Cloud Computing and Distributed Systems

Year: 2017/2018

Afternotes

5th March 2019

Authors:
Antonio Emanuele Cinà 854866
Luca Daniel 851269

Index

1 Introduction to Distributed system **1**
1.1 Advantages . . . 1
1.2 Characteristics . . . 1
1.3 Goals . . . 2
1.4 Computer System vs Distributed System . . . 2
1.5 Open problems . . . 3
1.6 Transparencies . . . 4
1.7 WWW as Example . . . 4

2 Distributed System **5**
2.1 Communication . . . 5
2.2 HW Organization . . . 5
2.2.1 Multiprocessor . . . 7
2.2.2 Multicomputer . . . 8
2.3 SW Organization . . . 9

3 Models of Systems **12**
3.1 Architectural Model . . . 12
3.1.1 Client-Server . . . 12
3.1.2 Peer-To-Peer . . . 13
3.1.3 Variants of Client-Server Model . . . 14
3.2 Fundamental model . . . 14
3.3 Interaction models . . . 15
3.4 Fault models . . . 15
3.5 Security model . . . 17

4 Client-Server communication **17**
4.1 Addressing . . . 17
4.2 Primitives . . . 18
4.2.1 Synchronous . . . 18
4.2.2 Asynchronous . . . 18
4.2.3 Primitives with buffer . . . 18
4.2.4 Primitives without buffer . . . 19
4.3 Protocols . . . 19
4.4 Networks/Reliability . . . 20

5 Interprocess Communication **20**
5.1 Transparency and Implementation . . . 21
5.2 Object Oriented Model . . . 21
5.3 Invocation Semantics . . . 22
5.4 RMI and RPC implementations . . . 23
5.4.1 RMI . . . 23
5.4.2 RPC . . . 24
5.5 Callback . . . 25

6 Interprocess Communication - Implementation & Design **25**

7 Request-Reply communication **26**
7.1 Primitives . . . 27
7.2 Reliability and Faults in Request-Reply . . . 28
7.3 Remote procedure call . . . 28
7.3.1 Semantics of RPC . . . 29
7.3.2 RPC fault management and semantics . . . 29
7.3.3 RPC semantics and server crash . . . 30

8 Multicast communication **31**
8.1 Atomicity and Reliability . . . 31
8.2 Multicast protocols: ordering . . . 31
8.3 Multicast protocols: implementation . . . 32
8.4 Reliability and Atomicity . . . 33
8.5 Atomic and totally ordered multicast . . . 33

9 Event notification **34**
9.1 Event service . . . 35

10 Synchronization and coordination in distributed systems **37**
10.1 Mutual exclusion . . . 37
10.2 Central server algorithm . . . 37
10.3 Distributed algorithm . . . 38
10.4 Ring algorithm . . . 39
10.5 Voting algorithm . . . 40
10.6 Comparison . . . 40
10.7 Election algorithms . . . 41
10.8 The bully algorithm . . . 42
10.9 Deadlock management . . . 43

11 Coordination and synchronization: clock **45**
11.1 Model definition . . . 45
11.2 Physical clock . . . 45
11.2.1 Clock synchronization algorithm: Distributed synchronous system . . . 47
11.2.2 Clock synchronization algorithm: Cristian . . . 47
11.2.3 Clock synchronization algorithm: Berkeley . . . 48
11.2.4 Clock synchronization algorithm: NPT . . . 48
11.3 Logical clock . . . 49
11.3.1 Causal Ordering . . . 50
11.3.2 Logical clock - Lamport timestamp . . . 50
11.3.3 Logical clock - vector clock . . . 51
11.4 Global state . . . 52
11.4.1 Consistent global state . . . 53
11.4.2 Distributed snapshot . . . 53

12 Transaction and concurrency control **56**
12.1 Concurrency control . . . 56
12.2 Serial equivalence . . . 58
12.3 Conflict of operations . . . 58
12.4 Nested transactions . . . 60
12.5 Concurrency control: Lock . . . 60

12.5.1 Strict two-phase locking . . . 61
12.5.2 Deadlock . . . 62
12.6 Lock compatibility (read, write and commit locks) . . . 63
12.7 Concurrency control: Optimistic . . . 64
12.7.1 Serializability . . . 65
12.8 Concurrency control: Timestamp . . . 66

13 Distributed Transaction **68**
13.1 Coordinator of a distributed transaction . . . 69
13.2 Atomicity . . . 70
13.3 Two phase commit protocol . . . 71
13.4 Fault and performance management . . . 72
13.5 Two phase commit protocol for nested transactions . . . 72
13.6 Protocol realization . . . 73
13.7 Concurrency control for distributed transactions . . . 73
13.8 Distributed deadlock . . . 74
13.9 Transaction recovery . . . 75
13.9.1 Logging . . . 76
13.9.2 Shadow versions . . . 77
13.9.3 Recovery of the two-phase commit protocol . . . 78

14 Distributed File System **80**
14.1 File system modules . . . 80
14.2 Distributed File System requirements . . . 81
14.3 Case of study . . . 81
14.3.1 File service architecture . . . 81
14.3.2 Sun Network File System . . . 82
14.3.3 Andrew File System . . . 83

15 Replication **85**
15.1 Consistency models . . . 86
15.1.1 Strict consistency . . . 86
15.1.2 Sequential consistency . . . 86
15.2 Causal consistency . . . 87
15.2.1 FIFO consistency . . . 87
15.2.2 Weak Consistency . . . 88
15.2.3 Relaxed Consistency . . . 88
15.2.4 Entry consistency . . . 89
15.2.5 Comparison of consistency models . . . 89
15.2.6 Eventual consistency (final) . . . 90
15.3 Replica positioning . . . 90
15.3.1 System model . . . 91

16 The role of group communication **93**
16.1 Fault-tolearnt services . . . 94
16.1.1 Passive (primary-backup) replication . . . 94
16.2 Active replication . . . 95
16.3 High available services . . . 96
16.3.1 The gossip architecture . . . 96
16.4 Transactions with replicated data . . . 99

16.4.1 Architectures for replicated transactions . . . 99
16.4.2 Available copies replication . . . 100
16.4.3 Network partitions . . . 100
16.4.4 Virtual partition . . . 101
16.4.5 Creating a virtual partition. . . . 102

17 New paradigms **103**
17.1 Mobility computing . . . 103
17.1.1 Types of code mobility . . . 104
17.1.2 Data space management . . . 104
17.1.3 Mobility design . . . 104
17.2 Ubiquitous systems . . . 105

18 Operating System Support **106**
18.1 Operating system layer . . . 106
18.2 Processes and threads . . . 107
18.2.1 Address space . . . 108
18.2.2 Creation of a new process . . . 108
18.2.3 Threads . . . 109
18.3 Scheduler and Thread . . . 111
18.4 Communication: invocation and call . . . 112
18.5 Organization and distributed operating system architecture . . . 114
18.6 Scheduling: process allocation . . . 115
18.7 Scheduling algorithms for distributed systems . . . 115

19 Cloud computing **117**
19.1 Terminology . . . 117
19.2 Motivations . . . 118
19.3 Definition . . . 118
19.3.1 Cloud . . . 118
19.3.2 IT Resource . . . 119
19.3.3 Cloud Consumers and Cloud Providers . . . 119
19.3.4 Scaling . . . 119
19.4 Cloud Service . . . 121
19.5 Cloud Computing Advantages . . . 122
19.6 Risks and Limits . . . 122
19.7 Cloud Delivery Models . . . 123
19.8 Cloud Deployment Models . . . 124
19.9 Potentialities and concerns . . . 125

20 Questions & Answers **126**
20.1 Introduction . . . 126
20.2 Interprocess communication . . . 126

1 Introduction to Distributed system

A **distributed system** is one in which components located as networking computers communicate and coordinate their actions only by passing messages. It is a set of autonomous nodes that interact and collaborate in order to reach a particular goal. Computers that are connected by a network may be spatially separated by any distance, and they communicate by exchanging information through a communication network. They may be on separate continents, in the same building or in the same room. Another definition can be the following one: a **distributed system** is composed by more than one autonomous computer systems that interact to reach a given goal. Nodes are autonomous since they can work alone, maintaining processes and data. The distributed systems are different from parallel systems, in which the main focus is the execution of a single application using several cores.

1.1 Advantages

Possible advantages of having a distributed systems are:

- **Resource Sharing**, is one of the first and important advantages of a distributed system. Sharing of hardware and software resources.
- **Heterogeneity**, it can be composed by different components, O.S., hardware, applications etc.
- **Reliability**, if there is a crash the system is still alive. The system is fault-tolerant up to some crash, so it can be measured.
- **Extendibility/Scalability**, extendibility refers to the possibility to include another node in the system, while scalability refers to the possibility of including a new node of the same type.
- **Performance**, they offer results in an efficient way and optimum values of throughput and response time.
- **Transparency**, is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that system is perceived as a whole rather than as a collection of independent components. In a simpler way: *It appears to the customer as a single system.*

1.2 Characteristics

A distributed system has particular characteristics, listed up in the following section:

- **Concurrency**, nodes can work in parallel.
- **Autonomous and asynchronous**, a node can survive alone and each machine has its own clock. There isn't a unique global clock.
- **Resource sharing** - coordination - access, resources can be shared and so a distributed system implement a strategy to manage them.
- **Lack of global time**, when processes need to cooperate they coordinate their actions by exchanging messages. Since each machine is composed by an own clock, they are not able to coordinate their clock. A solution is to consider *timestamp.*
- **Faults independent**, all computer systems can fail, and it is responsibility of system designers to plan for the consequences of possible failures. Distributed systems can fail in new ways. Possible faults of nodes don't affect other nodes in the network. We can

consider in fact, nodes independent to each other in terms of faults. Processes on them may not be able to detect whether the network has failed or has become unusually slow.

All these characteristics are offered by Internet.

Figure 1: Internet as Distributed System

1.3 Goals

- **economy**, since data can be shared is not necessary to replicate them.
- **software**, their implementation follows particular standard and also application based on them.
- **flexibility**, gives a clear interface to use the system.
- **availability**, system detects fault and applies recover operations.
- **performance**, optimal performance in terms of response time, throughput, parallelism and bottleneck reduction.
- **locality** and control distribution, security and efficiency.
- **transparency**.

1.4 Computer System vs Distributed System

A computer system is characterized by single hardware, system software and application software for data or control. A distributed system, instead, is composed by distributed hardware, can have or not distributed software/application to manage data or control. Distributed hardware means that the system is composed by more than two computer systems, interconnected by a communication network and each computer system is independent from the others but can interact with them. Control is essential to manage physical or logical resources on the system, it can be:

- **centralized**, unique entity responsible to manager resources.
- **distributed**, computer systems on the net cooperate to reach the solution.
- **hierarchical**, the system is more scalable since all the functions are subdivided in different modules.

Manage data means that is necessary that the system implements some strategies to administrate resources and provide them when they are required. Data can be replicated, multiple copies in different locations, or partitioned, portion of data are stored in various locations.

1.5 Open problems

The development of a DS deals with some particular issues:

- **data sharing**, understands how and what resources must be shared and it provides system to retrieve them efficiently.
- **heterogeneity**, enables users to access services and run applications over a heterogeneous collection of computers and networks. Heterogeneity must consider different networks, computer HW, operating system (SW), programming languages and applications.
- **Concurrency**, there is the possibility that several clients will attempt to access a shared resource at the same time. Process that administrates shared resources could take one client request at a time. Processes have to communicate in order to synchronize the access to the shared resources.
- **Openness**, the system is not completely defined but it is possible to extend it, for instance including a new machine. Open distributed systems are based on the provision of a uniform communication mechanism and published interfaces for access to shared resources. Open distributed systems can be constructed from heterogeneous hardware and software, possibly from different vendors.
- **Middleware**, is an intermediate layer that provides an unique public interface. It defines a public and clear interface to use the features provided by the system.
- **Mobility**, the term mobile code is used to refer program code that can be transferred from one computer to another and run at the destination. For instance Java application can be executed in different systems, since they create a virtual environment to run the code.
- **Security**, is necessary to develop a secure system that doesn't allow unauthorized users to read/write data. Security for information resources has three components: *confidentiality* (protection against disclosure to unauthorized individuals), *integrity* (protection against alteration or corruption), and *availability* (protection against interference with the means to access the resources).
- **Scalability**, a system is defined scalable if it will remain effective when there is a significant increase in the number of resources and the number of users. The system works well independently on the amount of customers and adding components doesn't change the way in which resources are managed.
- **Quality of Service (QoS)**, provides a good QoS, which is the main nonfunctional properties of systems that affect the quality of services experienced by clients and users. *QoS* consider reliability, security and performance of the entire system.
- **Fault management**, faults need to be transparent to the user. The three main steps are identification, masking and recovery. One of the most common recovery algorithm is "checkpoint and rollback" in which we go back to the last checkpoint to recover the last consistent state. Notice that global clock does not exists, meaning that time for checkpoints can be different. Moreover, we define *"availability"* the probability of having the system working.
- **Transparency**, system is perceived as a whole rather than as a collection of independent components.

1.6 Transparencies

A system can provide different levels of transparencies:

- **Access** transparency, enables local and remote resources to be accessed using identical operations.
- **Location** transparency, enables resources to be accessed without knowledge of their location.
- **Concurrency** transparency, enables several processes to operate concurrently using shared resources without interference between them.
- **Replication** transparency, enables multiple instances of resources to be used to increase reliability and performance without knowledge of the replicas by users or application programmers.
- **Failure** transparency, enables the concealment of faults, allowing users and application programs to complete their tasks despite the failure of hardware or software components.
- **Mobility** transparency, allows the movement of resources and clients within a system without affecting the operation of users or programs.
- **Performance** transparency, allows the system to be reconfigured to improve performance as loads vary.
- **Scaling** transparency, the system and applications can expand in scale without change to the system structure or the application algorithms.

1.7 WWW as Example

The World Wide Web, also called WWW, can be seen as a large distributed system based on Internet and its characteristics are:

- **open system**, it can be extended and implemented in new ways without disturbing its existing functionality. Operations are based on communication standards and document or content standards that are freely published and widely implemented. It is open with respect to the types of resource that can be published and shared on it.
- **Client-Server architecture**, it is based on client-server architecture using standard rules for interaction (HTTP).
- **Transparency**, using DNS (Domain Name System) it reach the location transparency. With the usage of symbolic server name it is not necessary for the client to know the exact physical address IP of the server. If the server is replicated the user does not know, so we also have the replication transparency. But it doesn't provide location transparency, since the structure of the file system should be known in order to construct the resource URL.

2 Distributed System

2.1 Communication

When we talk about *"communication"*, we can separately talk about communication entities and communication paradigms. **Communication entities** can answer to the question "What is communicating?", instead, **communication paradigm** concerns "How they communicate?". Entities can be called: nodes or processes (*system-oriented entities*) or components, objects or web services (*problem-oriented entities*). These entities can communicate using different paradigms:

- **Interprocess communication**, refers to the relatively low-level support for communication between processes in distributed systems, including message-passing primitives, direct access to the API offered by Internet protocols (socket programming) and support for multicast communication.
- **Remote Invocation** covers a range of techniques based on a two-way exchange between communicating entities in a distributed system and resulting in the calling of a remote operation, procedure or method. In other words, it is based on the idea of ask/use methods implemented by a remote process. Some techniques: *Request-reply* protocols (client-server), *Remote procedure calls* (calling procedures in remote processes), *Remote method invocation* (invoke a method in a remote object).
- **Indirect communication**, allows a strong degree of decoupling between senders and receivers. In particular, senders don't need to know who they are sending to, furthermore, senders and receivers don't need to exist at the same time. Some techniques are: *group communication*, *publish-subscribe*.

2.2 HW Organization

The main goal of the hardware organization is to improve the performance maintaining the cost as low as possible. There can be developed Centralized System, that increases the CPU speed but has some physical constraints, or *Distributed Systems*, for which there can be different various models divided according to the **Flynn classification**:

Figure 2: Flynn Classification

- **SISD**, it corresponds to the Von Neuman model with single program executed by a CPU and data are stored inside a memory. CPU apply FETCH, DECODE and EXECUTE operations for each instruction of the program. In order to improve the performance of the system some techniques are adopted like: parallelism, use different ALU to execute more instruction on the same time, pipelining, simultaneously execute more operations for different instructions, or usege of hierarchies of memory (cache levels).

Figure 3: SISD

- **SIMD**, typical of parallel execution in which there's a unique program executed on a set of data. This system is implemented inside a single PC, so there is a unique program counter that decides which is the next instruction to be executed of the program. Also in this case there can be different architectures that determine how to organize the system, like: A vector of ALU working on a vector of data and produce a vector of output.

Figure 4: SIMD

- **MISD**, Multiple Instructions Single Data.

Figure 5: MISD

- **MIMD**, typical of distributed system in which different CPU executes different program on different data stream. Each CPU is associated with his Program Counter, memory, data and programs and they cooperate to execute and provide a common service.

Figure 6: MIMD

A **MIMD** system, implemented by distributed system, can be organized in different ways, Multiprocessor or Multicomputer.

2.2.1 Multiprocessor

Multiprocessor, is characterized by a set of CPUs connected in a communication network in which there is a shared memory. The advantage of this design is that it reduces transmission delay since CPUs are strictly coupled. Some architectures are proposed:

- **Bus**, CPUs are connected in a bus and they share a memory. They have to interact to decide who can access to the shared memory. Each time that a process needs a resource it asks to the shared memory. We can imagine that it is necessary to implement a synchronization system to access to the memory.

Figure 7: Multiprocessor - Bus architecture

An improvement of this architecture is based on the usage of a private memory, in which each CPU has also a private cache memory where are stored the most recently used words. If the required word is stored inside the **cache** so there is an hit and it is not necessary to access to the shared memory. The greater is the hit rate the better are the performance. Data are written in cache and inside the shared memory.

Figure 8: Multiprocessor - Bus with cache memory usage

- **Switch**, the shared memory is subdivided into n modules and particular element, called switch, are used to decide the right path. With this implementation different CPU can access to different memory module at the same time. The disadvantage of this solution is the cost of switch. N memory modules and N CPUs require N^2 switches.

Figure 9: Multiprocessor - Switch

- **Omega Network**, respect to switch architecture switches are organized as a binary tree, with the advantage of decreasing the number of required switches (n/2 log n). The drawback is given by communication delay of each switch.
- **NUMA** (*Not Uniform Memory Access*), each CPU has a local memory, this brings a increase of performance since CPU save most of data of shared memory inside local memory. It is necessary to develop an allocation algorithm for programs and data.

2.2.2 Multicomputer

Each computer system is composed by a private local memory and they can be local at different positions, loosely coupled, this brings some transmission delay and loss of transmission speed. The computer systems are autonomous but they are connected by a network and they interact using it. Also here there can be different architectures:

- **Bus**, computer systems are connected by a bus line but there is no shared memory. They can access to remote resources (like printer or file server). It is necessary to implement a system to reduce traffic and some communication network protocols. It is expansible.

Figure 10: Multicomputer - Bus

- **Switch**, computer systems can be organized as a grid of switch or an hypercube, that reduces the number of steps and switch necessary to connect all the nodes.

Figure 11: Multicomputer - Switch

- massive parallelism, supercomputer with thousands of CPU.
- Clusters of workstations, computer system connected with available components and there is particular special design to optimize performance and reliability.

2.3 SW Organization

A distributed system can be classified also based on the software architecture adopted:

- **loosely-coupled**, machines are completely independent and they share resources. Each computer system can be identified and it is able to operate independently of other in such a way that a network fault does not block the operation of single computer system.
- **tightly-coupled**, machines are independent but there is a controller that coordinates them. We can say that there exists a hierarchy structure.

