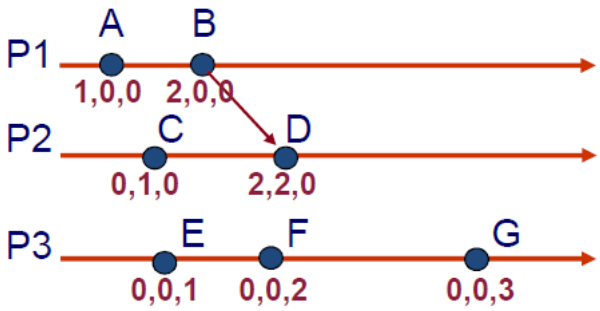
In scalar clocks, from the *happens-before* relationship we know that $e \to e' \implies L(e) < L(e')$ but, as we can see, this is an implication, not a double implication, so the reverse doesn't necessary hold. In order to solve this problem there were introduced the vector clocks.

Implementation In vector clocks each process maintains a vector of N values (where N is the number of processes), such that every process has a snapshot of the situation of the other processes.
So, let's define the vector $V_i[N]$ as the vector of the process i. Each process i knows something about the situation of other processes, so if we have $V_i[j] = k$, this means that process i knows that k events occurred in process j.

- *Initial condition*: every process has a vector where all cells are initialized at 0, so $V_i[j] = 0 \forall i, j$

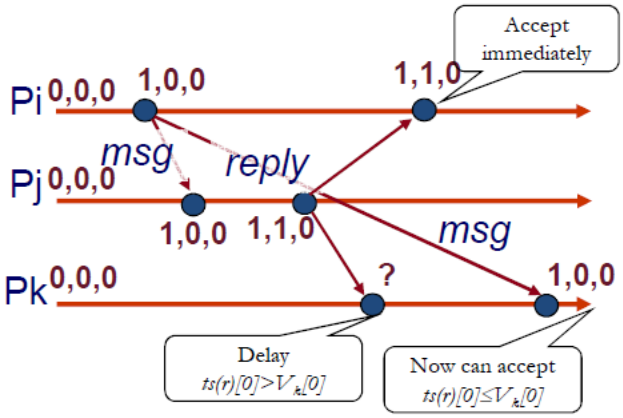
- Every time a message is sent by process i, this is timestamped with L_i
- Increments:
  1. $V_i[i]$ is incremented just before process i sends a message
  2. Upon receipt of a message, process i sets $V_i[j] = \texttt{MAX}(ts(msg), V_i[j])\ \forall i \neq j$, where $ts(msg)$ is the timestamp contained in the message received, and then increments $V_i[i]$.

Figure 21: An example of vector clock implementation



Causal delivery We can also use vector clocks, with slight variations, in order to implement causal delivery of messages in a totally distributed way. In practice, we increment the clock only when sending a message and, on receive, we just merge, without any increment.

Figure 22: An example of vector clock implementation



5.4 Mutual exclusion

The **mutual exclusion** aims to prevent interference and ensure consistency of resource access among several processes. In order to face up with this problem we assume to have

reliable channels and processes. Moreover, the mutual exclusion has to satisfy two main requirements: **safety**, which means than only one process may execute in the critical section at a time, and **liveness**, which means that all requests to enter/exit the critial section eventually succeed.

5.4.1 Mutual exclusion with scalar clocks

The procedure is pretty simple and it exploits the scalar clock methodology with the goal to use timestamps as a metric for resource assignment.

In details, a process multicasts a resource request message to all processes, including its timestamp. When processes receive the message, they have three options:

1. If it doesn't hold the resource and it's not interested to hold it, simply replies with an ACK
2. If it holds the resource, puts the requests into a local queue ordered according to timestamps
3. If it doesn't hold the resource, it's interested to hold it but it has already sent a request message, it replies with an ACK if the incoming request message has a lower timestamp, otherwise put it on a local queue

When releasing a resource, a process acknowledges all the requests queued and the resource is granted to a process when its request has been acknowledged by all processes.

5.4.2 Token ring solution

Another possible solution used to provide mutual exclusion is the **token ring**. In order to apply this mechanism, the processes must be logically arranged in a ring (don't care if their physical connections are not arranged in a ring). The idea is that only the owner of the token can access a certain resource. The token is forwarded from process to process through the ring network. When the token is received by a process P can perform three different behaviours:

- If P is not interested in accessing the resource, it forwards the token
- If P is interested in accessing the resource, P retains the token and accesses the resource.