The coupling can deal with both software and hardware viewpoints: a loosely coupled software provides its application running independently, and several applications can interact if installed

in several machines; a loosely coupled hardware is going to indicate the physical machines and their numbers, their independency and how they interact among each other. A distributed system is obtained by combining these features, so it can have independent applications on independent machines and so on.
We can distinguish several Operating Systems depending on the type of the system we are dealing with:

- *Uniprocessor system*: for single central system the OS provides the virtualization of the machine and manages the resources and processes, in particular using a so called ready queue, the queue of ready processes. The model for the OS can be classified in two categories: **microkernel** model which is made by layered components, in such a way that upper components provide functionalities to the lower parts; **monolithic** model in which all the components are stored.
- *Network management in a distributed system* (**NOS**)
- *Distributed system* (**DOS**)
- *Multiprocessor system* (**MOS**)

- **NOS** has loosely coupled software on a loosely coupled hardware. It's a network of machines each with its own memory, CPU, OS and devices. The remote execution requires that the job can see the resource location.

Figure 12: NOS architecture

- **DOS** has a tightly coupled software on a loosely coupled hardware. Everything is managed by global manager, without sharing memory.
  It is an operating system that produces a single system image for all the resources, in other words it represents the implementation of the abstraction of a single time-sharing system (system image) on a set of machines, possibly heterogeneous, interconnected and with the same operating system. A DOS is an operating system that runs on several machines whose purpose is to provide a useful set of services, generally to make the collection of machines behave more like a single machine. The distributed operating system plays the same role in making the collective resources of the machines more usable that a typical single-machine operating system plays in making that machine's resources more usable. Usually, the machines controlled by a distributed operating system are connected by a relatively high quality network, such as a high speed local area network. Most commonly, the participating nodes of the system are in a relatively small geographical area, something between an office and a campus. A DOS should provide:

- Common IPC (Inter-Process Communication), provides mechanisms to allow the processes to manage shared data.
- Global protection, provides protection mechanisms to maintain secure system and data.
- Process management, provides mechanisms to manage processes. For instance adopt a load balancing to schedule possible requests.
- File system, allows a unique view and access way to the file system.
- System interface, gives a unique and homogeneous interface to give the maximum transparency.

Figure 13: DOS architecture

- **MOS** has a tightly coupled software on a tightly coupled hardware with global shared memory. In this OS, there's unique global execution queue and several CPUs that executes ready processes. In order to have an advantage from the presence of the cache it's important that the scheduler considers also in which CPU have already run processes ready for the execution. This will lead to an higher **hit rate**.

Figure 14: MOS architecture

3 Models of Systems

The architecture of a system is its structure in terms of separately specified components and their interrelationships. The overall goal is to ensure that the structure will meet present and likely future demands on it.

3.1 Architectural Model

An architectural model provides the definition of what are components, structures of the system and functions associated to them. There are different types of processes: *client* (it requires resources), *server* (it provides resources), or *peer* (it is both client and server). These processes are connected by a communication network in which they can communicate and interact. Architectural models can be classified in **client-server** or **peer processes**. The choice of one of this possible architectures affects performance, reliability, scalability, cost and security of the system. A **client-server** architecture can be based on *layers* or tiers. A Layer architecture splits complex system into a number of layers, in which each level uses functions of the below layer and in turn, it will offer other services to the upper levels. Each level offers an abstraction of its implementation of the below layers to the upper layers. Tiered architectures are complementary to layering. Whereas layering deals with the vertical organization of services into layers of abstraction, tiering is a technique to organize functionality of a given layer and place this functionality into physical nodes. This technique is most commonly associated with the organization of applications and services. Each tier level is used as level of abstraction.

3.1.1 Client-Server

Client-server architecture is the most common and based on the idea that there are some processes, called **clients**, that ask for resources and special processes, called **servers**, that serve requests and replying on them. In a client-server model there can be different clients that apply a request to the server, and in order to satisfy all the requests the server maintains a queue in which stores them. When the client submits a request it waits until it receives the result. The time between send a request and receive the corresponding response is called *response time*.

Figure 15: Client-Server single-tier architecture

Considering a more complicated architecture with two tiers response time can be greater, since also the server has to wait for the required resource.

Figure 16: Client-Server multi-tier architecture

Client-Server architecture can be composed by a set of servers that provide the same services and they can interact to reduce the load of each server. If the load of a server decrease so the response time improves. Respect to the previous simple model, with only one server, these kind of architectures are also fault tolerant, since if at least one is up the system can provide response. Some of these architectures introduce a new component called **proxy**, used to reduce the traffic and provide cache functionalities. The proxy pattern is a commonly recurring pattern in distributed systems designed particularly to support location transparency in remote procedure calls or remote method invocation.

3.1.2 Peer-To-Peer

Peer-to-peer systems represent a paradigm for the construction of distributed systems and applications in which data and computational resources are contributed by many hosts on a network, all of which participate in the provision of a uniform service. Processes can be everywhere, but they have to cooperate in order to solve problems of consistency of the resources and to synchronize their actions.
In this architecture, all the processes involved in a task or activity play similar roles, interacting cooperatively as peers without any distinction between client and server processes or the computers on which they run. In practical terms, all participating processes run the same program and offer the same set of interfaces to each other. While the client-server model offers a direct and relatively simple approach to the sharing of data and other resources, it scales poorly.

Figure 17: Peer-to-peer architecture

A key problem for peer-to-peer systems is the placement of data objects across many hosts and subsequent provision for access to them in a manner that balances the workload and ensures availability without adding undue overheads.

3.1.3 Variants of Client-Server Model

There are some variants of the client-server model, for example there could be **mobility** of the node, of the code (program can be moved) or data. *Applets* are a well-known and widely used example of mobile code – the user running a browser selects a link to an applet whose code is stored on a web server; the code is downloaded to the browser and runs there. An advantage of running the downloaded code locally is that it can give good interactive response since it does not suffer from the delays or variability of bandwidth associated with network communication. When a client request something to the server, the reply given by the server is an applet code that will be executed from the client hardware. A *mobile agent* is a running program (including both code and data) that travels from one computer to another in a network carrying out a task on someone's behalf, such as collecting information, and eventually returning with the results. Mobile agents might be used to install and maintain software on the computers within an organization or to compare the prices of products from a number of vendors by visiting each vendor's site and performing a series of database operations.

3.2 Fundamental model

All of the previous models are composed of processes that communicate with one another by sending messages over a computer network. All of the models share the design requirements of achieving the performance, correctness and reliability characteristics of processes and networks and ensuring the security of resources in the system. Fundamental models decide to focus their attention to some particular aspects:

- **Interaction models**, how processes interact. In the analysis and design of distributed systems they are concerned especially in how processes communicate (message passing). The interaction model must reflect the facts that communication takes place with delays.

- **Fault models**, focus the attention on the problem of possible faults of the system. They define and classify faults, providing a basis for the analysis of their potential effects and for the design of systems that are able to tolerate faults of each type while continuing to run correctly.
- **Security model**, a distributed systems can be exposed to attack by both external and internal agents. Security model defines and classifies the forms that such attacks may take, providing a basis for the analysis of threats to a system and for the design of systems that are able to resist them.

3.3 Interaction models

The behavior and state of a distributed system can be described by a distributed algorithm – a definition of the steps to be taken by each of the processes of which the system is composed, including the transmission of messages between them. Messages are transmitted between processes to transfer informations between them and to coordinate their activities. Each process has its own state, consisting of the set of data that can access and update, including the variables in its program. The state belonging to each process is completely private – that is, it cannot be accessed or updated by any other process. There are some factors that affect performance and reliability of interaction system:

- **latency**, the delay between time of starting transmission and time starting to receive the message by receiver. It is also defined as the time to obtain a service.
- **band**, the total amount of information that can be transmitted in the network in a given time.
- **jitter**, variation in the time taken to deliver a series of messages. *Jitter* is relevant to multimedia data. For example, if consecutive samples of audio data are played with differing time intervals, the sound will be badly distorted.
- **Lack of global time** is another problem to consider. There isn't a unique clock but each process has its own clock and their usually are not synchronized. One solution can be to implement a virtual clock used by each process to verify the current state of the system.
- A distributed system can be also classified as **synchronous** or **asynchronous**. In *synchronous* system processes are synchronized and so it is more easy to develop communication protocols since there is a bound on execution, transmission and clock. Instead, *asynchronous* system are not bounded, so it is necessary to develop different strategy to provide communication protocols.

3.4 Fault models

In a distributed system both processes and communication channels may fail. The failure model defines the ways in which failure may occur in order to provide an understanding of the effects of failures.

Figure 18: Communication channels can suffer from arbitrary failures

There are different types of failure:

Class of failure	Affects	Description
Fail-stop	*Process*	Process is blocked, it can't execute and provide nothing. Other processes think that it is available and they are not able to detect that it is blocked.
Crash	*Process*	Process halts and remains halted. Other processes may not be able to detect this state.
Omission	*Channel*	A message inserted in an outgoing message buffer never arrives at the other end's incoming message buffer.
Send-omission	*Process*	A process completes a send, but the message is not put in its outgoing message buffer.
Receive-omission	*Process*	A message is put in a process's incoming message buffer, but that process does not receive it.
Arbitrary	*Process* or *channel*	Process/channel exhibits arbitrary behavior: it may send/transmit arbitrary messages at arbitrary times, commit omissions; a process may stop or take an incorrect step. The term arbitrary or Byzantine failure is used to describe the worst possible failure semantics, in which any type of error may occur. For example, a process may set wrong values in its data items, or it may return a wrong value in response to an invocation.

Table 1: Omission and arbitrary failures.

Other kinds of failure are called **timing failures** and they are associated only to *synchronous systems*, that have the constraint of time bounds.

Class of failure	Affects	Description
Clock	*Process*	Process's local clock exceed the bounds on its rate of drift from real time.
Performance	*Process*	Process exceeds the bounds on the interval between two steps.
Performance	*Channel*	A message's transmission takes longer than the stated bound.

Table 2: Timing failures.

A communication is considered reliable when there is **validity** (messages are delivered) and **integrity** (delivered messages are ordered and correct).

3.5 Security model

A distributed system can be interest of possible attacks, so it is necessary to develop system and strategies for authentication, cryptography and for maintaining integrity and security of the entire system and resources.

4 Client-Server communication

There are some issues to design a communication system for client-server architecture, such as **addressing**, **primitive**, **protocols** and **reliability**.

4.1 Addressing

Addressing refers to the ability of a client to understand where is the server (*server localization*). There can be different solution to solve these problems and answer to questions like "Where is the server?".
The first one is based on the usage of **LAN**, in which there is the assumption that client knows the address of the server and server gets also the address of the client when it receives a request.

Figure 19: Lan

The problem of this solution is that there is *no location transparency* and in case of relocating of the server it is necessary to apply *modification on the client.*

The second strategy is based on the used of **broadcast** strategy. Client sends via broadcast a special packet `locate` to identify the server and discover where it is located. Only the server answers with a message in which it is specified its physical location. This solution offers *location transparency.*

Figure 20: Broadcast

Here the problem regards the usage of channel, since now there is an increasing of traffic due to broadcast flooding.

The last solution is given by the usage of a particular entity called **Name Server**, which provides to the client the physical address of the server. The client sends to *NS* a request to get the server address and *NS* answers replying with a message containing the physical address of the server.

Figure 21: Name server

This solution provides location transparency for the server but not for the NS and for each communication the client must contact the NS, that can be considered as a bottleneck.

4.2 Primitives

Primitives can be different considering asynchronous and synchronous systems or if the receiver use or not a buffer. The classic type of primitives are `send` and `receive`.

4.2.1 Synchronous

Processes are blocked when they use primitive. In *synchronous* system with the different primitives we have the following behavior:

- `send` the sender specifies the send buffer and destination address. Process remains blocked until the buffer has been sent.
- `receive` the invocation doesn't return control to the calling process until the message has been received and stored into a destination buffer.

4.2.2 Asynchronous

Processes can go on, primitives don't block the execution of other processes. With the usage of different primitives, we have the following behavior:

- `send`, the control returns to the process as soon as the buffer to be sent is copy into the buffer of the kernel (local). Process is blocked just the time to copy message inside buffer.
- `receive` indicates to the kernel the local base address and the size of the buffer where data are received and it returns control to the calling process.

The advantage of an *asynchronous* system is that caller continues the computations concurrently to the communication, but the drawback is that it is difficult to recognize when communication is terminated.

4.2.3 Primitives with buffer

Receiver has a buffer in which it can store messages sent by sender. It can be introduced for *asynchronous* and *synchronous* systems. The procedure is described as following:

- The process that invokes the receive, asks the kernel to create the mailbox and specifies its local address (ex: port number).

- The kernel keeps the mailbox in the kernel space.
- Messages arrived with the process address, are put in the mailbox. If there's a pending receive the message is passed to the process, eventually unblocking it, otherwise the message is kept until an invocation doesn't arrive.
- If the mailbox is full the kernel could discard the message.

4.2.4 Primitives without buffer

If there's no buffer, communications must be synchronous, sender is blocked until it receives an unblock. The procedure is so defined:

- The caller is blocked when it invokes the `receive(addr, &m)`, where `addr` is the address of the sending process and `&m` is the memory space of the invoking process.
- Kernel unblocks the caller when message is received and copied in the buffer.
- If the message is received and there are no more pending `receive`, kernel cancels the message and the client tries again.
- If the `receive` is from a server that is executing operations for a client and another client sends a message, there is a `race condition` since the new client has to wait for a new `receive` from the server.

4.3 Protocols

Fragmentation and **re-composition** of messages might be needed at application level if the message is too long. Messages will be split in **packets**.

Types of Messages			
Code	**Type**	**From - To**	**Description**
REQ	Request	C → S	The client ask for a service
REP	Reply	S → C	The server sends a reply
ACK	Ack	C/S → S/C	The previous message arrived
AYA	Are you alive?	C → S	Send to the server when it doesn't answer
IAA	I'm alive!	S → C	Server reply to a AYA message
TA	Try again	S → C	Server is busy, no space. Server is overloaded
AU	Address unknown	S → C	No process uses this address

`Ack` from server to client can be useful to get some information about the communication or execution performance and it allows the client to send another request. Instead `ack` from client to server can be useful to distinguish request and discard copy of reply.
Some examples of the *client-server* interaction can be found in the following image:

Figure 22: Examples of interaction Client-Server

4.4 Networks/Reliability

Communication in distributed system depends on the usage of a network that connects computer systems. Networks can be distinguished considering their size (LAN, MAN, WAN) or transmission technology (wired, wireless), and the different level of **QoS** offered. In this case the **QoS** provided by a network takes care of its performances (transmission speed, latency, band), reliability and security. The implementation of a distributed system has to consider also this aspect, for instance latency is useful for the development of synchronous systems. Distributed Systems are mainly implemented on application level.

Figure 23: TCP/IP layers

5 Interprocess Communication

Communication between processes is essential to allow them to cooperate and reach specific goals. Interprocess communication define how processes communicate inside the network. IPC

can be implemented in different ways:

- **Remote Procedure Call**, allowing a calling process to call a procedure in a remote node as if it is local. Provide maximum transparency, processes call remote procedures as they are local.
- **Remote Method Invocation**, remote objects are called with a remote invocation. It is necessary to maintain a remote interface to give information about what kind of methods the object offers and how remote processes can invoke them. Note that the caller global environment cannot be used as well as pointers for memory allocation.
- **Events**, asynchronous notification to objects of events. Indirect communication is defined as communication between entities in a distributed system through an intermediary with no direct coupling between the sender and the receiver(s).

RMI and RPC are analogous to the local method invocation and to the local procedure call, but with a different implementation.

5.1 Transparency and Implementation

One aspect that is very important for an *IPC* is the level of transparency provided to the processes. The main goal is to call remote procedures as they are local. In this case RPC offers the maximum level of transparency, but it has some problems to distinguish local and remote procedure. The development of RPC and RMI includes:

- **Marshalling**, the process of taking a collection of data items and assembling them into a form suitable for transmission in a message. In other words, it is a technique to codify parameters in order to be sure that the structure given to the procedure is preserved. This problem rises from the possibility of a computer system to represent data in different ways. *Marshalling* codify parameters to transform them into a common representation, improving reliability and performance.
- **Invocation semantic**, determines what kind of semantic is given to the processes.
- **Binding**, strategy to identify and connect to the server. It can be ***static***, decided a priori how to reach the server, or ***dynamic***, discovers where is the server dynamically. The first one is faster since there is a direct communication to the server but the second one offers an high level of location transparency.

The **Middleware** provides transparency, the procedures call are used in same way if they are local or remote. It supports implementation for RPC and RMI and if the middleware allows different programming languages, it usually specifies the final notation using an **Interface Definition Language** (IDL). **Interfaces** are implemented in order to make the operations transparent, programs are organized in a set of modules that cooperate. The interface has only the information needed for available methods and it doesn't define the final implementation.
Two essential modules for RPC are:

- **Client stub**, used to catch the request and to translate it into a remote action.
- **Server stub**, receives the request, selects the right procedure, execute it and return possible results to the client.

5.2 Object Oriented Model

RMI is based on the **Object Orient Model** (*OOM*), in which components are called objects. An object is an entity that contains set of data and methods that define its behavior. From an implementation point of view an *OOM* offers:

- **Reference to object**, it is an alternative identifier for an object. Objects can be passed as arguments or obtained as results.
- **Interface**, it defines methods and how to use them but it doesn't specify how they are implemented.
- **Methods**, operations that define the behavior of objects of the same class. They can be invoked from external objects.

Usually Object Oriented Programming languages are supported by an entity called *Garbage Collector*, used to recovery memory space of objects that are not referenced.

Figure 24: Remote and local invocation methods

On the previous picture we can see an example of interaction using remote and local methods.

1. Object A invokes a remote invocation to object B, since it is out of the environment of A.
2. Later B invokes local methods of C and D.
3. At the end E applies a remote invocation to a method of F, since they are inside the same system but in different environments.

5.3 Invocation Semantics

Remote procedure calls provides a range of invocation semantics that specify the implementation of the invocation of the protocol **Request-Reply**. Invocation semantics are defined in base of different implementation choices considering:

- Retransmission of the request.
- Duplicate filtering in the server in case of re transmission.
- At arrival of a retransmitted request: re-execution or re-send of results.

There can be different types of **call semantic**, referring to the reliability of the RPC or RMI from the caller point of view. Local operations guarantee *Exactly Once* semantic, since the caller knows that the request will be sended and executed exactly once.

Fault tolerance measures			*Call semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

Figure 25: Invocation Semantic

- **Maybe**, the request is sent only once independently of the fact that the client receives a reply or not. There is no fault tolerance measure, there might be loss of the invocation or the result or crashing of the server. The client doesn't know if the RMI o RPC has been successful or not.
- **At least once**, there is re-transmission until the client doesn't get a reply. There might be possible duplicate executions. It can be used only for ***idempotent*** operations, which return the same result at every execution and that don't produce any change of state.
- **At most once**, there is retransmission until the client doesn't get a reply, but the server remembers the operations already executed and don't replay the same operations. The server maintains an history table in which stores client id and operations executed for him. When a request arrives to the server it verifies if the request has already been performed for the same client, and in case of positive answer, it returns the result without replay the execution. For this reason operations for this semantic are called ***non-idempotent***, since each time they are executed, they produce a change of state inside the server.
  At *most once* semantic must solve the problem of deciding how big is the history table and when remove an entry.

5.4 RMI and RPC implementations

RPC and RMI are very similar, the choice of adopting one of them essentially regards the programming language adopted, for instance oop languages (like Java) tend to adopt RMI. In this section we want to focus our attention on how this two techniques are implemented and which are the modules that interact.