When operations on the resource are finished, this one is released and the token forwarded to the next process.

5.5 Leader election

We know that many distributed algorithms and procedures need a coordinator, but in this case we have the agreement problem, that is, all processes in the systems must agree on a new leader.

In order to do it, we have to do some assumptions. First of all **nodes must be distinguishable** and, moreover, the **system must be closed**, in the sense that processed know each other and their IDs.

5.5.1 Bully election algorithm

The idea of this algorithm is that, once detected the coordinator failure, a new election starts. If the algorithm runs right, at the end, the process with the highest ID will be elected as coordinator, hence the name of the algorithm. In order to make this algorithm works right, we will have to do some additional assumptions: **links are reliable** and **it's possible to detect the failed process**.

Procedure

1. Process P detects that the coordinator is no more available and so it starts a new election by sending an *ELECT* message, including its ID, to all other processes with higher IDs.
2. When a process P' receives an *ELECT* message it responds to the sender and starts a new election, if it has not already started.
3. If no one responds to a process P'' that has sent an *ELECT* message, P'' sends a *COORD* message to all other processes in the net with a lower ID.

It's important to notice that if the previous coordinator process can hold a new election when it comes back up, and so take back the role of coordinator, since we assume that, having been a coordinator, has a higher ID than the current coordinator.

5.5.2 Ring-based algorithm

This algorithm is based on the assumption that nodes (processes) are arranged in a physical or logical ring network. The procedure is quite similar w.r.t. the *bully election algorithm*.

Procedure

1. Process P detects that the coordinator is no more available and so it starts a new election by sending an *ELECT* message, including its ID, to the next alive neighbor.
2. When a process P' receives an *ELECT* message it checks if its ID is in the list. If so, it change the message type to *COORD*, otherwise it adds its own ID to the list. In both cases the new message is forwarded to the next alive neighbor.
3. When a process P' receives a *COORD* message it takes the highest ID in the list and set it as coordinator.

Note that, when the *COORD* message is received, the host is also informed about the remaining members of the ring and if multiple processes starts a new election procedure, the result will converge to the same content and so to the same (new) coordinator.

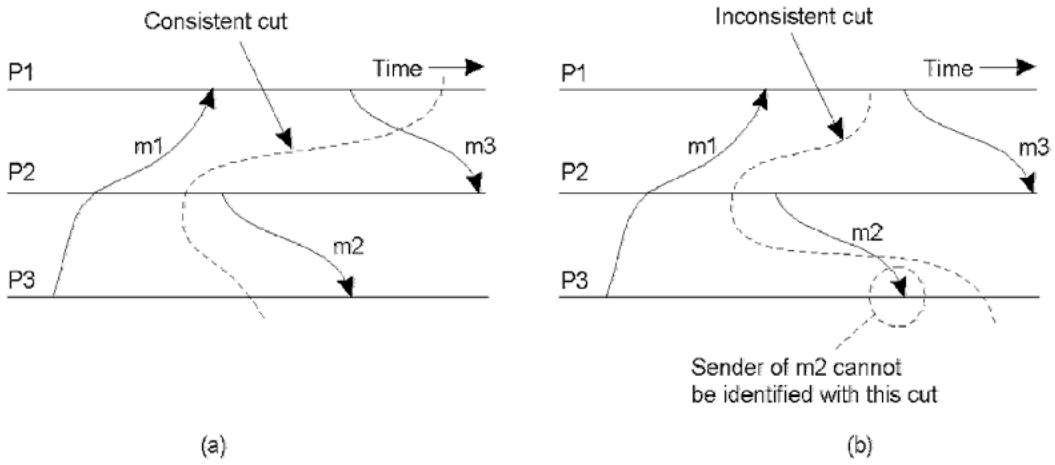
5.6 Collecting global state

Since we have a distributed system, a global state consists of the local state of each process and the messages in transit over the links. It's important to notice that collecting a global state is easier if we have a global clock, but in distributed systems we don't have it, so, potentially, we have to record the process states in different times and merge the information in such a way that we have a consistent global state.