5.4.1 RMI

The structure of an RMI communication system is composed by different modules that interact to reach the final goal:

- **Communication module**, it is responsible for providing a specified invocation semantic. There are 2 communication modules, one inside the client environment and the other one inside the server environment. The two components carry out the request-reply protocol between client and server, specifying the message type, the request and the remote reference of the object to be invoked.

- **Remote reference module**, it is responsible for the translation between local and remote object references and for creating remote object references. It contains a **remote object table** that records the correspondence between local object references in that process and remote object references.
- **Proxy**, it makes remote method invocation transparent to the client, that sees the invocation as if it were local. It is also responsible for marshalling and un-marshalling.
- **Dispatcher**, it is defined inside the server. When it receives a request message from the communication module, it selects the appropriate method in the skeleton required by the request.
- **Skeleton**, it implements the methods in the remote interface.

Figure 26: Internal implementation of RMI

5.4.2 RPC

The structure of an RPC communication system is composed by different components that interact to reach the final goal:

- **Stub**, its main role is to make transparent the remote call and execute marshalling. There are two kinds of stubs: **server stub** and **client stub**. *Client stub* makes transparent the remote call and execute marshalling, instead *server stub* applies un-marshalling and execute the service procedure required.
- **Dispatcher**, it opens the request message and it selects the procedure correspond to the request.

Figure 27: Internal implementation of RPC

5.5 Callback

In some cases a client can send continuous requests to the server and in this way the transmission delay and waiting time have a strong effect to the performance.
Request-Reply protocol is not ideal. A possible solution is to *reverse the interaction*: server notifies the client when it's available. The client sends a remote object containing a methods that the server can invoke. The server keeps a list of records containing clients and their requests. When an **event** occurs the server calls all the interested clients. The advantage of using this approach is that it avoids repeated calls from the clients that increase the performances, but on the other hand, the drawback is that the server has to keep a list of the interested clients and realize RMI to all the callback in the list. Regarding to the indirect communication there isn't an event space.

Figure 28: RMI Callback

6 Interprocess Communication - Implementation & Design

Communication inside distributed systems can be carried out using different implementations. There are many possible kinds of communication, we will illustrate so the three most important. We will deal with:

- **Socket**, creates, at low level, a channel between processes on different hosts. For each application it is necessary to define protocol for data coding and decoding. This solution lacks of transparency.
- **Remote Procedure Call** or **RPC** allows client programs to call procedures transparently in server programs running in separate processes and generally in different computers from the client.

- **Remote Method Invocation** or **RMI** allows an application running on a local machine to invoke the methods of another application running on a remote machine. Locally only a **remote object reference** (*Object callback*) is created, that is actually active on a different host. A program invokes methods through such a local reference. All the invocations of the methods to be transmitted are handled by the **Object Request Broker** (*ORB*).

Middleware provides transparency to the interprocess communication. It is above the transport layer adopted from the network and it gives a interface to the upper levels providing transparency on the protocol adopted (TCP or UDP).

- **UDP** represents a datagram communication so without ordering and reliability. Interface to *UDP* is based on **message passing** and to enable sending a **single message** from sender to receiver. It uses an **unreliable** communication which is based on a *not blocking send* and a *blocking receive* with possible timeouts.
  With UDP there can be different types of **faults** like: *omission* (loss of messages, ...), and *arbitrary* (unordered delivery). It can be useful for applications that don't require strong reliability (like DNS).
- **TCP** opens a stream communication. From its nature TCP doesn't provide location transparency since client has to specify IP address and port number of the server, but a solution can be the usage of a name server.
  Interface to *TCP* is based on **bidirectional stream** and to enable sending a **stream of data** between sender and receiver. It is based on **producer-consumer** communication.
  With TCP there can be different types of **faults** like: possible block of due to the *buffer* of the destination socket, control of the *correctness*, *duplication* and *ordering*, control of the *message loss*.

Figure 29: Sockets and ports

Possible faults can occur in different cases respect to UDP. Possible block due to the buffer of the destination socket, duplicate, ordering, correctness or message loss. Furthermore, it is impossible to distinguish network and destination process fault.

7 Request-Reply communication

When we are dealing with client-server architecture, the form of **request-reply** communication is supported. In this type of communication, client and server exchange messages, described as *send* and *receive* operations, which can be datagrams if using UDP, streams for TCP.

Figure 30: Request-Reply communication

7.1 Primitives

The request-reply protocol is composed by the following *primitives*:

- *doOperation*
- *getRequest*
- *sendReply*

The way in which this 3 functions are implemented correspond to the choice of a specific semantic provided by the communication system.
The method *doOperation* is used by clients to invoke remote operations, specifying the server and the operation needed to be executed. The client is usually blocked until the server performs the requested operation since this type of communication is **synchronous** in the normal case. Asynchronous request-reply communication is rare, but it can exists, and it might be useful in situations where clients are allowed to retrieve replies later. The method *getRequest* is used by the server process to receive requests. The server will reply to the client using *sendReply* method when it has finished to invoke the requested operation. The client which receives the reply from the server is unblocked.

The information to be transmitted in a request message or a reply message is shown in the figure below:

messageType	*int (0=Request, 1= Reply)*
requestId	*int*
objectReference	*RemoteObjectRef*
methodId	*int or Method*
arguments	*array of bytes*

Figure 31: Request-Reply message structure.

The first field indicates the type of the message, if it's a request or a reply message. *requestId* is the message identifier. The *doOperation* method generates a *requestId* for each message and the server remembers these identifiers. The third field is a remote reference, and the fourth field is an identifier for the operation to be invoked.

Notice that in request-reply communication each message have a **unique message identifier** by which it may be referenced. This message identifier is composed by a *requestId* which is a sequence of integers, and an *identifier* for the sender process, to be unique in the distributed system.

7.2 Reliability and Faults in Request-Reply

So far we have seen the basic characteristics of the request-reply communication protocols, but we also can consider additional properties such as *reliability* of the message delivering.

In particular, if the primitives are implemented over UDP protocol, they suffer from:

- Order of delivering not guaranteed
- Loss of messages
- Process failures

In order to cope with these problems, *doOperation* can use a **timeout**, when the client is waiting for the server's reply. After a timeout, *doOperation* sends the request message repeatedly until either it gets a reply or it is sure that the delay is due to lack of response from the server rather than to lost messages. When the server receives several request messages as it happens using the *timeout*, it should recognize multiple messages with the same request identifier and filter out the duplicates.

Another problem can be the **re-execution** of the operations: if the reply from the server has been lost, the server should execute the operation again to obtain the result unless it has stored the interested result. An *idempotent* operation could be useful in these cases. Otherwise, the **history** of transmitted reply messages can be maintained. The problem of maintaining the history table is that it can occupy the memory: in order to deal with this problem, the client can make only one request at a time, so the server can interpret each request as acknowledgement of its previous reply, so the history need to contain only the last reply message sent to each client. However periodically the messages in the history are discarded after a period of time, because the client process can terminate and does not send the acknowledgement.

7.3 Remote procedure call

In RPC, procedures on remote machines can be called as if they are procedures in the local address space. In particular *Interface definition languages* (IDLs) are designed to allow procedures implemented in different languages to invoke one another. An IDL provides a notation for defining interfaces in which each of the parameters of an operation may be described as for input or output in addition to having its type specified.

The primitives of request-reply protocols can be implemented in different ways to provide different delivery guarantees:

- *Retry request message*: asking to retransmit the request message until either a reply is received or the server reasonably considered failed.

- *Duplicate filtering*: filtering out duplicate requests at the server.
- *Retransmission of results*: keeping a history of result messages to enable lost results to be retransmitted without re-executing the operations at the server.

7.3.1 Semantics of RPC

First of all, we say that the termination is **normal** when the client receives the message from the server. The termination is considered **abnormal** if the client does not receive the reply.

- **Maybe semantics**: RPC may be executed once or not at all. If the result message has not been received after a timeout and there are no retries, it is uncertain whether the procedure has been executed. Maybe semantics is useful only for applications in which occasional failed calls are acceptable.
- **At least once semantics**: it is achieved by the retransmission of request messages as idempotent operation. The client calls with timeout, and if the answer has not arrived yet, it tries again repeating the attempts a finite number of times. The server executes the received calls and sends the answers, but it does not keep the state of previous calls.
- **At most once semantics**: it is achieved by retries masking any omission falures of the requests. The messages contain a call sequence number. The server keeps the results of the last call executed with sequence number for each request and it checks whether it is a new one. Only in this case executes it, otherwise it sends the already computed result.

7.3.2 RPC fault management and semantics

The faults can be classified into two main cathegories:

- **Communication**: working processes can not be able to communicate temporarly.
- **Node crash**: possible recovers, we should build the correct semantics.

In particular, the possible faults are the following:

- the cannot locate the server process
- loss of the client request message
- loss of the reply of the server
- server crash during the call
- client crash during the call

We can try to distinguish the several situations in which the client sends the request then the timeout reaches the deadline while it waits for the answer:

- if the client does not try again: semantics RPC is *maybe*.
- if the client tries until it gets a reply and the answer is AU, the call has *abnormal termination* with RPC semantics is *exactly once*.
- if the client tries until it gets a reply and the answer is the reply, if the server has no memory of previous execution, the server can execute the duplicated requests and so RPC semantics is *at least once*.
- if the server keeps the result of the last execution in a buffer, in case of duplicated requests it is detected and it sends again the result; RPC semantics is *at most once*.

Furthermore we should distinguish the two types of crashes: *during or after the execution* or *before* the execution. In any case, if the server crashes, with *exactly once semantics* we would have normal termination with one execution, abnormal termination with no execution. This type of semantics rises the problem of inconsistency between the client and server. Atomic transaction or implement the RPC *at most once* can be thought as a solution to this problem.

Figure 32: RPC semantics

7.3.3 RPC semantics and server crash

Crashing of server can happen during or after the execution phase, and this is totally transparent to the client. The unique information that it has is the fact that the timeout occurs. **Semantics exactly** once is very difficult to reach on a distributed system, it has to satisfy the behavior "all or nothing". Meaning that in case of normal termination the request is executed only once, otherwise in case of abnormal termination no execution of the request. In presence of server crash provide exactly once semantic it is necessary to use **atomic actions**, which allow client and server to coordinate their action in order to guarantee "all or nothing" property. But this approach is not practicable, and what is essentially done is to implement **at most once semantic**.
Summary of the two semantics:

- **At least once RPC**
  - `client`, sends request setting a timeout. If the timeout is over try again until it receives an answer or for a finite number of times.
  - `server`, executes the received call and sends the answer. It does not keep memory of the previous calls.
- **At most once RPC**
  - `client`, sends request setting a timeout. If the timeout is over try again until it receives an answer or for a finite number of times.
  - `server`, it store an history of the last call for each client. For each request checks whether it is a new one and only in this case executes it, otherwise it sends the previous computed result. Only in case of crash of the server this state is loss.

8 Multicast communication

Communication can be defined not only for one-to-one, but there other two possible cases:

- **broadcast**, consists on a one-to-all communication system. A process send the message to all the other processes. Primitives are designed to send the message to all the processes.
- **multicast**, consists on a one-to-many communication system. A process send the message to a specific group of processes. Primitives are designed to send the message to set of receivers.

Multicast communication is very frequent and there are different motivations:

- **fault tolerance - service replication**, since the request is sent to a set of servers in parallel if one of them crash the other can reply to the request. So it is tolerant to server crashing.
- **replication of data**, data can be replicated in different servers.
- **performance**, the request can be executed by a group of servers in parallel. This approach improves the performance of the system. For instance the update of several copies of different files can be performed by a specific group.

Multicast communication brings also some issues for which it is necessary to design a strategy for managing them. For instance, sender send a request to a group of servers and how it has to deal the answer? There can be different strategies: no wait, wait only for one answer, wait for some answer (how many?) or wait for all answers.

Figure 33: Multicast

8.1 Atomicity and Reliability

Multicast communication can also provide two different properties:

- **Atomicity**, there is the guarantee that all the components inside the group receive the message. It is received by all or none.
- **Reliability**, the delivery is guarantee.

An **unreliable** multicast communication system sends only once the message. It is also available the combinations: atomic and unreliable, since if one of servers does not receive the message the others discard the message.

8.2 Multicast protocols: ordering

Another important issue for multicast protocol is ordering, messages are transmitted to a group in multicast arrive to each component of the group according to the sending order. Consider the following figure:

Figure 34: Message ordering

A and B are two events sent by two different senders and A is sent before B. Ordering problem consists to develop a strategy that ensures that receivers receive the two messages with the same order. Single atomic multicast keep the FCFS order of the messages, so when there is only one sender the order is guaranteed, instead considering if there are multiple senders this strategy doesn't ensure the order.
Some strategies are considered to solve this problem:

- **FIFO ordering**, FIFO ordering is concerned with preserving the order from the perspective of a sender process. If a process sends one message before another, it will be delivered in this order at all processes in the group.
- **Causal ordering**, it considers a logical ordering, called also casual, and the messages are delivered considering that constraint. If an event A precedes an event B then the messages are delivered according to that order.

In a **totally ordering multicast** if a message is delivered before another message at one process, then the same order will be preserved at all processes.

8.3 Multicast protocols: implementation

Multicast can be supported by different types of implementations of the two primitives: `send` and `receive`. The way in which they are implemented define the characteristics of the multicast system. On the following figure we can see a basic implementation of `send` primitive, not reliable and atomic.

```
procedure multicast (dest: array of  PortId; m: messaggio);
    var i: cardinal;
    begin
        for i:=0 to Max (dest) do Send (dest(i),m);
    end;
```

Figure 35: Basic send procedure

For a reliable multicast is necessary to check possible loss of messages, crashing of sender, wait the answer and also consider possible retransmission in case of time-out. The implementation have also to consider the occurrence of faults like message omission an sender crash. A possible solution is to use a monitor with the following roles:

- check the current transmission
- manage possible retransmission
- remove crashed processes, if a process is waiting for an ack but the server crashed it will wait for ever.
- notifies the group's components when a process is added, or when a process is excluded.

8.4 Reliability and Atomicity

Atomicity and reliability bring with them a non indifferent cost since:

- *sender* sends the message to the group and waits for the ack from all the components inside the group.
- if all the ack arrive it'ok, otherwise if they do not arrive within a timeout it is necessary to re-transmit the message.

Another problem for atomicity regards the management of sender fault. In this case to preserve atomicity every destination process sends an ack and wait for confirm, the sender once the multicast is completed sends back to confirm to everyone. But this solution is not so efficient, and so we can consider **Hold-back queue** as alternative.
With **Hold-back queue** the destination keep the message suspended before delivery, up to the arrival of the confirm. In order to guarantee the ordering and atomicity the messages are numbered and every message is delayed until the previous ones arrive.

Figure 36: Hold-Back queue

Another solution consists to the usage of **negative ack**, the messages are numbered according to the order from the server and the destination process sends a **NACK** if the message number of the arrival is not in the order of the sequence. The destination process does not send the **ACK** and the sender has a copy (history) of the sent messages. But using only ack is difficult to understand when it's time to clear the history, so occasionally destination processes send positive ack with number of received message using piggybacking technique.

8.5 Atomic and totally ordered multicast

In order to guarantee atomicity and totally order each process is associated with a unique *totally ordered identifier*. A message is **stable** in the destination if other messages with smaller identifier cannot arrive. Destination pass to upper levels (application) only stable messages. This solution is easy to implement if sender and receivers agree with a **shared identifier sequence** used to assign identifier to the messages. In order to decide this sequence some solutions are proposed:

- using a timestamp from physical or logical clock
- using a sequencer process
- using a protocol among the group processes to generate an identifier

Regarding the implementation it can defined:

- **centralized**, unique manager that makes the ordering. But the drawback is that it can became a bottleneck since all the request should pass from him.
- **distributed**, coordination of the receivers that makes an agreement on the ordering. In general what is adopted in this case is the usage of distributed algorithms.

Figure 37: IP multicast communication

9 Event notification

In general we refer to objects that want to receive information and be notified by an event for which it is subscribed. This approach introduces a new paradigm called **publish-subscribe**.

Figure 38: Eventi notification

A publish-subscribe system is a system where publishers publish structured events to an event service and subscribers express interest in particular events through subscriptions which can be arbitrary patterns over the structured events. For example, a subscriber could express an interest in all events related to this textbook, such as the availability of a new edition or updates to the related web site. The task of the publish-subscribe system is to match subscriptions against published events and ensure the correct delivery of event notifications. A given event will be delivered to potentially many subscribers, and hence publish-subscribe is fundamentally a one-to-many communications paradigm. In other words: an object generates events and publishes the type of event that can be notified, and another one that wants to receive **notifications** executes a subscribe to the event types of interest. This approach preserve heterogeneity since publisher can assume different types and events can be with different types. It is commonly adopted by *asynchronous systems*, and it is different from callback since the last one is a one-to-one communication sistem, instead in event notification more than one objects can be interested on a particular event and all of them receive a notification when it occurs.

Figure 39: Eventi notification

Two fundamental characteristics of publish-subscribe systems are:

- **Heterogeneity**, event-generating objects publish the types of events they offer, and that other objects subscribe and provide an interface for receiving and dealing with the resultant notifications.
- **Asynchronicity**, notifications are sent asynchronously by event-generating publishers to all the subscribers that have expressed an interest in them to prevent publishers needing to synchronize with subscribers – publishers and subscribers need to be decoupled.

9.1 Event service

Event service is a service that manages the event space, for which the main goal is to keep publisher and subscriber independent. Publisher uses event service to produce events and subscriber uses it to specify the event of interest in order to receive the notification. Event service maintain a database of published events and of interest.

Figure 40: Event service

Notification can be modeled considering different solutions:

- **Direct notification**, notification is performed directly by the object of interest to the subscriber, no overhead is introduced. The drawback is that when there are a lot of subscribers there is an huge overhead inside event space.
- **Notification via observer**, use another entity called **observer** for managing the notification. There can be adopted more than one observers and for each observer is associated a set of subscriber as a hierarchical system. This solution improves scalability, fault management and in terms of performance each observer has a local queue of notifications, instead of using a unique one like direct communication. This solution is also the best one in terms of security since there is no direct communication with the object and the object of interest is inside the local environment.
- **Notification via observer with external object**, observer time to time ask if the event is done. Scalability, many object of interest for the same observer.

Figure 41: Notification models

The observer can implement different functionalities:

- **forward observer**, deliver notification from different objects.
- **filtering**, reduce the number of notification received from an observer.
- **event model**, interest to specific relations among events.
- **mailbox**, introduces a delay in the notification delivery when the subscriber is not ready. This technique is useful to improve fault management.
- **Priority**, observer can manage priority for notification.

10 Synchronization and coordination in distributed systems

This section introduces a collection of algorithms whose goals vary but that shares an aim that is fundamental in distributed systems: for a set of processes to synchronize and to coordinate their actions or to agree on one or more values.
When we talk about synchronization is important to distinguish:

- **Internal:** agreement of a set of processes. In this case, for example we can consider an *election* algorithm.
- **External:** forced by an external agent.

In distributed systems there are several problems to analyze as:

- Clock synchronization.
- Mutual exclusion.
- Election algorithm.
- Atomic transactions.
- Deadlock management.

10.1 Mutual exclusion

Mutual exclusion in distributed systems can be considered as the coordination activity among a set of processes to share resources and keep consistency in the distributed system.
Distributed mutual exclusion of a set of resources is characterized by three main features:

- **Safety:** at most one process at time executes the critical section.
- **Liveness:** A process that asks to enter in a critical section eventually enters it and at the end it leaves it (no *deadlock* and no *starvation*).
- **Ordering:** entering a critical section occurs accordingly to the causal ordering.

10.2 Central server algorithm

It has a simple implementation, in fact, it simulates the primitives offered by the centralized system through a coordinator process. Its standard behavior is structured as follow:

- A process asks the coordinator to enter and it is possibly blocked.
- The answer is an access **token**.
- At receiving time of the answer, it enters the critical section.
- At the exit time from the critical section it communicates to the coordinator (it returns the *token*).

Figure 42: Server Managing a mutual exclusion token for a set of processes: centralized algorithm.

10.3 Distributed algorithm

Distributed algorithm works under the assumption that there exists a global ordering of events (*reliable transmission*). According to the different phases of the algorithm are present different protocols as:

- Protocol to enter/exit critical section:
  - A process that wants to enter a critical section sends a message to all the other processes, via *multicast* with the name of the critical section, its own identifier and the local *timestamp*.
  - It waits for the answer from all the processes.
  - Once all the `OK` have been received, it enters the critical section.
  - At the exit time from the critical section, it sends `OK` to all the processes in the local queue.
- Protocol to receive a request:
  - Be out of the required critical section and it doesn't want to enter → sends `OK` to the sender.
  - Be in the critical section → it doesn't answer and put the message in a local queue.
  - Wants to enter the critical section → compares the *timestamp* and the oldest has higher priority, if it is the other process, it sends the `OK`, otherwise, if it is itself, it put the message in the local queue.

The exposed algorithm take the name of **Ricart and Agrawala's** algorithm.

Figure 43: Multicast synchronization

A distributed algorithm satisfies the criteria of mutual exclusion (*safety*, *liveness* and *ordering*). Moreover, it is totally distributed, without a central element of control. Some disadvantages instead are the high traffic generated by N processes that require $2(N-1)$ messages for the multicast request and the answers, the fault of the system if one of the processes crashed (other processes will wait its answer) and the possible bottleneck introduced by processes.

10.4 Ring algorithm

Processes share a *token* using a **local ring** structure, there are so a circular ordering of the processes. The first process has a *token* that uses and then it forwards to the next one. The process that has the *token* is enabled to access the critical section.
This particular algorithm verifies condition 1 and 2 but not condition 3. It has the following features:

- **Costs:** $[1, N-1]$ messages to obtain the token, 1 message to exit the critical section and $[1, N-1]$ messages to synchronize on the critical section.
- **Reliability:** in some situations it is necessary to rebuild the logical ring (process faults) or to make an election of the next process that will have the token (process have a token faults).
- **Performance:** the algorithm always requires bandwidth to transmit the token even if none ask for the critical section.

Figure 44: A ring of processes transferring a mutual exclusion token.

10.5 Voting algorithm

In order to enter a critical section, it is necessary to synchronize only a subset of the interested processes, election algorithms are in fact used in the subset. The processes make a vote to decide which process can enter the critical section:

- Vote set V_i is subset of $\{p_1, \ldots, p_N\}$, associated to each process p_i.
- A process that wants to enter a critical section sends a message to all the other processes in V_i.
- It waits for all the *reply*.
- Once all the `OK` have been received, it enters the critical section.
- At the exit time from the critical section, it sends *release* to all the other members of V_i.
- A process p_j in V_i that receives the request $\rightarrow$ if its state is `HELD` or it already answered after having received the last message *release* doesn't answer and put the request in a local queue, otherwise it immediately answer with a *reply*.
- A process that receives a *release* takes a requests from the queue and sends a *reply*.

This algorithm takes the name as **Maekawa's** algorithm.

10.6 Comparison

Algorithms	**Cost of critical section (enter/exit)**	**Delay (enter/exit)**	**Problems**
Centralised	3	2	Coordinator fault
Distributed	$2(n-1)$	$2(n-1)$	Process crash
Ring	$[1, n]$	$[0, n-1]$	Token loss Process crash
Vote	$2\sqrt{n}$ $\sqrt{n}$	$2\sqrt{n}$ $\sqrt{n}$	Process crash of the voting group

10.7 Election algorithms

The goal of this kind of algorithms is the fact that at the end of the algorithm all the processes agree on an **elected process** that act as **coordinator** in the distributed system.
The following assumptions are needed:

- All the processes are uniquely numbered.
- The coordinator has usually the highest number among the "live" processes.
- Each process knows the number of all the other processes.

Generally, a process calls an election and at the end of the algorithm, all the processes agrees on the same **elected process**.

Circular ordering. One possible election algorithm can be the *circular ordering*. It is structured as a logical ring composed by an ordered list of live processes in which the election message is sent only to the next process. The algorithm needs two further assumptions: the communication is reliable and processes can have faults. It works as follows:

- Every process at the beginning is **non participant**, it will become **participant** adding its name to the list.
- If the election message id is **greater**, it passes and becomes participant.
- If the election message id is **smaller**, it put its own id and passes (if it is not already participant).
- If the election message id is **the same**, then is a **coordinator** and becomes a *non participant*, sends an **elected** message.
- Every other process that receives an **elected** message is marked as *non participant*.

Figure 45: A ring-based election in progress.

In the worst case, election needs $3n - 1$ messages with an equal response time.
Fault tolerance is low but can be improved by introducing techniques of fault identification and ring reconstruction. Moreover, this algorithm verifies only the second condition of mutual exclusion.

10.8 The bully algorithm

This algorithm has the same assumption of the *circular ordering* one. Moreover, each process can communicate with processes having higher ID. The messages exchanged are: election, answer and coordinator. It works as follows:

- A process starts the election by sending an **election** message to all the processes with **higher** id, and waits for the reply.
- If the reply does not arrive within a timeout a coordinator is nominated (self-election!) and communicate the news to the other processes by sending a coordinator message to all the processes with lower id.
- Otherwise it waits for the arrival of a coordinator message and if it does not arrive, starts another election.
- A process that *receives* a **coordinator message** saves the # and consider that process the coordinator.
- A process that *receives* an election message sends a reply to the sender and starts a new election, unless it has already done it, even if there is another coordinator.

Figure 46: Example of execution of Bully algorithm.

A process starts an election in the following cases:

- When a process is reactivated after a fault, and if it has the higher id it becomes the new coordinator and let the other process know it even if there is another coordinator).
- When a process realizes that the coordinator does not answer.
- When a process receives an election message from a process with lower id.

The *bully* algorithm has the following costs:

- Worst case: $O(n^2)$ messages (Election starts from the lower id).
- Best case: $O(n-1)$ messages (Election starts from the process with higher id).

It verifies condition 2 (*liveness*) but not condition 1 (*safety*) in case in which the process is substituted by another one with the same id. There's also a problem of *timeout* accuracy due to the use of synchronous communications.

10.9 Deadlock management

Deadlock could be a side effect of the **mutual exclusion**. It could happen because of:

- **Resource allocation without pre-emption:** the system cannot force a process to release a resource.
- **Hold and wait:** a process that holds a resource keeps holding it (it blocks the resource) and can wait for another resource.
- **Circular waiting:** there exists a closed path in the allocation graph.

Figure 47: Deadlock conditions

There are different types of deadlock:

- **Communication deadlock:** if the resource is a buffer.
- **Direct store and forward deadlock:** if it involves two nodes.
- **Indirect store and forward deadlock:** if it involves more than two nodes.

In order to manage the possible presence of deadlocks a **PAID** technique is used (Prevent - Avoid - Ignore - Detect):

- **Prevention:** applied by:
    - allowing only one resource allocation.

  - pre-allocate resources before execution.
  - resource allocation ordering.
  - Age rule.

- **Avoidance:** It determines the stable states on the basis of the process needs (only if it's safe, it acquires the resources).
- **Ignoring:** applied to terminate processes in cyclic waiting.
- **Detection:** applying *Chandy-Misra-Haas* algorithm:

  1. A blocked process P_1 starts a test (structure composed of $ID_{\text{blocked process}}$, $ID_{\text{sending test process}}$, $ID_{\text{receiving test process}}$).
  2. P_2 receives the test and, if it doesn't need other resource, it stops the test. Otherwise, if it is blocked by P_3, it forwards the test.
  3. If a process receives a test with $ID_{\text{blocked process}} = ID_{\text{receiving test process}}$, it detects the deadlock.

About *deadlock removing*, it is possible to terminate processes that create it or performing a roll-back to consistent state.

11 Coordination and synchronization: clock

Time is an important and interesting issue in distributed systems, for several reasons. First of all, time is a quantity we often want to measure accurately. In order to know at what time of day a particular event occurred at a particular computer it is necessary to synchronize its clock with an authoritative, external source of time. Second, algorithms that depend upon **clock synchronization** have been developed for several problems in distribution. These include maintaining the consistency of distributed data, authentication protocols, eliminating the processing of duplicate updates, ordering events and serialization of transactions. In distributed systems there's no **global physical clock**, meaning that there can be skew between computer clocks on the same network. In order to provide clock synchronization different algorithms are designed.
These algorithms can be developed considering **physical clocks** and **logical clocks**.

11.1 Model definition

Considering the following example that, for assumption, communicates using message passing.

Figure 48: Clocks

There are N processes $p_1, p_2, ..., p_N$ connected in a network. Each process has a **local clock** and there is no shared memory, they can communicate only using message passing.
We define ordering with the following notation:

$$e \rightarrow_i e' \qquad \text{if the event e occurs before } e' \text{ in process } p_i$$

$$h =< e_i^0, e_i^1, ... > \qquad \text{state history of process i}$$

Where the **history** represents the sequence of events that take place on the process.

11.2 Physical clock

Synchronization algorithms based on **physical clocks** operate using local clock of each process. With the notation $C_i(t)$ we define the local clock of process p_i at physical time t. It is commonly used to consider the timestamp, time indicating when the event occurs. Two events are successive only if the clock resolution (the time between two successive updates of the clock value) is less than the time interval between the two successive events. For example: if the local clock consider only minutes all the events occurring between $0 < t < 60$ have the same arrival time. Different local clock can have different values, and it is possible to define:

- **Skew**, difference between two clocks.
- **Drift rate**, frequency of the local clock is different from an ideal one.

A possible idea consists to synchronize local clocks with the UTC (Coordinated Universal Time), which provides the real time of the earth. Synchronization can be:

- **external**, if it is forced by external agents. Formally speaking:

$$|S(t) - C_i(t)| < D \qquad 1 \leq i \leq N \qquad \forall \in I$$

Where $S(t)$ is the real time (provided for instance by UTC), D is the precision and I is the interval of synchronization. This procedure synchronizes the clock of each process i with precision D from the real time.

- **internal**, if there is an agreement among a set of processes. Formally speaking:

$$|C_i(t) - C_j(t)| < D \qquad 1 \leq i \leq N \qquad \forall \in I$$

All the clocks are synchronized with precision D.

Figure 49: External and Internal synchronization

A physical clock H is correct if the drift is within a given threshold $\rho > 0$. Formal definition: Given t, t' if $t' > t$ then $(1-\rho)(t'-t) \leq H(t') - H(t) \leq (1+\rho)(t'-t)$. A clock C is defined as **monotone** if it only goes on forward. Some events can break monotonicity of a clock, just to give an idea, recovery procedure set the current time to a previous time.

From the physical clock we can define also software clock which is given by:

$$C_i(t) = aH_i(t) + b \qquad \text{a,b constants}$$

.

Figure 50: Clock deviance from real time

11.2.1 Clock synchronization algorithm: Distributed synchronous system

We begin by considering the simplest possible case: synchronization between processes in a synchronous system. In a **synchronous system**, bounds are known for the *drift rate* of clocks, the *maximum message transmission delay*, and the time required to execute each step of a process. One process sends the time t on its local clock to the other in a message m. In principle, the receiving process could set its clock to the time $t+T_{com}$, where T_{com} is the time taken to transmit m between them. The two clocks would then agree since the aim is internal synchronization, it does not consider external synchronization with UTC.

Figure 51: Clock synchronization: Sync. System

On this image we have two processes p and p' that want to communicate using message passing. p sends a message with the local clock t and p' receives the message and sets its own local clock to $t + T_{com}$. From a distributed synchronous system we know that:

- There is always a **minimum transmission time**, T_{min}, that would be obtained if no other processes executed and no other network traffic existed. It can be measured or conservatively estimated.
- In a synchronous system, by definition, there is also an upper bound max T_{max} on the time taken to transmit any message.

From these results some possible choice can be discussed to determine the threshold T_{com}:

- set T_{com} threshold equal to T_{max} or T_{min}, with possible **shew error** equal to:

$$\text{error} \leq \Delta = (T_{max} - T_{min})$$

- set T_{com} threshold equal to $(T_{max} + T_{min})/2$, with possible **shew error** equal to:

$$\text{error} \leq \frac{\Delta}{2}$$

In general from a better analysis it is possible to say that for a synchronous system, the optimum bound that can be achieved on clock skew when synchronizing N clocks is $\Delta(1 - 1/N)$.

11.2.2 Clock synchronization algorithm: Cristian

Most distributed systems found in practice are **asynchronous**: meaning that message delays are not bounded and there is no maximum bound on message transmission delays. For an asynchronous system, we can only say only that $T_{com} = min + x$, where $x > 0$ and it is not known in a particular case. Some algorithms are designed in order to implement clock synchronization for asynchronous systems. The first one was given by **Cristian**, in 1989, which suggested the usage of a **time server**, connected to a device that receives signals from a source of UTC, to synchronize computers externally.

Figure 52: Cristian Algorithm

Every machine periodically asks for the time to the time server and when it receives the reply:

- It checks the clock.
- It computes the network delay (at least T_{min}) as T_{round}, round trip time.
- It sets the clock to $t + T_{round}/2$.

From the point of view of the time server when it receives a time request the server puts the time t in the reply message just at the last moment useful for sending. The precision D given by this technique is equal to $\pm(T_{round}/2 - T_{min})$.
This algorithms works well when synchronization delay is close to zero. But in any case it has some drawbacks like: the usage of a central server reduces and limits reliability and performance, or malicious interference of the clients. Some aspects can be improved for example using multicast instead of using a single and central server or introduce authentication technique for the clients.

11.2.3 Clock synchronization algorithm: Berkeley

Another interesting algorithm for clock synchronization in asynchronous systems is called **Berkeley**. Berkeley uses a coordinator computer as the master, also called **active time server**. Unlike in Cristian's protocol, this server periodically polls the other computers whose clocks are to be synchronized, called **slaves**. The slaves send back their clock values to it. The master estimates their local clock times by observing the round-trip times (similarly to Cristian's technique), and it averages the values obtained (including its own clock's reading). The coordinator indicates who has to speed-up or speed-down the clock using the average value. The idea is that this average cancels out the individual clocks' tendencies to run fast or slow. The master eliminates any occasional readings associated with larger times than this maximum.
Instead of sending the updated current time back to the other computers – which would introduce further uncertainty due to the message transmission time – the master sends the amount by which each individual slave's clock requires adjustment. This can be a positive or negative value.
The Berkeley algorithm eliminates readings from **faulty clocks**. Such clocks could have a significant adverse effect if an ordinary average was taken so instead the master takes a **fault-tolerant average**. That is, a subset is chosen of clocks that do not differ from one another by more than a specified amount, and the average is taken of readings from only these clocks. In other words, the average does not consider the times too far or with abnormal values, meaning that is more **fault-tolerant** than a simple average.
But as Cristian's algorithm Berkeley uses a central server that limits reliability and performance of the entire system.

11.2.4 Clock synchronization algorithm: NPT

Cristian and Berkeley algorithm are based on the usage of a centralized system, which limits some properties of the entire system. Another approach is given by **distributed algorithms**.

The **Network Time Protocol** (NTP) defines an architecture for a *time service* and a protocol to distribute time information over the Internet. NTP provides:

- synchronization of the physical clocks respect to UTC for all the clients inside the network.
- **reliable** service that is fault tolerant to connection loss. Fault tolerance is given by redundancy of server and path.The servers can reconfigure so as to continue to provide the service if one of them becomes unreachable.
- **scalability**, allows frequent synchronization also in presence of many nodes.
- To provide protection against interference with the time service, whether malicious or accidental: The time service uses authentication techniques to check that timing data originate from the claimed trusted sources. It also validates the return addresses of messages sent to it.

The NTP service is provided by a network of servers located across the network. **Primary servers** are connected directly to a time source, like UTC. Instead **secondary servers** are synchronized, ultimately, with primary servers. The servers are connected in a logical hierarchy called a **synchronization subnet**, whose levels are called **strata**.

Figure 53: NTP subnet

The greater is the number of servers adopted the greater is the reliability provided, but this decrease the precision since the synchronization takes time and bring delays.
Server synchronization can be done:

- **multicast**, common with high speed LAN with small delay.One or more servers periodically multicasts the time to the servers running in other computers connected by the LAN, which set their clocks assuming a small delay. Low precision.
- **procedure call**, the server is passive and waits for requests like Cristian's algorithm. More precision.
- **symmetrical**, servers exchange messages with timestamp, it is commonly used for low levels. More precision.

11.3 Logical clock

In distribute systems since we cannot synchronize clocks perfectly across a distributed system, we cannot in general use physical time to find out the order of any arbitrary pair of events occurring within it. Another interesting solution is given by **logical clock**, but first it is necessary to give some notions of **causal ordering**.

11.3.1 Causal Ordering

On this section we will define some formal notation that will be adopted later. Consider N processes $p_1, p_2, ..., p_N$: We write $e \rightarrow_i e'$ if event e occurs before e' in process p_i. When p_i sends a message m to p_j the event `send(m)` precedes event `receive(m)`:

$$send(m) \rightarrow receive(m)$$

Causal ordering defines order between events and it has some properties:

- if $\exists p_i \rightarrow e' \quad \Longrightarrow \quad e \rightarrow e'$
- $\forall$ message $m \quad send(m) \rightarrow receive(m)$
- if e, e', e'' are events:
$$e \rightarrow e', e' \rightarrow e'' \quad \Longrightarrow \quad e \rightarrow e''$$
- event a and b can be concurrent and we denote this property as $a||b$. Meaning that they are not related.

Figure 54: Causal ordering example

11.3.2 Logical clock - Lamport timestamp

Logical clock was introduced by Lamport in 1978, and it represents the causal ordering (*happened-before*) of events. It is independent from physical clock and it is basically a software counter. Each process p_i keeps its own **logical clock**, L_i , which it uses to apply so-called **Lamport timestamps** to events. We denote the timestamp of event e at p_i by $L_i(e)$. Processes update their logical clocks and transmit the values of their logical clocks in messages as follows:

- **LC1**: L_i is incremented before each event is managed at process p_i. In other words, $L_i = L_i + 1$ $\forall$ event that occurs in p_i the logical clock is **incremented**.
- **LC2**: if p_i sends a message m it sends in **piggybacking** the value $t = L_i$. if p_j receives a message (m, t) it sets $L_j = max(L_j, t)$ and applies **LC1** for the event `receive(m)`.

Figure 55: Logical clock algorithm example

Logical clock has the monotonicity property, meaning that:

$$\text{If} \quad e \to e' \quad \Longrightarrow \quad L(e) < L(e')$$

Note that this relation is not an if-and-only-if, so if we know that $L(e) < L(e')$ we cannot infer that $e \to e'$.