5.6.1 Distributed snapshot

A **distributed snapshot** reflects a state in which the distributed system might have been and obviously this snapshot must be *consistent*. Moreover, we introduce also the *cut* definition. A **cut** is the union of the histories of all processes in a system up to a certain event. Note that, a cut defined as before can also be not consistent *(for example, look at the figure 23)*. So, we define also the *consistent cut*. A **consistent cut** is a cut in which for every event e included, the cut includes also all the events that happened before e. In other words, when a message received on process P' is recorded, the message sending on process P must be recorded as well, but the contrary is not required.

Figure 23: An example of consistent and not consistent cut



5.6.2 Chandy-Lamport algrithm

Chandy-Lamport algorithm is a snapshot algorithm that is used in distributed systems for recording a consistent global state of an asynchronous system. In order to perform this algorithm we assume we have reliable links and nodes (processes) and a FIFO policy concerning the queue transit method.

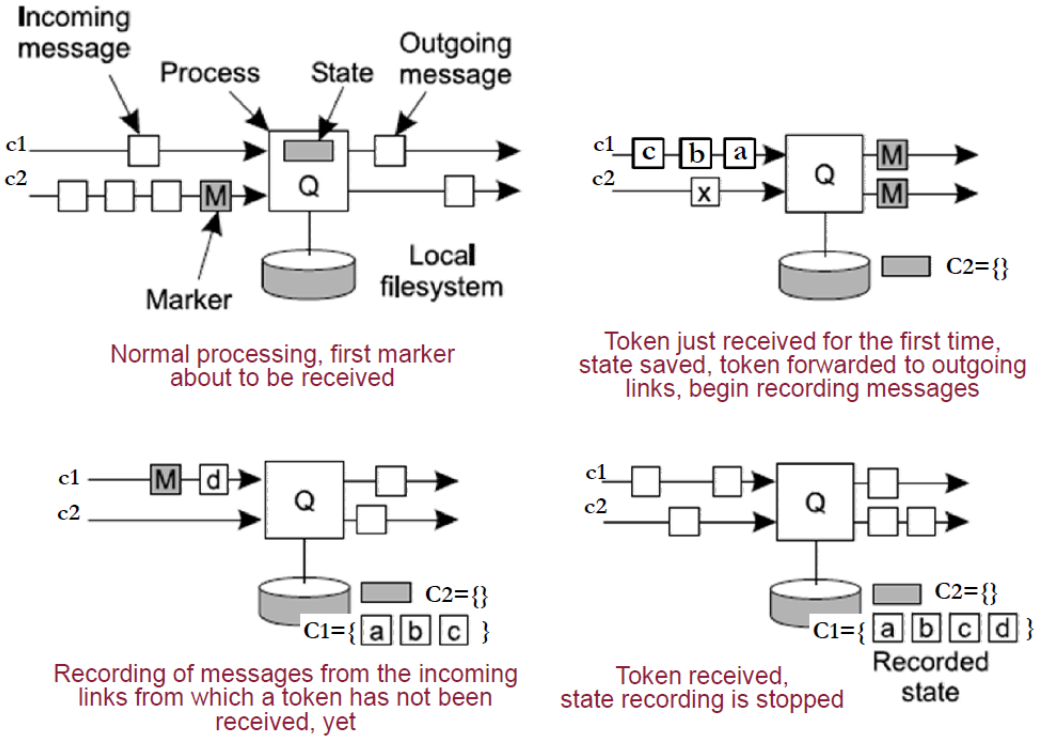
Procedure A process P runs the algorithm when a specific token is received or, in alternative, when process P itself decides to start the procedure.

- The observer process (the process taking a snapshot):
  1. Saves its own local state
  2. Sends a snapshot request message bearing a snapshot token to all other processes
- A process receiving the snapshot token for the first time on any message:
  1. Sends the observer process its own saved state
  2. Attaches the snapshot token to all subsequent messages (to help propagate the snapshot token)

- When a process that has already received the snapshot token receives a message that does not bear the snapshot token, this process will forward that message to the observer process

Note that, it's not required to block the computation, in fact, the operation of collecting the snapshots is interleaved with processing. Moreover, we can start the *distributed snapshot* in different locations at the same time, since we can associate a specific identifier to each snapshot in order to identify them.