11.3.3 Logical clock - vector clock

Some pairs of distinct events, generated by different processes, have numerically identical Lamport timestamps. However, we can create a global ordering of events by taking into account the pair (timestamp,p_i). If e is an event occurring at p_i with local timestamp T_i , and e' is an event occurring at p_j with local timestamp T_j , we define the global logical timestamps for these events to be (T_i, i) and (T_j, j), respectively. And we define:

$$(T_i, i) < (T_j, j) \quad \Longleftrightarrow \quad T_i < T_j \quad \vee \quad (T_i = T_j \wedge i < j)$$

This ordering has no general physical significance (because process identifiers are arbitrary), but it is sometimes useful. A different ordering that overcomes the iff limitation of the Lamport definition, $L(e) < L(e') \not\Rightarrow e < e'$, is the **vector clock**. To each process p_i we associate a **vector of clocks** V_i used for local timestamp.

VC1: Initially, $V_i[j] = 0$, for $i, j = 1, 2 \ldots, N$.

VC2: Just before p_i timestamps an event, it sets $V_i[i] := V_i[i] + 1$.

VC3: p_i includes the value $t = V_i$ in every message it sends.

VC4: When p_i receives a timestamp t in a message, it sets $V_i[j] := max(V_i[j], t[j])$, for $j = 1, 2 \ldots, N$. Taking the component-wise maximum of two vector timestamps in this way is known as a *merge* operation.

Figure 56: Vector clock algorithm

- $V_i[i]$ represents the event number occurred in p_i and marked by p_i.
- $V_i[j]$ represents the event number occurred in p_j and that potentially affected p_i.

Then:

$$V(e) < V(e') \qquad e < e'$$

$$V = V' \quad \Longleftrightarrow \quad V[i] = V'[i] \qquad \forall i = 1, ..., N$$

$$V \leq V' \iff V[i] \leq V'[i] \quad \forall i = 1, ..., N$$
$$V < V' \iff V[i] \leq V'[i] \quad \wedge \quad V \neq V' \quad \forall i = 1, ..., N$$

Figure 57: Vector clock algorithm example

The main drawback of this strategy is that it requires memory space to store the vector V and message dimension increase proportional to N.

11.4 Global state

Another fundamental problem consists to verify global properties in a distributed system. We begin by giving the examples of a distribute system in which deadlock between two or more processes can happen.

Figure 58: Global state: deadlock

As it is possible to see is fundamental to study properties of the system for know better what it can happen and possible issues. Knowing the **global state** of the system sometimes can solve some problems.
The essential problem is the absence of global time. If all processes had perfectly synchronized clocks, then we could agree on a time at which each process would record its state – the result would be an actual global state of the system. From the collection of process states we could tell, for example, whether the processes were deadlocked. But we cannot achieve perfect clock synchronization, so this method is not available to us.

Each process p_i is associated to a local state history: $h_i =< e_i^0, e_i^1, ... >$.

- e_i^k k-th event in the local history of p_i. Local state history of process p_i up to k is so defined: $h_i^k =< e_i^0, e_i^1, ..., e_i^k >$.
- s_i^k state of p_i just after occurrence of event e_i^k.
- s_i^0 initial state of p_i.

Global state history of the set of processes $\{p_1, p_2, ..., p_N\}$.

$$H = \bigcup_{i=1}^{N} h_i$$

$$S = \{s_1, s_2, ..., s_N\} \qquad \textbf{global} \quad \textbf{state}$$

But now the question is spontaneous: Which are the significant states? **Distributed snapshot** is an algorithm used to derivate a global state in which the distributed system can be. It defines a **consistent global state**.
A cut of the system's execution is a subset of its global history that is a union of prefixes of process histories:

$$C = \bigcup_{i=1}^{N} h_i^{c_i} \qquad C \to \{e_1^{c_1}, ..., e_N^{c_N}\}$$

11.4.1 Consistent global state

A cut of a distributed system is said to be **consistent** if for each event included in the cut, it also includes the related events according to *happened-before*:

$$\forall e \in C \quad e' \to e \qquad \Longrightarrow \qquad e' \in C$$

Figure 59: Consistent and Inconsistent cut

A global state is said to be consistent if it corresponds to a consistent cut. A **global execution** is a succession of global consistent states $S_0 \to S_1 \to S_2 \to ....$ A **consistent run** is an ordering of the events in a global history that is consistent with this happened-before relation o $\to$ on H.

11.4.2 Distributed snapshot

Chandy and Lamport describe a **snapshot algorithm** for determining global states of distributed systems, which we now present. The goal of the algorithm is to record a set of process and channel states (a snapshot) for a set of processes p_i $(i = 1, 2, ..., N)$ such that, even though the combination of recorded states may never have occurred at the same time, the recorded global state is consistent. The algorithm records state locally at processes; it does not give a method for gathering the global state at one site. An obvious method for gathering the state is for all processes to send the state they recorded to a designated collector process, but we shall not address this issue further here. The algorithm assumes that:

- reliable channel and communication. Neither channels nor processes fail – communication is reliable so that every message sent is eventually received intact, exactly once.
- Channel is unidirectional and provide FCFS-ordered message delivery.
- The graph of processes and channels is strongly connected (there is a path between any two processes).

- Any process may initiate a global snapshot at any time.
- The processes may continue their execution and send and receive normal messages while the snapshot takes place.

A process that starts records its own state and then sends a special message (**marker**) on all the outgoing channels to other processes to gather the global state. A process that receivers a marker message if it has not recorded its state it records it and forwards the marker on the outgoing channel, otherwise if it has already recorded its state it records the channel state (message sequence received on the channel from its last registration and before receiving the marker).

Marker receiving rule for process p_i
On receipt of a *marker* message at p_i over channel c:
if (p_i has not yet recorded its state) it
records its process state now;
records the state of c as the empty set;
turns on recording of messages arriving over other incoming channels;
else
p_i records the state of c as the set of messages it has received over c since it saved its state.
end if

Marker sending rule for process p_i
After p_i has recorded its state, for each outgoing channel c:
p_i sends one marker message over c
(before it sends any other message over c).

Figure 60: Chandy-Lamport algorithm

Figure 61: Distributed snapshot 1

Figure 62: Distributed snapshot 1

12 Transaction and concurrency control

A transaction defines a sequence of server operations that is guaranteed by the server to be atomic in the presence of multiple clients and server crashes. Formally, a **transaction** is an operation sequence on the server, involving a set of processes and/or shared resources, that are guaranteed to be **atomic** (by the server) in presence of **concurrency** and **faults**. The goal of transactions is to ensure that all of the objects managed by a server remain in a consistent state when they are accessed by multiple transactions and in the presence of server crashes.
Properties of the transaction:

- **Isolation**, they are free from interference by operations being performed on behalf of other concurrent clients. In other words, all the operations are executed successfully or if just one is not completed, one has to reconstruct the initial state.
- **All or nothing**, either all of the operations must be completed successfully or they must have no effect at all in the presence of server crashes. In other words, there is no interference with the other transactions, partial effects are not visible.

An easy way to remember transaction properties is to use the word **ACID**:

- **Atomicity**, a transaction must be all or nothing.
- **Consistency**, a transaction takes the system from one consistent state to another consistent state. Multi-threading and possible **interleaving** of methods can generate different result in absence of any type of control.
- **Isolation**, each transaction must be performed without interference from other transactions
- **Durability**, after a transaction has completed successfully, all its effects are saved in permanent storage. Data saved in a file will survive if the server process crashes.

A client's banking transaction

Transaction T:
a.withdraw(100);
b.deposit(100);
c.withdraw(200);
b.deposit(200);

Figure 63: Example of a transaction

In general **transactions** can be part of the middleware.

12.1 Concurrency control

A server that supports transactions must **synchronize the operations** sufficiently to ensure that the isolation requirement is met. One way of doing this is to perform the transactions **serially** – one at a time, in some arbitrary order. Unfortunately, this solution would generally be unacceptable for servers whose resources are shared by multiple interactive users. For instance, in our banking example it is desirable to allow several bank clients to perform online banking transactions at the same time as one another. The aim for any server that supports transactions is to **maximize concurrency**. Therefore transactions are allowed to execute concurrently if this would have the same effect as a serial execution – that is, if they are **serially equivalent**

or **serializable**. In other words, transactions are serializable if the serial and concurrent run are equivalent. It is necessary to introduce a new entity called **coordinator** that manages concurrency.
The coordinator gives each **transaction an identifier**, or **TID**. The client invokes the `openTransaction` method of the coordinator to start a new transaction – a transaction identifier or TID is allocated and returned. At the end of a transaction, the client invokes the `closeTransaction` method to indicate its end – all of the recoverable objects accessed by the transaction should be saved. If, for some reason, the client wants to abort a transaction, it invokes the `abortTransaction` method – all of its effects should be removed from sight. A transaction is achieved by cooperation between a client program, some recoverable objects and a coordinator. The client specifies the sequence of invocations on recoverable objects that are to comprise a transaction. To achieve this, the client sends with each invocation the transaction identifier returned by `openTransaction`. One way to make this possible is to include an extra argument in each operation of a recoverable object to carry the TID.
The transaction manager has to take care also of these possible issues that can income:

- **lost update**, changing data not correctly managed, and that determines the loss of the update of a result. Reading old values and use for execution.

transaction *T* :	**transaction *U* :**
balance = b.getBalance(); *b.setBalance(balance*1.1);* *a.withdraw(balance/10)*	*balance = b.getBalance();* *b.setBalance(balance*1.1);* *c.withdraw(balance/10)*
balance = b.getBalance(); $200	
	balance = b.getBalance(); $200
	*b.setBalance(balance*1.1);* $220
*b.setBalance(balance*1.1);* $220	
a.withdraw(balance/10) $80	
	c.withdraw(balance/10) $280

A=100, B=200, C=300 -> 10% twice: $242
Update of U is lost

Figure 64: Lost update

- **retrieval disparity** or **inconsistent retrieval**, data values are not consistent for all the copies. If shared data change is not atomic, it is necessary to deal also copies and also inconsistent values derived from a possible recovery.

transaction *V* :		**transaction *W*:**	
a.withdraw(100) *b.deposit(100)*		*aBranch.branchTotal()*	
a.withdraw(100);	$100		
		total = a.getBalance()	$100
		total = total+b.getBalance()	$300
		total = total+c.getBalance()	
b.deposit(100)	$300	⋮	

A=B=200 -> total: $400
Use of inconsistent information

Figure 65: Incosistent retrieval

12.2 Serial equivalence

As we anticipate before, we have serially equivalent transactions only if the concurrent and serial execution lead to the same results. If the transactions **interfere** one can define a combination of operations in the sequences (interleaving) such that it is **serially equivalent**. This means that transactions have the same values of variables to be read and produce the same results at the end of the execution.
An analysis of serial equivalence between transactions is so useful for avoiding lost update and inconsistency.

transaction *T*:		**transaction *U*:**	
balance = b.getBalance()		*balance = b.getBalance()*	
*b.setBalance(balance*1.1)*		*b.setBalance(balance*1.1)*	
a.withdraw(balance/10)		*c.withdraw(balance/10)*	
balance = b.getBalance()	$200		
*b.setBalance(balance*1.1)*	$220		
		balance = b.getBalance()	$220
		*b.setBalance(balance*1.1)*	$242
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$278

T works on B before that U reads

Figure 66: Serially equivalent

12.3 Conflict of operations

Serial equivalence is not satisfied in presence of **conflict operations**. The operations are in **conflict** if the result(effect) of the combined execution depends on the execution order. The following image contains the compatibility table between operations:

Operations of different transactions		*Conflict*	*Conflict*
read	*read*	No	Because the effect of a pair of *read* operations does not depend on the order in which they are executed
read	*write*	Yes	Because the effect of a *read* and a *write* operation depends on the order of their execution
write	*write*	Yes	Because the effect of a pair of *write* operations depends on the order of their execution

Figure 67: Conflict Rules

From this we can conclude that:

Two transactions are serially equivalent $\iff$ *all the operation pairs in conflict are executed with the same order on all the objects they refer to.*

Serial equivalence specify a set of rules to define concurrency control protocol between transactions.
Concurrency can be managed using 3 different ways:

- **Lock**
- **Optimistic**
- **Timestamp - ordering**

transaction *T*:	**transaction *U*:**
x = read(i)	
write(i, 10)	
	y = read(j)
	write(j, 30)
write(j, 20)	
	z = read (i)

The pairs of operations with conflict do not access with the same order to the two objects i and j

T uses i before U,
U uses j before T

Figure 68: Non-serially equivalent interleaving operations

The execution of a set of transactions even serial equivalent is not free from the problem of **dirty reads** (incorrect) due to execution without success of transactions (**abort**). The solution consists to delay the commit of the transactions, and not commit it immediately.
Other events that can bring some wrong effects to the results are given by the aborting operation of a transaction or by multiple `write` operations to the same object (**premature writes**) . The execution of a transaction with abort can cause other abort (**domino effect**). A way to solve this problem consists on imposing that the `read` of objects happens only if the previous `write` on that object have been executed by completed transactions (commit). Or another strategy consists on delaying the execution of `read`. The resolution of this problem results fundamental in order to be able of recovering the previous state (undoing operations). A good implementation consists on recovering the previous state using of **before images** for all the write operations in each transaction. In order to guarantee correctness: one postpones the execution of a write until the transaction that made the object update (commit or abort) just before completes.

Generally, it is required that transactions delay both their read and write operations so as to avoid both **dirty reads** and **premature writes**. The executions of transactions are called **strict** if the service delays both read and write operations on an object until all transactions that previously wrote that object have either committed or aborted. The **strict execution** of transactions enforces the desired property of isolation.

12.4 Nested transactions

More complicated transactions, *top-level*, can be composed by other different transactions, *sub-transactions*, like modules. A subtransaction appears atomic to its parent with respect to transaction failures and to concurrent access. Subtransactions at the same level, such as T_1 and T_2, can run concurrently, but their access to common objects is serialized. If one or more of the subtransactions fails, it doesn't mean that also the top-level transaction fails but the parent transaction could record the fact and then commit, with the result that all the successful child transactions commit. For example, a transaction to deliver a mail message to a list of recipients could be structured as a set of subtransactions, each of which delivers the message to one of the recipients.
The advantages of splitting a transaction in multiple one are: increasing of performance, since transaction at the same level can be executed in parallel and fault-tolerance, not the entire transaction fails but only subtransactions,

Figure 69: Nested transactions

The rules for committing of nested transactions are provided as follow:

- a transaction may commit or abort only after its child transactions have completed.
- if the parent aborts, all of its subtransactions are aborted.
- if a subtransaction aborts, the parent can decide whether to abort or not.

12.5 Concurrency control: Lock

Transactions must be scheduled so that their effect on shared data is serially equivalent. A server can achieve serial equivalence of transactions by serializing access to the objects. The most common used solution consist on **locking** the access to the shared objects. The idea of **lock** is to provide access with **mutual exclusion**, only once at a time, to objects that are modified.

All the pairs of operations of two transactions in conflict have to be executed with the same order. Each transaction does not get a new lock after having released another lock and in order to avoid dirty reads and premature writes each lock is maintained up to the transaction end. Lock can be assigned considering the following lock compatibility:

For one object		*Lock requested*	
		read	*write*
Lock already set	*none*	OK	OK
	read	OK	wait
	write	wait	wait

Figure 70: Lock compatibility matrix

The operation conflict rules tell us that:

- If a transaction T has already performed a read operation on a particular object, then a concurrent transaction U must not write that object until T commits or aborts.
- If a transaction T has already performed a write operation on a particular object, then a concurrent transaction U must not read or write that object until T commits or aborts.

12.5.1 Strict two-phase locking

Strict two-phase locking is a basic algorithm that provides concurrency control using lock strategy.

When an operation accesses an object within a transaction:

- if the object is not already locked, it is locked and the operation proceeds.
- if the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
- if the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
- if the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds. Where promotion is prevented by a conflicting lock rule 2 is applied.

For nested transactions it is necessary that each set of nested T does not see partial results of other sets of nested T (for atomicity property) and each T in the set of the nested T must not see partial results of other T in the set itself.
Every lock that is acquired by a successful subtransaction is inherited by its parent when it completes. Inherited locks are also inherited by ancestors. Note that this form of inheritance passes from child to parent! The top-level transaction eventually inherits all of the locks that were acquired by successful subtransactions at any depth in a nested transaction. This ensures that the locks can be held until the top-level transaction has committed or aborted, which prevents members of different sets of nested transactions observing one another's partial effects. The subtransactions are not concurrently executed with the father, but inherits the needed lock temporary.
Now the following rules regards the allocation and release of lock in nested transactions:

- a subtransaction acquires a **read-lock** if no transactions active has a write lock and who has the **write-lock** is his ancestor.

- a subtransaction acquires a write-lock if no transaction active has a read or write lock and who has the **read-lock** or write-lock is his ancestor.
- when a subtransaction **commit** the lock are inherited by the father with the same type.
- when a subtransactions **abort** the lock are lost, if the father has some of them he keeps them.

12.5.2 Deadlock

The usage of lock technique can bring the system in deadlock. **Deadlock** is a state in which each member of a group of transactions is waiting for some other member to release a lock. It can happen from indirect dependency on the objects.

transaction *T*		transaction *U*	
Operation	**Locks**	**Operation**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
•••	waits for *U*s	*a.withdraw(200);*	waits for *T*s
	lock on *B*	•••	lock on *A*
•••		•••	
•••		•••	

Figure 71: Lock deadlock

A **wait-for graph** can be used to represent the waiting relationships between current transactions. In a wait-for graph the nodes represent transactions and the edges represent wait-for relationships between transactions. There is an edge from node T to node U when transaction T is waiting for transaction U to release a lock.
Deadlocks may be detected by finding cycles in the wait-for graph. Having detected a deadlock, a transaction must be selected for abortion to break the cycle.

Figure 72: Wait-for graph

Lock timeouts are a method for resolution of deadlocks that is commonly used. Each lock is given a limited period in which it is invulnerable. After this time, a lock becomes vulnerable. Provided that no other transaction is competing for the object that is locked, an object with a vulnerable lock remains locked. However, if any other transaction is waiting to access the

object protected by a vulnerable lock, the lock is broken (that is, the object is unlocked) and the waiting transaction resumes. The transaction whose lock has been broken is normally aborted. This solution solves the problem of deadlock, but it brings some drawbacks like: overhead, due the fact the timeout wastes time, it can happen that some transactions abort also without actual deadlock, which is very common for long transactions.
Another solution is to **prevent deadlock**, locking all of the objects used by a transaction when it starts. This would need to be done as a single atomic step so as to avoid deadlock at this stage. Such a transaction cannot run into deadlocks with other transactions, but this approach unnecessarily restricts access to shared resources. Deadlocks can also be prevented by requesting locks on objects in a predefined order, but this can result in premature locking and a reduction in concurrency.

12.6 Lock compatibility (read, write and commit locks)

Even when locking rules are based on the conflicts between read and write operations and the granularity at which they are applied is as small as possible, there is still some scope for increasing concurrency. We discuss two approaches that have been used to deal with this issue:

- **Two-version locking:** the setting of exclusive locks is delayed until a transaction commits.
- **Hierarchic locks:** mixed-granularity locks are used.

Two-version locking. This is an optimistic scheme that allows one transaction to `write` tentative versions of objects while other transactions `read` from the committed versions of the same objects. `read` operations only wait if another transaction is currently committing the same object. This scheme allows more concurrency than read-write locks. Transactions cannot commit their `write` operations immediately if other uncompleted transactions have read the same objects. Therefore, transactions that request to commit in such a situation are made to wait until the reading transactions have completed. **Deadlocks** may occur when transactions are waiting to commit. Therefore, transactions may need to be aborted when they are waiting to commit, to resolve deadlocks.

For one object		*Lock to be set*		
		read	*write*	*commit*
Lock already set	*none*	OK	OK	OK
	read	OK	OK	wait
	write	OK	wait	–
	commit	wait	wait	–

Figure 73: Compatibility table for two-version locking.