Figure 24: An example of a Chandy-Lamport algorithm execution



Theorem The distributed snapshot algorithm selects a consistent cut.

Proof

Let e_i and e_j be two events occurring at process p_i and p_j, respectively, such that $e_i \rightarrow e_j$. Suppose e_j us part of the cut and e_i is not. This means e_j occurred before p_j records its state, while e_i occurred after p_i saved its state.

If $p_i = p_j$ this is trivially impossible.

If $p_i \neq p_j$, let's consider $m_1, ..., m_h$ be the sequence of messages that give rise to the relation $e_i \rightarrow e_j$.

If e_i occurred after p_i saved its state then p_i sent a marker ahead of $m_1, ..., m_h$. By FIFO ordering over channels and by the marker propagating rules it results that p_j received a marker ahead of $m_1, ..., m_h$. By the marker processing rule it results that p_j saved its state before receiving $m_1, ..., m_h$, i.e., before e_j , which contradicts our initial assumption.

6 Fault tolerance

6.0.1 Introduction

Building systems that depends on **availability**, **reliability**, **safety**, **maintainability**.
A **failure** is the result of an *error*, that is a caused by a *fault*.
A system is said to be **fault tolerance** if it can provide its services even in the presence of faults.
Faults are of three types:

- *Transient faults*: faults which occur once and disappear
- *Intermitted faults*: faults which appear and disappear with no apparent reason (worst faults)
- *Permanent faults*: faults which continue to exist until the failed components are repaired

Failures are of three types:

- *Omission failures* (e.g. crash)
- *Timing failures* (e.g. timeout latency of a channel)
- *Byzantine* (continue executing but something different from you expect)

Redundancy is best way to solve these problems. We can have several types of redundancy:

- *Information redundancy* (e.g. Hamming codes)
- *Time redundancy* (i.e. try again)
- *Physical redundancy*

Let's take the client-server architecture as an example. In the client-server architecture nobody tells me if the server crashed and at what point of their procedure is crashed. There are many strategies useful to bypass this problem and obtaining a result, but it's important to highlight that never I can guarantee exactly one result, such as in the *printer client-server example*. All strategies turn out to be suboptimal, since I can get at most one or at least one result.

When clients crash the computation running on server called by that clients are said to be **orphan**. In this case the server can have several approaches:

- *Extermination*: RPC's are logged by the client and orphans killed by a client request after a reboot (costly because I have to keep all logged and there are also problems with network partitions)
- *Reincarnation*: when a clients reboots it starts a new epoch and sends a broadcast message to servers, who kill old computations started on behalf of that client
- *Gentle reincarnation*: as before but servers kill old computations only if owner cannot be located
- *Expiration*: remote computation expire after some time and so, clients wait to reboot to let remote computations expire

---

6.1 Protection against process failures

6.1.1 Process resilience

Let us assume a context in which channels are reliable but process are not. In this context **Redundancy** can be used to mask the presence of faulty processes, with redundant process groups. In practice, the work that should be done by a process is taken care by a group of processes, and so the healthy processes can continue to work when come of others fail.
Two possible organizations:

- *Flat group*
- *Hierarchical group* with workers and a coordinator

However, using groups is not so easy because we must keep track of processes composing a group and, moreover, having a coordinator seems easier but it's a single point of failure. Finally, it is important to notice that if too many processes crash or leave, the group has to be rebuilt.
When we have a group it's also crucial to understand how large a group has to be:

- If processes fail silently, then *k+1* processes allow the system to be *k-fault tolerant*
- If failures are Byzantine, we need *2k+1* processes in order to achieve *k-fault tolerance* and also have a working voting mechanism.

6.1.2 Agreement in a process group

By *agreement in a process group* we mean the reaching a decision among all non-faulty processes, based on the initial values.
In order to reach an *agreement in a process group*, these properties must hold:

- *Agreement:* no two processes decide on different values
- *Validity*: if all processes start with the same value, then that value is the only possible decision value
- *Termination*: all non-faulty processes eventually decide

When the communication channels are not reliable it's not possible to reach a consensus, but *it's possible when we have communication channels reliable and non-reliable processes.*

FloodSet Algorithm
Procedure

- Let v_0 be a pre-specified default value
- Each process maintains a variable W *(subset of V)* initialized with its start value
- For $f + 1$ rounds:
  - Each process sends W to all other processes
  - It adds the received sets to W
- After $f + 1$ rounds:
  - if $|W| = 1$: decide on W's element, so all processes agree on the same value

– if $|W| > 1$: decide on v_0 or, in alternative, use a common function to decide *(e.g. $min(W)$)*

We notice that with this algorithm, each process needs to know the entire set W only if $|W| > 1$. To improve this algorithm we can broadcast W at the first round and we go on only if $|W| > 1$.

Agreement in a process group in present of byzantine faults
If we have a process group in which there may be byzantine faults the properties change:

- *Agreement:* no two **non-faulty** processes decide on different values
- *Validity*: if all **non-faulty** processes start with the same value, then that value is the only possible decision value
- *Termination*: all non-faulty processes eventually decide

This problem can be formalized as the generals problems with traitors. Lamport (1982) showed that if there are m traitors, $2m + 1$ loyal generals are needed for an agreement to be reached, for a total of $3m + 1$. Moreover, *it's proved that is not possible to reach an agreement if there is one faulty process in an asynchronous system.*

6.2 Reliable group communication

Let us assume the following properties:

- Groups are fixed
- Processes are non-faulty
- Channel is faulty

In order to achieve a *reliable group communication* we can have two strategies:

- Positive acknowledgements: we send every time *ACKs (Pessimist approach)*
- Negative acknowledgements: we send *NACKs* when something goes wrong *(Optimist approach)*

Note that, the second approach has a problem: *how the client know that him has to receive a packet and so send a NACK?*

6.2.1 Non-faulty processes with faulty channels

Scalable reliable multicast The main *idea* is that if a process discovers that something went wrong, notify the error to the other receivers in the net and then also to the sender.
More in detail, when an host discovers that something went wrong, *e.g. it didn't receive a packet*, it notifies the error to all the other receivers with an *NACK* message in such a way to cause the suppression their feedback.
Once do that, the host sends a *NACK* to the sender after waiting for the timeout to expire. The timeout is indicated by the letter T in the image below and it's set randomly. In this way the sender receives only one *NACK*.
Note that, the situation doesn't change if two ore more hosts has the same *random delay* timeout.

Hierarchical feedback control The main *idea* is that receivers are organized in groups headed by a coordinator and groups are organized in a tree routed at the sender.

More in detail, the coordinator takes care about its group and it can remove a message from its buffer if it has received an ACK from:

- All the receivers in its group
- All of its child coordinators

The main *problem* here is that the hierarchy has to be built and maintained.

6.2.2 Faulty processes with non-faulty channels

If we have faulty processes and non-faulty channel, things becomes harder. In this case we have to face with the *atomic multicast problem*, that is, a message must be delivered either to all the members of a group or to none, and that the order of the messages be the same at all receivers.
The typical example is the order of command in a database transaction.
Ideally we want to achieve **close synchrony**:

- We want that 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
- We want that a multicast to a process group is delivered to its full membership

However, *close synchrony* cannot be achieved in the presence of failures.

6.2.3 Failure detection

In most of systems, a failure detection mechanism is needed in order to understand if something went wrong during the execution of a procedure. However, it's important to notice that, even if we detect a failure we cannot know whether a failed process has received and processed a message. With this introducation, the model becomes:

- Crashed processes are purged from the group and have to join again
- Messages from a correct process are processed by all correct processes
- Messages from a failing process are procced either by all correct members or by none
- Only relevant messages are received in a specific order

Virtual synchrony It's a form or reliable multicast in which group view changes are delivered in a consistent order with respect to other multicasts and with respect to each other.

In order to better understand what *virtual synchrony* is, let us introduce some definitions:

- **Group view:** set of processes to which a message should be delivered as seen by the sender at sending time
- **View change**: it happens when a process joins or leaves the group, possibly crashing

In other words, all multicasts must take place between view changes, so that, a multicast make sense when it's received before that the "news" that a process crashed arrive.
In practice:

- We must guarantee that messages are always delivered before or after a view change
- If the view change is the result of the sender of the message m leaving, the message is either delivered to all group members before the view change is announced or it's dropped.