In the previous image it is possible to see the compatibilities of different locks. This variation on strict two-phase locking uses three types of lock: a `read` lock, a `write` lock and a `commit` lock:

- Before a transaction's **read operation** is performed, a `read` lock must be set on the object. The attempt to set a `read` lock is successful unless the object has a `commit` lock, in which case the transaction waits.

- Before a transaction's **write operation** is performed, a `write` lock must be set on the object. The attempt to set a `write` lock is successful unless the object has a `write` lock or a `commit` lock, in which cases the transaction waits.

Hierarchic locks. Gray, in the 1978, proposed the use of a hierarchy of locks with different granularities. At each level, the setting of a parent lock has the same effect as setting all the equivalent child locks. This economizes on the number of locks to be set. In Gray's scheme, each node in the hierarchy can be locked, giving the owner of the lock explicit access to the node and giving implicit access to its children.

For one object		*Lock to be set*			
		read	*write*	*I-read*	*I-write*
Lock already set	*none*	OK	OK	OK	OK
	read	OK	wait	OK	wait
	write	wait	wait	wait	wait
	I-read	OK	wait	OK	OK
	I-write	wait	wait	OK	OK

Figure 74: Compatibility table for hierarchic locks.

Gray also proposed a third type of intention lock, that combines the properties of a `read` lock with an intention to `write` lock.
Hierarchic locks have the advantage of reducing the number of locks when mixed- granularity locking is required. The compatibility tables and the rules for promoting locks are more complex. The mixed granularity of locks could allow each transaction to lock a portion whose size is chosen according to its needs. A long transaction that accesses many objects could lock the whole collection, whereas a short transaction can lock at finer granularity.

12.7 Concurrency control: Optimistic

Locking is not the unique solution and it has some drawbacks that should be considered:

- For system that does not have concurrent access to shared data locking is inefficient and brings an useless overhead.
- deadlock is difficult to solve and prevent.
- To avoid cascading aborts, locks cannot be released until the end of the transaction. This may reduce significantly the potential for concurrency.

Another solution is given by **optimistic concurrency control**, which assumes that there are no conflicts for the operations. In **optimistic** schemes, a transaction proceeds until it asks to commit, and before it is allowed to commit the server performs a check to discover whether it has performed operations on any objects that conflict with the operations of other concurrent transactions, in which case the server aborts it and the client may restart it. The aim of the check is to ensure that all the objects are correct.
In other words, during **commit time** server performs a check and whether there have been

interferences and in that case it removes the transactions.

Optimistic strategy has no deadlock state and it provides maximum level of concurrency. But the main drawback is that for heavy load the interference probability and conflict is very high.

Transactions pass from these phases:

- **Work phase**, each transaction has a copy called, tentative version, of each object that is updated by a write operation.
- **Validation phase**, when a transaction wants to terminates it is validated to establish whether or not its operations on objects conflict with operations of other transactions on the same objects. If the validation is successful, then the transaction can commit. If the validation fails, then some form of conflict resolution must be used and either the current transaction or, in some cases, those with which it conflicts will need to be aborted. It is performed one at a time.
- **Update phase**, If a transaction is validated, all of the changes recorded in its tentative versions are made permanent. Read-only transactions can commit immediately after passing validation. Write transactions are ready to commit once the tentative versions of the objects have been recorded in permanent storage.

12.7.1 Serializability

When transactions enters into the validation phase they are numbered in increasing order. In order to provide validation of the transaction serializability the following rules are considered:

T_1	T_2	Rule
write	*read*	1. T_2 must not read objects written by T_1
read	*write*	2. T_1 must not read objects written by T_2
write	*write*	3. T_2 must not write objects written by T_1 and T_1 must not write objects written by T_2

Figure 75: Validation rules

If the update phase is exclusive rule 3 is satisfied. Note that this restriction on write operations, together with the fact that no dirty reads can occur, produces strict executions. To prevent overlapping, the entire validation and update phases can be implemented as a critical section so that only one client at a time can execute it. The validation of a transaction must ensure that rules 1 and 2 are obeyed by testing for overlaps between the objects of pairs of transactions T_v and T_i.

Figure 76: Optimistic serializability validation

There two possible strategy to check overlapping:

- **backward**, checks with the previous transactions already evaluated.
- **forward**, compare with future transactions not already evaluated.

Both the two strategies check if the transaction reads a values that was changed later by another one.

12.8 Concurrency control: Timestamp

The last solution that we present on this document is based on the idea of using ordering. To each transaction is associated a **timestamp** (unique label). The timestamp defines its position in the time sequence of transactions. Requests from transactions can be totally ordered according to their timestamps. Each object has a **read timestamp** and **write timestamp** that registers the time of the last transaction that accessed it. So for assumption for each object there is a unique copy.
The object authorizes the operation (read or write) only to transactions with timestamp greater (more recent) of the object timestamp. Transactions that arrive late abort their execution. The validation is so performed immediately and it is simple and quick.
This solution does not present deadlock state. A transaction's request to write an object is valid only if that object was last read and written by earlier transactions. A transaction's request to read an object is valid only if that object was last written by an earlier transaction. The following rules are considered during the validation phase:

Rule	T_c	T_i	
1.	*write*	*read*	T_c must not *write* an object that has been *read* by any T_i where $T_i > T_c$. This requires that $T_c \geq$ the maximum read timestamp of the object.
2.	*write*	*write*	T_c must not *write* an object that has been *written* by any T_i where $T_i > T_c$. This requires that $T_c >$ the write timestamp of the committed object.
3.	*read*	*write*	T_c must not *read* an object that has been *written* by any T_i where $T_i > T_c$. This requires that $T_c >$ the write timestamp of the committed object.

Figure 77: Timestamp control rules

Here we present an example of transactions execution and validation:

		Timestamps and versions of objects					
T	*U*	*A*		*B*		*C*	
		RTS	*WTS*	*RTS*	*WTS*	*RTS*	*WTS*
		{}	***S***	{}	***S***	{}	***S***
openTransaction							
bal = b.getBalance()				*{T}*			
	openTransaction						
*b.setBalance(bal*1.1)*					***S***, *T*		
	bal = b.getBalance()						
	wait for T						
a.withdraw(bal/10)	•••		***S***, *T*				
commit	•••		***T***		***T***		
	bal = b.getBalance()			*{U}*			
	*b.setBalance(bal*1.1)*				***T***, *U*		
	c.withdraw(bal/10)						***S***, *U*

Figure 78: Timestamp in transactions T and U

13 Distributed Transaction

A transaction becomes **distributed** if it invokes operations in several different servers. So instead of having a single server there is a set of them that execute the transaction operations. These servers coordinate to guarantee the same result, and sometimes this operation is not so easy such that it will be necessary to introduce a coordinator entity or implement different coordination protocol. There are two different ways that distributed transactions can be structured:

- **flat transactions**, are composed by a set of operations that are executed sequentially. Each request is completed before going on to the next one. When servers use locking, a transaction can only be waiting for one object at a time.
- **nested transactions**, are composed by other transactions, called subtransactions, which can run concurrently if they are at the same level.

Figure 79: Distributed transactions

Considering the previous picture the transaction on the left there is a flat transaction that invokes operations sequentially on objects in servers X, Y and Z. On the right there is a nested transaction composed by two subtransactions. The subtransactions T_1 and T_2 open further subtransactions T_{11}, T_{12}, T_{21}, and T_{22}, which access objects at servers M, N and P. In the nested case, subtransactions at the same level can run concurrently, so T_1 and T_2 are concurrent, and as they invoke objects in different servers, they can run in parallel. The four subtransactions T_{11}, T_{12}, T_{21} and T_{22} also run concurrently.

Figure 80: Nested banking transaction

13.1 Coordinator of a distributed transaction

Servers that execute requests as part of a distributed transaction need to be able to communicate with one another to coordinate their actions when the transaction commits. A client starts a transaction by sending an `openTransaction` request to a **coordinator**. The coordinator returns the resulting **transaction identifier** (TID) to the client, it can be for example a pair (server ID, #transaction local). Transaction identifiers for distributed transactions must be unique within a distributed system. At the end is responsible for committing or aborting it. Each server that manages an object used by a transaction is called **participant** in the transaction. Each participant:

- It executes operations of the transaction T.
- It is responsible for keeping track of all of the recoverable objects at that server that are involved in T.
- It is responsible for cooperating with the coordinator to complete the *commit* or call a `abortTransaction`.

During the progress of the transaction, the coordinator records a list of references to the participants, and each participant records a reference to the coordinator. Each time that a new participant is added, it executes a `join` to inform the coordinator that a new server is executing the transaction.

Figure 81: Coordinator

13.2 Atomicity

When a distributed transaction comes to an end, the **atomicity** property of transactions requires that either all of the servers involved commit the transaction or all of them abort the transaction. To achieve this, one of the servers takes on a coordinator role, which involves ensuring the same outcome at all of the servers. The manner in which the coordinator achieves this depends on the protocol chosen.

There are essentially two protocols that try to guarantee atomicity among all the servers:

- **One phase commit**, the coordinator informs all the participants of the result (commit/abort) and waits for the ack from each of them. It repeats the operation up to completion. But in this way it can happen that atomicity is not satisfied. In fact, if the client executes `commit` it does not allow a server to abort if it is necessary or it does not allow a coordinator to detect possible server faults.
- **Two phase commit**, all the participants can do `abort`. The client asks the coordinator to execute the `commit` and activate protocol two phase commit.

Two phase commit is the most common and used algorithm and it is composed essentially by two distinct phase:

- Phase 1: **voting**, coordinator asks if all are ready to commit.
- Phase 2: **decision**, communicates `commit` only if all are ready.

This protocol allows the servers to communicate with one another to reach a joint decision as to whether to commit or abort. In general: If any one participant votes to abort, then the decision must be to abort the transaction, otherwise If all the participants vote to commit, then the decision is to commit the transaction.
The problem is to ensure that all of the participants vote and that they all reach the same decision.

13.3 Two phase commit protocol

The two-phase commit protocol is implemented following the following algorithm:

- Phase 1 (**voting phase**): The coordinator sends a `canCommit?` request to each of the participants in the transaction. When a participant receives a `canCommit?` requests it replies with its vote (Yes or No) to the coordinator. Before voting Yes, it prepares to commit by saving objects in permanent storage. If the vote is No the participant aborts immediately.
- Phase 2 (**decision**): The coordinator collects the votes, if there are no failures and all the votes are Yes the coordinator decides to commit the transaction and sends a `doCommit` request to each of the participants.
  Otherwise the coordinator decides to abort the transaction and send `doAbort` requests to all participants that voted Yes.
  Participants that voted Yes are waiting for a `doCommit` or `doAbort` request from the coordinator. When a participant receives one of these messages it acts accordingly and in the case of commit, makes a `haveCommitted` call as confirmation to the coordinator.

Figure 82: Two Phase commit functions

The general behavior can be summarized with the following schema:

Figure 83: Communication in two phase commit protocol

13.4 Fault and performance management

There are various stages in the protocol at which the coordinator or a participant cannot progress its part of the protocol until it receives another request or reply from one of the others. In order to avoid that some entities wait for a long time without any probability to get an answer it is introduced the usage of **timeouts**. For instance a participant that is waiting for a `doCommit` message set a timeout, if the timeout fires it will abort the transaction.
The critic point for fault management happens if the coordinator crashes after that it sent `doCommit` to all the partecipants. The problem consists on the fact that later it will be no able to check if all the participants have commited or not.

Considering the performance offered by the algorithm. Provided there is no fault, the two-phase commit protocol involving N participants can be completed with N `canCommit?` messages and replies, followed by N `doCommit` messages. That is, the cost in messages is proportional to $3N$, and the cost in time is three rounds of messages. The `haveCommitted` messages are not counted in the estimated cost of the protocol, which can work correctly without them – their role is to enable servers to delete stale coordinator information. In the worst case, there may be arbitrarily many server and communication failures during the two-phase commit protocol.

13.5 Two phase commit protocol for nested transactions

As we have seen before nested transactions are composed by top-level transaction and subtransactions. When a subtransaction completes, it makes an independent decision either to commit provisionally or to abort. A **provisional commit** is different from being preared to commit: nothing is backed up in permanent storage. If the server crashes subsequently, its replacement will not be able to commit. After all subtransactions have completed, the provisionally committed ones participate in a two-phase commit protocol, in which servers of provisionally committed subtransactions express their intention to commit and those with an aborted ancestor will abort. In other words provisional commit means that coordinator and transactional servers are ready to commit subtransaction but they have not yet done so.

Figure 84: Distributed nested transaction

From the previous picture we can see that even if T_{11} abort T_1 in this example is in provisional commit state. It can happen if the transaction T_{11} is not considered fundamental: example multiple thread to find an element, if at least one of this thread find the element I can ignore the other ones. But there are also case in which T_{11} plays a fundamental role, and in this case T_1 should abort.

The top-level transaction plays the role of coordinator in the two-phase commit protocol. The second phase of the two-phase commit protocol is the same as for the non- nested case. The

coordinator collects the votes and then informs the participants as to the outcome. When it is complete, coordinator and participants will have committed or aborted their transactions.

13.6 Protocol realization

Two phase commit protocol can be realized as:

- **Flat**, the coordinator of the top-level transaction sends `canCommit?` messages to the coordinators of all of the subtransactions in the **provisional commit list**. Partecipants that receive the message replies with it vote Yes or No if they can commit or not. The participant check the list to verify whether a subtransaction already provisional commited is descendent of a transaction already aborted. In this case also the subtransaction has to abort.
- **Hierarchical**, the coordinator of the top-level transaction communicates with the coordinators of the subtransactions for which it is the immediate parent. It sends `canCommit?` messages to each of them, which in turn pass them on to the coordinators of their child transactions. Each participant collects the replies from its descendants before replying to its parent.

13.7 Concurrency control for distributed transactions

The servers apply to their own objects the protocols of concurrency control for distributed transactions like **locking**, **optimistic** and **timestamp**. Servers must coordinate to ensure serial equivalence. This implies that if transaction T is before transaction U in their conflicting access to objects at one of the servers, then they must be in that order at all of the servers whose objects are accessed in a conflicting manner by both T and U.

Locking. In a distributed transaction, the locks on an object are held locally (in the same server). The **local lock manager** can decide whether to grant a lock or to make the requesting transaction wait. However, it cannot release any locks until it knows that the transaction has been committed or aborted at all the servers involved in the transaction.
When locking is used for concurrency control, the objects remain locked and they are unavailable for other transactions during the atomic commit protocol. Since each server manages the locks locally and independently from the others, it is possible that different servers may impose different orderings on transactions. These different orderings can lead to cyclic dependencies between transactions, giving rise to a **distributed deadlock** situation.

T			*U*		
write(A)	at *X*	locks *A*			
			write(B)	at *Y*	locks *B*
read(B)	at *Y*	waits for *U*			
			read(A)	at *X*	waits for *T*

Figure 85: Locking in distributed transactions

Timestamp. In distributed transactions, we require that each coordinator provides **globally unique timestamps**. A globally unique transaction *timestamp* is provided to the client by the coordinator the first time. The transaction timestamp is passed to the coordinator at each server whose objects perform an operation in the transaction.

To achieve the same ordering at all the servers, the coordinators must agree as to the ordering of their timestamps. A *timestamp* consists of a $< local\ timestamp, server - id >$ pair.
When *timestamp* ordering is used for concurrency control, conflicts are resolved with a possible abort.

Optimistic. With *optimistic* concurrency control, each transaction is validated before *commit execution*. Transaction numbers are assigned at the beginning of the validation phase and transactions are serialized according to the order of their transaction numbers. A distributed transaction is validated by a collection of independent servers, each of which validates transactions that access its own objects. This validation takes place during the first phase of the *two-phase commit protocol*.

T		*U*	
read(A)	at *X*	*read(B)*	at *Y*
write(A)		*write(B)*	
read(B)	at *Y*	*read(A)*	at *X*
write(B)		*write(A)*	

Figure 86: Example of commitment deadlock with optimistic concurrency control

As it is possible to see in the previous image, transactions access the objects in the order T before U at server X and in the order U before T at server Y. Now if we suppose that T and U start validation at about the same time, but server X validates T first and server Y validates U first. From a previous rule of the validation protocol, we know that only one transaction may perform validation and update phases at a time. Therefore each server will be unable to validate the other transaction until the first one has completed. This is an example of **commitment deadlock**. If parallel validation is used, transactions will not suffer from *commitment deadlock*, but this imply that only one transaction at a time can enter during the validation phase.

13.8 Distributed deadlock

Most deadlock detection schemes operate by finding cycles in the transaction *wait-for graph*. In a distributed system involving multiple servers being accessed by multiple transactions, a global *wait-for graph* can in theory be constructed from the local ones. There can be a cycle in the global *wait-for graph* that is not in any single local one, there can be in fact a **distributed deadlock**. We can understand so that, to detect a *distributed deadlock*, it is required to find a cycle in the global transaction *wait-for graph* that is distributed among the servers that were involved in the transactions. An example of the *global wait-for graph* is the following:

Figure 87: Example of a global *wait-for graph*

A simple solution is to use **centralized deadlock detection**, in which one server takes on the role of **global deadlock detector**. When it finds a cycle, it makes a decision on how to resolve the deadlock and tells the servers which transaction to abort. Centralized deadlock detection is not a good idea, because it depends on a single server to carry it out. It suffers from the usual problems associated with centralized solutions in distributed systems.

Phantom deadlock. Using this strategy it can be possible to find a phantom deadlock. A deadlock that is *"detected"* but it is not really a deadlock is called a **phantom deadlock**. In a distributed deadlock detection, as the procedure will take some time, there is a chance that one of the transactions that holds a lock will meanwhile have released it, in which case the deadlock will no longer exist.

Edge chasing. A distributed approach to deadlock detection uses a technique called **edge chasing**. In this approach, the *global wait-for graph* is not constructed, but each of the servers involved has knowledge about some of its edges. The servers attempt to find cycles by forwarding messages called **probes**, which follow the edges of the graph throughout the distributed system. A *probe* message consists of transaction wait-for relationships representing a path in the *global wait-for graph*. When a cycle is detected, a transaction in the cycle is aborted to break the deadlock. **Transaction priorities** is a variant of edge chasing that reduces the number of probe messages.

13.9 Transaction recovery

The atomic property of transactions requires that all the effects of committed transactions and none of the effects of incomplete or aborted transactions are reflected in the objects they accessed. This property can be described in terms of two aspects: **durability** and **failure atomicity**. *Durability* requires that objects are saved in permanent storage and will be available indefinitely thereafter. *Failure atomicity* requires that effects of transactions are atomic even when the server crashes.

We will assume that when a server is running it keeps all of its objects in its volatile memory and records its committed objects in a **recovery file**. Therefore recovery consists of restoring the server with the latest committed versions of its objects from permanent storage.
The requirements for *durability* and *failure atomicity* are not really independent of one another and can be dealt with by a single mechanism, the **recovery manager**.
The tasks of a **recovery manager** are:

- To save objects in permanent storage (in a recovery file) for committed transactions.
- To restore the server's objects after a crash.
- To reorganize the recovery file to improve the performance of recovery.
- To manage recovery file.

Any server that provides transactions needs to keep track of the objects accessed by clients' transactions. At each server, an **intentions list** is recorded for all of its currently active transactions. An *intentions list* of a particular transaction contains a list of the references and the values of all the objects that are modified by that transaction. When a transaction is committed, that transaction's *intentions list* is used to identify the objects it affected. Each object in the *recovery file* is associated with a particular transaction by saving the intentions list in the *recovery file*. Following table shows a summary of the types of entry included in a *recovery file*:

Type of entry	**Description of contents of entry**
Object	A value of an object.
Transaction status	Transaction identifier, transaction status (*prepared*, *committed*, *aborted*) and other status values used for the two-phase commit protocol.
Intentions list	Transaction identifier and a sequence of intentions, each of which consists of $<objectID, p_i>$, where P_i is the position in the recovery of the value of the object.