Generally speaking, multicasts take place in epochs separated by group memebership changes.

Retaining the virtual synchrony property, we can identify different orderings for the multicast messages. In particular we can have:

- Unordered multicasts
- FIFO-ordered multicasts
- Causally-ordered multicasts

In addition, the above orderings can be combined with a *total ordering* requirement, so whatever is the chosen order, messages must be delivered to every group member in the same order. Note that, *virtual synchrony* includes this property in its own definition.
Moreover, a virtual synchronous reliable multicast ordering totally-ordered delivery of messages is defined as **atomic multicast**.

Implementation Over the years, several implementations of *virtual synchrony* have been proposed, for example the one implemented in *ISIS*, that is a fault tolerant distributed system.

However, the basic idea is composed by few key points:

- We assume that processes are notified of view changes
- When a process receives a view change message, stop sending new messages until the new view is installed, instead it multicasts all pending unstable messages to the non faulty members of the old view, marks them as stable and multicasts a flush message
- Eventually, all the non faulty members of the old view will receive the view change and do the same. Obviously, view changes duplicates are discarded.
- Each process installs the new view as soon as it has received a flush message from each other process in the new view.

After all these points are met, processes can restart sending new messages.
In conclusion, if we assume that the group membership does not change during the execution of the protocol above we have that at the end all non faulty members of the old view receive the same set of messages before installing the new view.

Example

1. Process 4 notices that process 7 has crashed and so it sends a view change
2. Process 6 sends out all its unstable messages, followed by a flush message
3. Process 6 installs the new view when it has received a flush message from everyone else

---

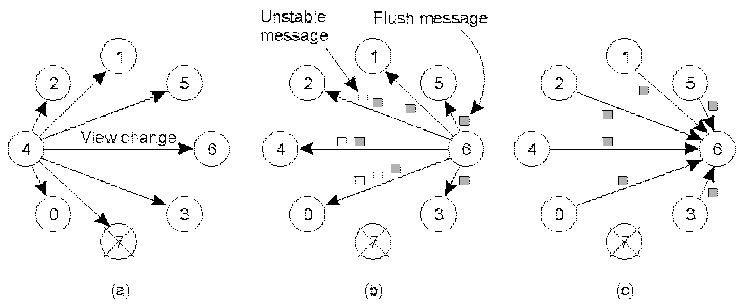


Figure 25: An example of view change

6.3 Distributed commits

Commit protocols are a very common way to achieve agreement, so that, two processes agree on the same value.
One important thing we have to guarantee in a commit protocol is the **termination** property. In particular we can have two types of termination:

- *Weak:* if there are no faults, all processes eventually decide
- *Strong:* all non-faulty processes eventually decide

6.3.1 Two-phase commit protocol

Two-phase commit protocol is a blocking protocol, it satisfies the weak condition and it's allow to reach termination in less than *f+1* rounds.

If a participant fails, after a timeout a coordinator can assume abort decision by participant.
If the coordinator fails:

- If participant blocked waiting for vote request, it can decide to abort
- If participant blocked waiting for global decision, it cannot decide on its own but it mast wait for the coordinator to recover *(blocking protocol)*

6.3.2 Three-phase commit

Three-phase commit protocol is a non-blocking protocol and it satisfies the strong termination condition. Moreover, it attempts to solve the problems of two-phase commit protocol by adding another phase in such a way that no state leads directly to *COMMIT* and *ABORT* and *COMMIT* state cannot be reached if agreement is impossible to achieve.

Note that, the *READY* state is the crucial one when coordinator abort. In this case others can be in *INIT*, *READY* or *PRECOMMIT*, but no one can be in *COMMIT* and so, in case it's impossible to achieve an agreement, I can always *ABORT*.

However, in case of coordinator failure, things are more complex, in the sense that I can have apply different behaviours depending on the situation I have in my system. In particular, we can have these cases:

- If we have a participant blocked waiting for vote request, it can decide to *ABORT*

- If we have a participant blocked waiting for global decision, it contacts other participant and then decide what to do:
  - At least one in *ABORT* state: *ABORT*
  - At least one in *COMMIT* state: *COMMIT*
  - At least one in *INIT* state: *ABORT*
  - Majority in *PRECOMMIT* state: *COMMIT*
  - Majority in *READY* state: *ABORT*

  Note that, no two participants can be in *PRECOMMIT* and the other in *INIT* state. Moreover, in alternative, it's possible to elect a new coordinator.

6.4 Recovery techniques

When processes resume working after a failure, they have to be taken back to a correct state.

There are two types of recovery:

- The system is brought back to a previously saved correct state (*backward*)
- The system is brought into a new correct state from which execution can be resumed (*forward*)

Recovery a previous state is possible only if there is a checkpoint or a logging system.

6.4.1 Checkpointing

We need to find a (possibly the most recent) consistent cut. The idea is going back until we find a couple of checkpoints which give us a consistent checkpoint, in some kind of domino effect.

One idea is to have an **independent checkpointing**, where every process records its own state. However, independent checkpoint is not trivial to implement, because interval between two checkpoints is tagged and each message exchanged by the processes must be recorded as a reference to the interval. It's important to notice that in this way the receiver can record the dependency between receiving and sending intervals with the rest of the recovery information.

When a failure occurs, the recovering process broadcast a dependency request to collect and compute dependency information. All processes stop and send their information to the recovering process, that computes the recovery line and sends the corresponding rollback request.

The recovery line can be computed using two different techniques, both ending to the same result:

- **Rollback-dependency graph**: the main idea of this technique is to propagate the crash backward. In this method we replace dependencies between messages with dependencies with checkpoints. One computed these dependencies, we start from the crash events and go back through the dependencies, represented by directed arrows, and we invalidate checkpoints that has some kind of dependency with some *"no more*

valid" checkpoint. When we cannot invalidate any other checkpoints we obtain the solution represented by the right-most valid checkpoint available for each process.

- **Checkpoint-dependency graph**: in this method we replace dependencies between intervals with dependencies between checkpoints.

Another idea is to have a **coordinated checkpointing**, since the previous algorithm is not convenient, looking at its complexity.

The idea is to have a coordinator that make a checkpoint requests to all processes. These ones have to stop and compute the local checkpoint. When done it send an ACK to the coordinator.

Another idea is to exploit the snapshot procedure, in order to not block the processes but the drawback is not so trivial.

In the end, we can merge the two approaches letting processes take checkpoints in an independent way but piggyback information on messages to allow the other processes to determine whether they should also take a checkpoint.

6.4.2 Logging

The **idea** is to log operations which occur in processes. However, the log operation must be done carefully since there may be messages logged and didn't send and viceversa. The most important thing is to log messages which contain all necessary information to replay the messages.

A message is stable if it can no longer be lost. Moreover, for every unstable message we define:

- *DEP(m)*: processes that depend on the delivery of m
- *COPY(m)*: processes that have a copy of m, but not yet a stable storage

We define also the **orphan**, that is an process that is in *DEP(m)* but all processes with *COPY(m)* have crashed. In this case, no surviving *orphans* must be left.

There is also a different kind of approach:

- *Pessimistic:* ensure that any unstable message m is delivered to at most one process. Processes cannot send any other message until it writes m to a stable storage. The goal is to not create any orphan.
- *Optimistic:* messages are logged asynchronously, with the assumption that they will be logged before any faults occur. If all processes in *COPY(m)* crashed, then all processes in *DEP(m)* are rolled back to a state where the are no longer in *DEP(m)*. This approach allows orphan creation but force them to roll back until they are no longer orphan, if all *COPY(m)* processes crashed.

7 Big data

7.1 Data science

Data science is an interdisciplinary field that uses scientific methods, processes, algorithms and systems to extract knowledge and insights from data in various forms, both structured and unstructured.
Data science is made possible by the increasing availability of large volumes of data, also called **Big Data**. Moreover, it's also became a **business model** for several companies which base their services on the data acquired and processed by their systems *(e.g. Google, Facebook, ...)*.