Table 3: Types of entry in a recovery file.

Two approaches can be used to create a *recovery file*: **logging** and **shadow version**.

13.9.1 Logging

In the logging technique, the *recovery file* represents a log containing the **history** of all the transactions performed by a server. The *history* consists of values of objects, transaction status entries and transaction intentions lists. During the normal operation of a server, its recovery manager is called whenever a transaction prepares to commit, commits or aborts a transaction. When the server is prepared to commit a transaction, the recovery manager appends all the objects in its intentions list to the recovery file, followed by the current status of that transaction together with its intentions list. When a transaction is eventually committed or aborted, the recovery manager appends the corresponding status of the transaction to its recovery file. After a crash, any transaction that does not have a committed status in the log is aborted.

The below image illustrates the log mechanism for the banking service transactions T and U. Entries to the left of the double line represent a snapshot of the values of A, B and C before transactions T and Ustarted. In this diagram, we use the names A, B and C as unique identifiers for objects. We show the situation when transaction T has committed and transaction U has prepared but not committed. When transaction T prepares to commit, the values of objects A

and B are written at positions P_1 and P_2 in the log, followed by a prepared transaction status entry for T with its intentions list $(< A, P_1 >, < B, P_2 >)$. When transaction T commits, a committed transaction status entry for T is put at position P_4. Then when transaction U prepares to commit, the values of objects C and B are written at positions P_5 and P_6 in the log, followed by a prepared transaction status entry for U with its intentions list $(< C, P_5 >, < B, P_6 >)$. Each transaction status entry contains a pointer to the position in the recovery file of the previous transaction status entry to enable the recovery manager to follow the transaction status entries in reverse order through the recovery file. The last pointer in the sequence of transaction status entries points to the checkpoint.

Figure 88: Example of log for banking service

13.9.2 Shadow versions

The logging technique records transaction status entries, intentions lists and objects all in the same file: the **log**. The **shadow versions** technique is an alternative way to organize a recovery file. It uses a **map** to locate versions of the server's objects in a file called a version store. The map associates the identifiers of the server's objects with the positions of their current versions in the version store. The versions written by each transaction are *"shadows"* of the previous committed versions.
When a transaction is prepared to commit, any of the objects changed by the transaction are appended to the version store, leaving the corresponding committed versions unchanged. These new as-yet-tentative versions are called shadow versions. When a transaction commits, a new map is made by copying the old map and entering the positions of the shadow versions. To complete the commit process, the new map replaces the old map.
To restore the objects when a server is replaced after a crash, its recovery manager reads the map and uses the information in the map to locate the objects in the version store. The below image illustrates this technique with the same example involving transactions T and U. The first column in the table shows the map before transactions T and U, when the balances of the accounts A, B and C are \$100, \$200 and \$300, respectively. The second column shows the map after transaction T has committed.

	Map at start				*Map when T commits*		
	$A \rightarrow P_0$				$A \rightarrow P_1$		
	$B \rightarrow P_0'$				$B \rightarrow P_2$		
	$C \rightarrow P_0''$				$C \rightarrow P_0''$		
	P_0	P_0'	P_0''	P_1	P_2	P_3	P_4
Version store	100	200	300	80	220	278	242
		Checkpoint					

Figure 89: Example of log for banking service

The version store contains a checkpoint, followed by the versions of A and B at P_1 and P_2 made by transaction T. It also contains the shadow versions of B and C made by transaction U, at P_3 and P_4. The map must always be written to a well-known place so that it can be found when the system needs to be recovered. The switch from the old map to the new map must be performed in a single atomic step.

13.9.3 Recovery of the two-phase commit protocol

The recovery management described previously, must be extended to deal with any transactions that are performing the *two-phase commit protocol* at the time when a server fails. The recovery managers use two new status values for this purpose: `done` and `uncertain`. A **coordinator** uses committed to indicate that the outcome of the vote is `Yes` and `done` to indicate that the *two-phase commit protocol* is complete. A participant uses `uncertain` to indicate that it has voted `Yes` but does not yet know the outcome of the vote.
In **phase 1** of the protocol, when the *coordinator* is prepared to commit, its recovery manager adds a coordinator entry to its recovery file. When it votes `Yes`, its recovery manager records a participant entry and adds an `uncertain` transaction status to its recovery file as a forced write. When a participant votes `No`, it adds an `abort` transaction status to its recovery file.
In phase 2 of the protocol, the recovery manager of the coordinator adds either a *committed* or an *aborted* transaction status to its recovery file, according to the decision.

Role	**Status**	**Action of recovery manager**
Coordinator	*prepared*	No decision had been reached before the server failed. It sends *abortTransaction* to all the servers in the participant list and adds the transaction status *aborted* in its recovery file. Same action for state *aborted*. If there is no participant list, the participants will eventually timeout and abort the transaction.
Coordinator	*committed*	A decision to commit had been reached before the server failed. It sends a *doCommit* to all the participants in its participant list (in case it had not done so before) and resumes the two-phase protocol at step 4.
Participant	*committed*	The participant sends a *haveCommitted* message to the coordinator (in case this was not done before it failed). This will allow the coordinator to discard information about this transaction at the next checkpoint.
Participant	*uncertain*	The participant failed before it knew the outcome of the transaction. It cannot determine the status of the transaction until the coordinator informs it of the decision. It will send a *getDecision* to the coordinator to determine the status of the transaction. When it receives the reply it will commit or abort accordingly.
Participant	*prepared*	The participant has not yet voted and can abort the transaction.
Coordinator	*done*	No action is required.

Table 4: Table for the different actions of the recovery manager.

14 Distributed File System

A distributed file system enables programs to store and access remote files exactly as they do local ones, allowing users to access files from any computer on a network. The performance and reliability experienced for access to files stored at a server should be comparable to that for files stored on local disks. Distributed file systems support the sharing of information in the form of files and hardware resources in the form of persistent storage throughout a network.
Motivation for using distributed file system:

- **availability**
- **fault tolerance**
- **performance**
- **heterogeneity**

A distributed file system in general offers the following functionalities:

- Sharing of resources
- Persistence for distributed shared objects.
- Distributed replicas of resources
- Consistency of resources, between multiple copies of data when updates occur. But respect to local file system it provides slightly weaker guarantees.

14.1 File system modules

The main role of a file system is to deal and manage files around the system. Each file contains data and a set of attributes used to make identify them or taking some decisions. File systems are responsible for the organization, storage, retrieval, naming, sharing and protection of files. They provide a programming interface that characterizes the file abstraction, freeing programmers from concern with the details of storage allocation and layout. Files contain both data and attributes. The **data** consist of a sequence of data items, accessible by operations to read and write any portion of the sequence. The **attributes** are held as a single record containing information such as the length of the file, timestamps, file type, owner's identity and access control lists. The shaded attributes are managed by the file system and are not normally updated by user programs.
And it is organized in modules:

Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	disk I/O and buffering

Figure 90: File system modules

File length
Creation timestamp
Read timestamp
Write timestamp
Attribute timestamp
Reference count
Owner
File type
Access control list

Figure 91: File attributes

14.2 Distributed File System requirements

In order to implement an useful distributed file system it is necessary to consider the following requirements:

- **Transparency**, the design must balance the flexibility and scalability that derive from transparency against software complexity and performance. The following forms of transparency are provided:
  - *Access transparency*, clients does not have knowledge on how to resources are accessed by the system.
  - *Location transparency*, clients are not affected by possible change of locations of resources.
  - *Performance transparency*, clients see always the same level of performance.
  - *Scalability transparency*, the service can be expanded by incremental growth to deal with a wide range of loads and network sizes.
  - *Mobility transparency*, if files are moved clients are not affected.
- **Concurrency**, changes to a file by one client should not interfere with the operation of other clients simultaneously accessing or changing the same file.
- **Replication**, in a file service that supports replication, a file may be represented by several copies of its contents at different locations. This has two benefits – it enables multiple servers to share the load of providing a service to clients accessing the same set of files, enhancing the scalability of the service, and it enhances fault tolerance by enabling clients to locate another server that holds a copy of the file when one has failed.
- **Heterogeneity**, The service interfaces should be defined so that client and server software can be implemented for different operating systems and computers.
- **Consistency**, when files are replicated or cached at different sites, there is an inevitable delay in the propagation of modifications made at one site to all of the other sites that hold copies, and this may result in some deviation from one-copy semantics.
- **Fault Tolerance**, the central role of the file service in distributed systems makes it essential that the service continues to operate in face of client and server failures.
- **Security**, there is a need to authenticate client requests so that access control at the server is based on correct user identities and to protect the contents of request and reply messages with digital signatures and encryption of secret data.
- **Efficiency**, a distributed file service should offer facilities that are of at least the same power and generality as those found in conventional file systems and should achieve a comparable level of performance.

14.3 Case of study

In this section we are going to study some particular file systems implementation.

14.3.1 File service architecture

The architecture is designed to enable a stateless implementation of the server module. It structures the file service as three components **flat file service**, a **directory service** and a **client module**.

Figure 92: File Service Architecture

The **flat file service** is concerned with implementing operations on the contents of files. *Unique file identifiers* are used to refer to files in all requests for flat file service operations. The implemented operations are idempotent.
The **directory service** provides a mapping between text names for files and their UFIDs. The directory service provides the functions needed to generate directories, to add new file names to directories and to obtain UFIDs from directories.
The **client module** also holds information about the network locations of the flat file server and directory server processes.

14.3.2 Sun Network File System

Sun Network File System is implemented though **NFS protocol**, which is a set of remote procedure calls that provide the means for clients for perform operations on a remote file store. NFS provides access transparency, user programs can issue file operations for local or remote files without distinction. It is composed by NFS Server and NFS Client, that communicate using remote procedure calls.
The **NFS Server module** is stored in the kernel on each computer that acts as an NFS server. Requests referring to files in a remote file system are translated by the client module to NFS protocol operations and then passed to the NFS server module at the computer holding the relevant file system. The **NFS client module** cooperates with the virtual file system in each client machine. It operates in a similar manner to the conventional UNIX file system, transferring blocks of files to and from the server and caching the blocks in the local memory whenever possible.

Figure 93: Sun Network File System

To use a remote file system the process uses `mount service` available on every server NFS. Each

server has to sign the list of the exportable file systems, and each client can mount them using `mount` command specifing server name and the paths.
Caching in both the client and the server computer are indispensable features of NFS implementations in order to achieve adequate performance
The server keeps in local memory blocks of files belonging to the exported file system. It is necessary to deal with consistency of data, and for that reason two strategies are adopted:

- **Write Through**, block are written on the disk when the server receives the write request by a client.
- **Write back**, blocks are not immediately written on the disk at the receiving time of the write request. Client can invoke the primitive `commit` to ask to the server to write all the blocks in the cache.

The client module caches the results of read and write operations, reducing the request number to the server. The main problem is that there can be different versions of the same file in different nodes, and so the solution is to assign the client the responsibility to check for data consistency in its cache. For every access to a shared file one has to verify the **validity cache condition**. Each element in the cache of the client has T_c, time of last validation, T_m, time of last modification. Chosen a threshold t it is necessary to verify that the time from last validation must not be greater than t.

14.3.3 Andrew File System

Like NFS allows the applications to access remote file systems without recompilation. AFS with respect to NFS has more **scalability** and it allows the following main features:

- Files are entirely transferred by the server to the clients in one operation.
- Clients keep in their own a cache local copy of the file (entire) received by the server.

AFS is designed to perform well with larger numbers of active users than other distributed file systems. The key strategy for achieving scalability is the caching of whole files in client nodes.

AFS is implemented as two software components that exist as UNIX processes called **Vice** and **Venus**.

Figure 94: Andrew file system architecture

Venus is a client working at upper level in each workstation, and Vice is a server that works at upper level. In other words: Vice is the name given to the server software that runs as a user-level UNIX process in each server computer, and Venus is a user-level process that runs in each client computer and corresponds to the client module in our abstract model. The file system of each workstation is formed by local files, not shared, and shared files, stored on the server and copies are cached on the local disks of workstations.
When a process in a workstation client open a remote file, and if there is no copy in the local cache the file is localized and a copy is required. That copy is stored in the local file system. Successive read/write operations of the process work on the local copy.
When the process close the file: if the local copy has been modified then it is sent back to the server, otherwise the copy is kept in the workstation for possible successive requests. In general read and write operations are not so frequent and the local cache is very large.
Andrew file system assumes:

- most of the file are small
- read is more frequent than write
- sequential access is more common
- most of the file are modified only by one client
- file are often used as burst.

So it could be not reasonable to use Andrew file system for database.
One of the most important problem is given by **Consistency of the cache**. When a module Vice (server) gives a file to Venus (client) it also sends a **callback promise**, which is a promise of the server to contact later the client when the file is modified by other clients. In the client cache to each file there is the callback associated and it can be in two possible states: **valid**, the server has not yet communicated the file changes, **cancelled**, the server did a recall communicating to invalidate the local copy of the file because of changes. When a server receives the change request of a file, it send a message to all those clients to which it previously promised a **callback**. Once connected by the server, the client put the callback associated to that file as cancelled. AFS is not stateless respect to NFS.

Another aspect that should be considered is the *fault tolerance*. When a workstation starts again after crash it must try to keep most of the content of its cache. It could have lost some callback by the server and so the cache has to be validated again. **Revalidation** of the cache works as follow:

- the client sends a cache validation request with the file to be validated and the date of the last change.
- if the server verifies that there were no changes starting from the date indicated by the client, then the file and the corresponding callback is validated.
- Otherwise the callback is set as cancelled.

In any case, each client has to validate again a callback once some time elapsed without receiving any server notification. Note also that this consistency strategy is applied only during `open` or `close` operations, and it is possible that different clients try to change at the same file so producing incosistent results.

15 Replication

In this section, we will analyze the *replication* of data, considered also as the management of a set of data copies among multiple computers. The motivations that lead us to discuss about this specific topics are the following:

- **Availability:** users require services to be highly available. A service should be accessible with reasonable response times about in the 100% of the cases. The factors that are relevant to high availability are:
  - *Server failures* in this case replication is a technique for automatically maintaining the availability of data despite server failures. If each of n servers has an independent probability p of failing or becoming unreachable, then the availability of an object stored at each of these servers is:

$$1 - p(\text{all servers failed or unreachable}) = 1 - p^n$$

  - *Network partitions and disconnected operation.* Mobile users may deliberately disconnect their computers or become unintentionally disconnected from a wireless network as they move around.
- **Fault tolerance:** a fault-tolerant service always guarantees strictly correct behavior or data despite a certain number and type of faults.
- **Performance:** the caching of data at clients and servers is by now familiar as a means of performance enhancement. For *stable* data, replication management is simple, instead, for *variable* data, one has to introduce protocols to guarantee consistency.

Traditionally, in a distributed system, data are shared among several machines, for this reason, processes can work on different copies distributed in a **data store**. Since, there's many data replication, it is present the need of ensure read and write consistency.

Consistency models can be considered as special agreements between processes and data store. In the following pages we will see several examples of consistency.

Figure 95: Representation of a data store distributed in several nodes.

15.1 Consistency models

Consistency models specifies a contract between programmer and system, wherein the system guarantees that if the programmer follows the rules, memory will be consistent and the results of reading, writing, or updating memory will be predictable. The expected behavior in a system is that a *read* gets the result of the last *write*. Now, it will be possible to analyze different types of consistency models.

15.1.1 Strict consistency

Each *read* of a data x returns the value corresponding to the result of the last *write*. The main features of this particular model are:

- It makes implicit assumptions of a global time.
- All the *write* are ordered.
- Every *write* is immediately visible to all.

Figure 96: Example of Strict consistency.

Strict consistency is the strongest consistency model. Under this model, a write to a variable by any processor needs to be seen instantaneously by all processors. In practice it cannot be implemented in a distributed system. In other words, this specific model can be used only in **uniprocessor** systems.

15.1.2 Sequential consistency

Results are the same as if all the processes were sequential and the operations of each process were in the specified order. Its features are the following:

- It's not time based.
- Internal operations cannot be differently ordered.
- All see the same interleaving and, as a consequence, all reads must have the same sequence.

Figure 97: Example of Sequential consistency.

Linearizable. we denote $ts_{OP}(x)$ as the *timestamp* of operation OP executed on data x, where OP can be both a read or a write operation. We can define the linearizable property as:

If $ts_{OP1(x)} < ts_{OP2(y)}$ then operation $OP1(x)$ precedes $OP2(y)$ in the operations sequence.

Sequential consistency can be considered as the serialization of the transactions but with different granularity, then, it is possible to conclude that:

Strict consistency → Linearizable → Sequential consistency

15.2 Causal consistency

All the processes must see with the same order those *write* potentially in causal relation. Concurrent *write* can be seen in any order by different machines.

Figure 98: Example of Causal consistency.

The first one is an example of causal consistency and **not** sequential consistency (neither strict). In the second image, by deleting $R(x)a$ in $P2$ it becomes causal consistent.
Read and write operations that are causally related are seen by every node of the distributed system in the same order. Concurrent writes may be seen in different order in different nodes. For instance, If there is an operation or event A that causes another operation B, then causal consistency provides an assurance that each other process of the system observes operation A before observing operation B. Therefore, causally related operations and concurrent operations are distinguishable in the causal consistency model. In more detail, if one write operation influences another write operation, then these two operations are causally related, one relies on the other. Concurrent operations are the ones that are unrelated by causality or causally independent. In particular, concurrent writes are independent operations, no one causes or influences the other.

On the previous image we have two examples: the left one preserve causal consistency, instead the right one. Write operations that are related by potential causality are seen by each process of the system in common order. Also, concurrent writes can occur in any order and can be seen in different orders by the processes in the system.

15.2.1 FIFO consistency

Process *write* are seen by all the other processes in the execution order (FIFO). Write of different processes can be seen in any order by different machines. The following example is not valid for Causal consistency because W(x)1 and W(x)2 are causal, so different processes must read it in the same sequence.

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

Figure 99: Example of Fifo consistency.

15.2.2 Weak Consistency

It is needed in order to synchronize variables and it works in the following way:

1. Access to synchronized variables sequentially consistent.
2. Operations on synchronized variables are not allowed until all the previous *write* are completed everywhere.
3. Data *read* or *write* operations are not allowed until all the previous synchronization operations are completed.

Figure 100: Example of Weak and not Weak consistency.

15.2.3 Relaxed Consistency

Relaxed consistency has been introduced since a data store can't recognize synchronized requests for *write* or for *read* (in this second case an useless overhead will be introduced).
In order to solve this, different synchronization operations will be used:

- **Acquire:** to enter a critical section.
- **Release:** just out of the critical section.

The procedure will work as follows:

1. Before executing a *read* or a *write* all the acquisitions of a process must be completed successfully.
2. Before executing a *release* all the previous *read* or *write* done by the process must be completed.
3. The access to the synchronized variables are with FIFO policy.

Figure 101: Example of Relaxed consistency.

Updates can be done in two different phases:

- At release time: **Eager**
- At acquire time: **Lazy**

Another approach could be the one of dividing the program in phases instead of critical section and before entering the phase $n + 1$, all the processes must have completed phase n.

15.2.4 Entry consistency

In this model, every shared data is associated to a synchronized variable and we access it with an acquire-release. In this way, there's an increase in the access time but also an increase in the parallelism by allowing multiple accesses to critical section.
The access to a synchronized variable could be **exclusive** or **not exclusive**.