7.2 Big data

In order to understand what big data are, we can refer to their main characteristics, also called "the four V's", which are:

- *Volume*: we have a lot of data
- *Velocity*: the data arrives fast and, moreover, information is relevant when it's "fresh"
- *Variety*: the data arrives (and can be stored too) in many different formats
- *Veracity*: the data is not always "correct", where with the term "correct" we intend that not all the data we acquire is relevant for the purpose we have

7.3 MapReduce framework

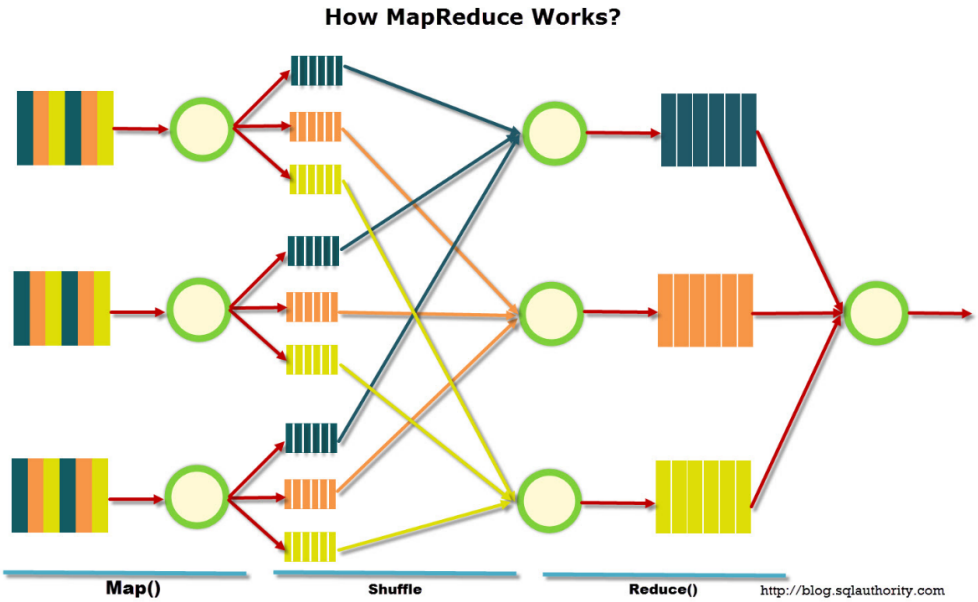
MapReduce is a software framework used to support the distributed computation on huge amount of data on computer clusters. It was introduced in 2004 by *Google* in order to approach the *Google Maps* design.
In the **MapReduce** framework the computing infrastructure is quite simple:

- Clusters of "normal" computers
- Hundreds to thousands of nodes
- No dedicated hardware, in order to be less expensive and easier to upgrade

7.3.1 Phases

- *Map:* processes individual elements and for each of them outputs one or more *¡key, value¿* pairs
- *Reduce:* processes all the values with the same key and outputs a value

Figure 26: Classic MapReduce framework



7.3.2 Platform

- *Schedule*: we need a scheduler that allocates resources for mappers and reducers. Typically we have one master and many workers. The master considers the locality of data when assigning a task to mapper in order to preserve network bandwidth and task are assigned to workers dynamically.
- *Data distribution*: in order to move data from mappers to reducers we need a infrastructure able to do that in a efficient way.
- *Fault tolerance*: we need a system that transparently handles the crash of one or more nodes
  - *Worker failure:* the master detects failure via periodic heartbeats. When a worker failure is detected, both completed and in-progress map tasks on that worker should be re-executed. On the contrary, only in-progress reduce tasks on that worker should be re-executed. This because completed reduce tasks have already wrote in the distributed file system.
  - *Master failure:* since the system state is check-pointed and saved in the *GFS (Global File System)*, a new master recovers and continues from the last known state.
    Note that, if we have *stragglers*, so tasks that take long time to execute, we need to reschedule any remaining executing task.

7.3.3 Strengths, limitations and typical applications

Strengths

- The developers write simple functions
- The system manages complexity of allocation and synchronization
- Good approach when we have to deal with large-scale data analysis

Limitation

- Very high overhead
- Lower raw performance than *HCP (High Performance Computing*
- Very fixed and rigid paradigm (the next step cannot start if the previous one is completed)

Typical applications

- Sequence of steps, each requiring map and reduce
- Series of data transformations
- *"Iterating until reach convergence"* problems

8 Bibliography

8.1 References

1. Chandy-Lamport algorithm (https://en.wikipedia.org/wiki/Chandy%E2%80%93Lamport_algorithm)