1. An access acquisition to a synchronized variable is not admitted for a process until all the update of the guarded shared data have been executed by the process.
2. Before admitting an exclusive access to a synchronized variable for a process, no other process can have a synchronized variable neither exclusive.
3. After an exclusive access to a synchronized variable every other not exclusive access to the synchronized variable must check with the process owner of the variable the most recent information.

Figure 102: Example of Entry consistency.

At *acquire* time all the data are made visible. After P makes an *acquire* all the other processes must pass the check of P.

15.2.5 Comparison of consistency models

Among the different consistency models, it is possible to consider the following comparisons.

Consistency	Description
Strict	Total absolute ordering of all the elements shared for access.
Sequential	All the processes must see the shared accesses in the same order. The accesses are ordered also according to a global timestamp (not unique).
Linearizable	All the processes see all the shared accesses in the same order. The accesses are not time ordered.
Causal	All the processes see the shared accesses in causal relation in the same order.
FIFO	All the processes see *writes* between them in the order as they used them. *Writes* of different processes are not always seen in the same order.

Table 5: Consistency models that don't use synchronized operations.

Consistency	Description
Weak	Shared data can be considered consistent only after the synchronization.
Relaxed	Shared data can be considered consistent only when the process exit the critical section.
Entry	Shared data of a critical section can be considered consistent only when the process enters a critical section.

Table 6: Consistency models with synchronized operations.

15.2.6 Eventual consistency (final)

The previous models were centered on data, this means that we search to have a consistent view of data for simultaneous accesses. *Eventual consistency* is instead **client centered**, it guarantees the correct access to the data store from the viewpoint of each client.
Eventual consistency guarantee to propagate the update to all the copies at the end, it is efficient if the clients always work on the same replica.
There are different models that describe the access:

- **Monotone reads:** if the process reads x ,every successive read of that process returns the same value or a more recent one.
- **Monotone writes:** a *write* of a process on x is completed before any other *write* on x of the same process.
- **Reads your write:** the effects of a *write* on x by a process are always visible by successive *read* on x by the same process.
- **Writes follow reads:** guarantees that a *write* on x by a process successive to a *read* on x by the same process gives the same results read or a more recent one.

15.3 Replica positioning

The data in our system consist of a collection of items that we shall call objects. An *object* could be a file, but each such logical object is implemented by a collection of physical copies called **replicas**. The replicas are physical objects, each stored at a single computer, with data and behavior that are tied to some degree of consistency by the system's operation. The replicas of

a given object are not necessarily identical, at least not at any particular point in time. Some replicas may have received updates that others have not received.
Replicas can be stored in memory in different ways:

- **Permanent:** they are static copies, mainly this is done for an historical archive. Ex: backup.
- **Server-initiated:** They are created dynamically and they are based on the knowledge of the topology.
- **Client-initiated:** They are based on the client cache.

Figure 103: Model of replica positioning.

Cache is an example which data is partial, incomplete and not necessary consistent.

15.3.1 System model

The model involves replicas held by distinct **replica managers**, which are software components that contain the replicas on a given computer and perform operations upon them directly. This general model may be applied in a client-server environment, in which case a *replica manager* is a server.
We shall always require that a replica manager applies operations to its replicas recoverably, in such a way that these operation can be recovered. This allows us to assume that an operation at a replica manager doesn't leave inconsistent results if it partially fails. Replica manager manages also possible faults and guarantee coherent behavior.

Figure 104: A basic architectural model for the management of replicated data.

In the previous image it is possible to see the general model of replica management.
A collection of replica managers provides a service to clients. The clients see a service that gives them access to objects, which in fact are replicated at the managers. Each client requests a series of operations, an operation may involve a combination of reads of objects and updates to objects. Requested operations that involve no updates are called *read-only* requests, requested operations that update an object are called *update* requests (these may also involve reads).
Each client's requests are first handled by a component called a **front end**. The role of the *front end* is to communicate by message passing with one or more of the replica managers, rather than forcing the client to do this itself explicitly. Front end introduces replication, location and access transparency. It is the vehicle for making replication transparent. A *front end* may be implemented in the client's address space, or it may be a separate process.
In general, it is possible to highlight five steps during the execution of a single request upon replicated objects:

1. **Request:** The *front end* issues the request to one or more replica managers in two possible ways:
   - Either the front end communicates with a single replica manager, which in turn communicates with other replica managers.
   - Or the front end multicasts the request to the replica managers.
2. **Coordination:** replica managers coordinate for executing the request consistently, they agree on whether the request is to be applied. They also decide the ordering of the request relative to others. The types of ordering defined are:
   - *FIFO ordering:* if a *front end* issues request r and then request r', any correct replica manager that handles r' handles r before it.
   - *Causal ordering:* if the issue of request r happened-before the issue of request r', then any correct replica manager that handles r' handles r before it.
   - *Total ordering:* if a correct replica manager handles r before request r', then any correct replica manager that handles r' handles r before it.
3. **Execution:** The replica managers execute the request, perhaps *tentatively*, in order to undo its effects later. Meaning that it can work on tentative version and update at the end.
4. **Agreement:** The replica managers reach consensus on the effect of the request, if they agree to the commit choice, that will be committed.
5. **Response:** One or more replica managers responds to the *front end*. In some systems, one replica manager sends the response. In others, the front end receives responses from a collection of replica managers and selects or synthesizes a single response to pass back to the client.

16 The role of group communication

The membership of groups can be created in different ways, in replication circumstances, there's a strong requirement for dynamic membership, in which processes join and leave the group as the systems executes. A group membership service implements several features:

- Provides an interface to manage group changes, group creation/delete.
- Implement a fault detector that marks the suspected processes.
- Notifies the group changes to the group members.
- Makes the group address expansion.

A group membership service maintains **group views**, which are lists of the current group members, identified by their unique process identifiers.
For each group g the group service delivers to any member process $p \in g$ a series of views. In general, a member *delivering a view*, when a membership change occurs and the application is notified of the new membership. Group decided a priori and can evolve during the time. Some basic requirements for **view delivery** are as follows:

- *Order:* if a process p delivers view $v(g)$ and then $v'(g)$, then no other process $q \neq p$ delivers $v'(g)$ before $v(g)$.
- *Integrity:* if a process p delivers view $v(g)$, then $p \in v(g)$.
- *Non-triviality:* if process q joins a group and is or becomes indefinitely reachable from process $p \neq q$, then eventually q is always in the views that p delivers.

Figure 105: Services for a group of processes.

A **view-synchronous** group communication system makes additional guarantees to those above about the delivery ordering of view notifications with respect to the delivery of multicast messages.
The guarantees provided by view-synchronous group communication are as follows:

- *Agreement:* correct processes deliver the same sequence of views and the same set of messages in any given view. If a correct process delivers message m in view $v(g)$, then all other correct processes that deliver m also do so in the view $v(g)$.

- *Integrity:* if a correct process p delivers message m, then it will not deliver m again. Furthermore, $p \in group(m)$ and the process that sent m is in the view in which p delivers m.
- *Validity:* correct processes always deliver the messages that they send. If a process q has not delivered a message, system notifies and successive views exclude q. Let p be any correct process that delivers message m in view $v(g)$. If some process $q \in v(g)$ does not deliver m in view $v(g)$, then the next view $v'(g)$ that p delivers has $q \notin v'(g)$.

Considering the following image, a group with three processes, p, q and r. Suppose that p sends a message m while in view (p, q, r) but that p crashes soon after sending m, while q and r are correct. One possibility is that p crashes before m has reached any other process. In this case, q and r each deliver the new view (q, r), but neither ever delivers m (**a**).
The other possibility is that m has reached at least one of the two surviving processes when p crashes. Then q and r both deliver first m and then the view (q, r) (**b**). It is not allowed for q and r to deliver first the view (q, r) and then m (**c**), since then they would deliver a message from a process that they have been informed has failed, nor can the two deliver the message and the new view in opposite orders (**d**).

Figure 106: Delivery of consistent messages.

16.1 Fault-tolearnt services

In the following pages we will examine how to provide a service that is correct, despite process failures, by replicating data and functionality at replica managers. Different models of replication for fault tolerance will be presented below.

16.1.1 Passive (primary-backup) replication

In the **passive** or **primary-backup** model of replication for fault tolerance, there is at any one time a single *primary* replica manager and one or more *secondary* replica managers (*backups* or

slaves). In the pure form of the model, *front ends* communicate only with the primary replica manager to obtain the service. The primary replica manager executes the operations and sends copies of the updated data to the backups. If the primary fails, one of the backups is promoted to act as the primary.
The sequence of events when a client requests an operation to be performed is as follows:

1. *Request:* the front end issues the request, containing a unique identifier, to the primary replica manager.
2. *Coordination:* the primary takes each request atomically, in the order in which it receives it. It checks the unique identifier, in case it has already executed the request, and if so it simply resends the response.
3. *Execution:* the primary executes the request and stores the response.
4. *Agreement:* if the request is an update, then the primary sends the updated state, the response and the unique identifier to all the backups. The backups send an acknowledgement.
5. *Response:* the primary responds to the front end, which hands the response back to the client.

Figure 107: The passive (primary-backup) model for fault tolerance.

This system obviously implements linearizability if the primary is correct, since the primary sequences all the operations upon the shared objects. If the primary fails, then the system retains linearizability if a single backup becomes the new primary and if the new system configuration takes over exactly where the last left off. In other words, if the primary fails another backup is used to substitute it. That is if:

- The primary is replaced by a unique backup (if two clients began using two backups, then the system could perform incorrectly).
- The replica managers that survive agree on which operations had been performed at the point when the replacement primary takes over.

To survive up to f process crashes, a passive replication system requires $f + 1$ replica managers.

16.2 Active replication

In the *active* model of replication for fault tolerance, the replica managers are state machines that play equivalent roles and are organized as a group. Front ends multicast their requests to

the group of replica managers and all the replica managers process the request independently but identically and reply. If any replica manager crashes, this need have no impact upon the performance of the service, since the remaining replica managers continue to respond in the normal way. Under active replication, the sequence of events when a client requests an operation to be performed is as follows:

1. *Request:* the front end attaches a unique identifier to the request and multicasts it to the group of replica managers, using a totally ordered, reliable multicast primitive. The front end is assumed to fail by crashing at worst. It does not issue the next request until it has received a response.
2. *Coordination:* the group communication system delivers the request to every correct replica manager in the same (total) order.
3. *Execution:* every replica manager executes the request. Since they are state machines and since requests are delivered in the same total order, correct replica managers all process the request identically. The response contains the client's unique request identifier.
4. *Agreement:* no agreement phase is needed, because of the multicast delivery semantics.
5. *Response:* each replica manager sends its response to the front end. The number of replies that the front end collects depends upon the failure assumptions and the multicast algorithm. If, for example, the goal is to tolerate only crash failures and the multicast satisfies uniform agreement and ordering properties, then the front end passes the first response to arrive back to the client and discards the rest (it can distinguish these from responses to other requests by examining the identifier in the response).

Figure 108: The active replication model for fault tolerance.

16.3 High available services

In the following pages, we consider how to apply replication techniques to make services highly available. Our attention now is on giving clients access to the service for as much of the time as possible. We now examine the design of a system that provide highly available services: the **Gossip architecture**.

16.3.1 The gossip architecture

Gossip architecture was developed as a framework for implementing highly available services by replicating data close to the points where groups of clients need it. The name reflects the fact that the replica managers exchange *"gossip"* messages periodically in order to convey the updates they have each received from clients.

Figure 109: Query and update operations in a gossip service.

A gossip service provides two basic types of operation: *queries* are read-only operations and *updates* modify but do not read the state. A key feature is that front ends send queries and updates to any replica manager they choose, provided it is available and can provide reasonable response times. The system makes two guarantees, even though replica managers may be temporarily unable to communicate with one another:

- *Each client obtains a consistent service over time:* in answer to a query, replica managers only ever provide a client with data that reflects at least the updates that the client has observed so far.
- *Relaxed consistency between replicas:* all replica managers eventually receive all updates and they apply updates with ordering guarantees that make the replicas sufficiently similar to suit the needs of the application.

The front end's version timestamp. In order to control the ordering of operation processing, each front end keeps a vector timestamp that reflects the version of the latest data values accessed by the front end. This timestamp, denoted *prev* in the above image, contains an entry for every replica manager. The front end sends it in every request message to a replica manager, together with a description of the query or update operation itself. When a replica manager returns a value as a result of a **query** operation, it supplies a new vector timestamp (*new* in the above image), since the replicas may have been updated since the last operation. Similarly, an **update** operation returns a vector timestamp (*Update ID* in the above image) that is unique to the update. Each returned timestamp is merged with the front end's previous timestamp to record the version of the replicated data that has been observed by the client.

Figure 110: Front ends propagate their timestamps whenever clients communicate directly

Replica manager state. Regardless of the application, a replica manager contains the following main state components:

- *Value:* this is the value of the application state as maintained by the replica manager. Each replica manager is a state machine, which begins with a specified initial value and is thereafter solely the result of applying update operations to that state.
- *Value timestamp:* this is the vector timestamp that represents the updates that are reflected in the value. It contains one entry for every replica manager. It is updated whenever an update operation is applied to the value.
- *Update log:* all update operations are recorded in this log as soon as they are received.
- *Replica timestamp:* this vector timestamp represents those updates that have been accepted by the replica manager. It differs from the value timestamp in general, of course, because not all updates in the log are stable.
- *Executed operation table:* to prevent an update being applied twice, the *"executed operation"* table is kept, containing the unique front-end-supplied identifiers of updates that have been applied to the value. The replica managers check this table before adding an update to the log.
- *Timestamp table:* this table contains a vector timestamp for each other replica manager, filled with timestamps that arrive from them in gossip messages. Replica managers use the table to establish when an update has been applied at all replica managers.

Figure 111: A gossip replica manager, showing its main state components.

16.4 Transactions with replicated data

An important aspects to deal with in replicated data is surely the *transactions*. A transaction on replicated objects should appear the same as one with non-replicated objects. The effect of transactions performed by clients on replicated objects should be the same as if they had been performed one at a time on a single set of objects. This property is called **one-copy serializability**. It is similar to, but not to be confused with, sequential consistency. Sequential consistency considers valid executions without any notion of aggregating the client operations into transactions.
The implementation of *one-copy serializability* is illustrated by *read-one/write-all*, a simple replication scheme in which *read* operations are performed by a single replica manager and *write* operations are performed by all of them.

16.4.1 Architectures for replicated transactions

A front end may either multicast client requests to groups of replica managers or send each request to a single replica manager, which is then responsible for processing the request and responding to the client. The replica manager that receives a request to perform an operation on a particular object is responsible for getting the cooperation of the other replica managers in the group that have copies of that object. Different replication schemes have different rules as to how many of the replica managers in a group are required for the successful completion of an operation. For example, in the *read-one/write-all* scheme, a *read* request can be performed by a single replica manager, whereas a *write* request must be performed by all the replica managers in the group.

Figure 112: Transactions on replicated data.

16.4.2 Available copies replication

Simple *read-one/write-all* replication is not a realistic scheme, because it cannot be carried out if some of the replica managers are unavailable, either because they have crashed or because of a communication failure. The **available copies** scheme is designed to allow for some replica managers being temporarily unavailable.
Normally, client requests are received and performed by a functioning replica manager. *read* requests can be performed by the replica manager that receives them. *write* requests are performed by the receiving replica manager and all the other available replica managers in the group.
For example, in the below image, the *getBalance* operation of transaction T is performed by X, whereas its *deposit* operation is performed by M, N and P. Concurrency control at each replica manager affects the operations performed locally. For example, at X, transaction T has read A and therefore transaction U is not allowed to update A with the *deposit* operation until transaction T has completed.

Figure 113: Available copies.

16.4.3 Network partitions

Replication schemes need to take into account the possibility of network partitions. A network partition separates a group of replica managers into two or more subgroups in such a way that the members of one subgroup can communicate with one another but members of different subgroups cannot communicate with one another. For example, in the following image, the replica managers receiving the *deposit* request cannot send it to the replica managers receiving the *withdraw* request.

Replication schemes are designed with the assumption that partitions will eventually be repaired. Therefore, the replica managers within a single partition must ensure that any requests that they execute during a partition will not make the set of replicas inconsistent when the partition is repaired.

Figure 114: Network partition.

Many approaches were discussed, which they categorize as being either optimistic or pessimistic with regard to whether inconsistencies are likely to occur. The optimistic schemes do not limit availability during a partition, whereas pessimistic schemes do.

- The **optimistic approach** allows updates in all partitions, this can lead to inconsistencies between partitions, which must be solved when partition is repaired.
- The **pessimistic approach** limits availability even when there are no partitions, but it prevents any inconsistencies occurring during partitions. When a partition is repaired, all that needs to be done is to update the copies of the objects. In other words it is based on the idea of **quorum**, only the partition that satisfy the quorum is available to the client. The problem here is that it can happen that does not exist a partition with quorum, and so no requests can be executed.
- The **Combined approach** is based on algorithm of virtual partition.

16.4.4 Virtual partition

A **virtual partition** is an abstraction of a real partition and contains a set of replica managers. Note that the term *"network partition"* refers to the barrier that divides replica managers into several parts, whereas the term *"virtual partition"* refers to the parts themselves.

Figure 115: Virtual partition.

16.4.5 Creating a virtual partition.

Phase 1:

- The initiator sends a *Join* request to each potential member. The argument of *Join* is a proposed logical timestamp for the new virtual partition.
- When a replica manager receives a *Join* request, it compares the proposed logical timestamp with that of its current virtual partition.
    - If the proposed logical timestamp is greater it agrees to join and replies *Yes*;
    - If it is less, it refuses to join and replies *No*.

Phase 2:

- If the initiator has received sufficient *Yes* replies to have *read* and *write* quora, it may complete the creation of the new virtual partition by sending a *Confirmation* message to the sites that agreed to join. The creation timestamp and list of actual members are sent as arguments.
- Replica managers receiving the *Confirmation* message join the new virtual partition and record its creation timestamp and list of actual members.

In other words: when a replica manager receives a request it tries to create a virtual partition as large as possible in order to reach the quorum. Replica managers can communicate using an alternative way. This strategy improves the availability since the virtual partition can be greater than the real one on the network.

17 New paradigms

In the following section we are going to analyze three new paradigms for computing that are: **mobility computing**, **ubiquitous computing** and **peer-to-peer computing**.

17.1 Mobility computing

Mobile computing is concerned with exploiting the connectedness of devices that move around in the everyday physical world; ubiquitous computing is about exploiting the increasing integration of computing devices with our everyday physical world. As devices become smaller, we are better able to carry them around with us or wear them, and we can embed them into many parts of the physical world – not just the familiar desktop or a server rack. And as wireless connectivity becomes more prevalent, we are better able to connect these new small devices to one another, and to conventional personal and server computers. Mobile computing is a paradigm that allows users to move their personal computers maintaining some connectivity to other machines.
The idea of mobility consists on moving the knowledge close to resources in order to improve the performance of the communication system, and it allows adapting the access of the clients to remote resources improving so the flexibility property of the system. An simple example is Amazon CloudFront. There are two possible levels of mobility:

- **Code mobility**, it is the capability to dynamically relocate at run time the distributed application components. It can lead to the relocation of only the code or also the state.
- **Mobile agent**, program in execution can migrate among machines in an heterogeneous networks. Migration time and the new location depend on interaction between agent and the environment.

Migration. We have seen that with mobile agent there is a sort of migration, and so we can define **migration of processes** or **migration of objects**. The main goal is to implement a **load balancing**, which a system that distributes load uniformly between machines on the network. It is very useful for small networks and it provides location transparency to the client.

Mobile code. Mobile code defines different paradigms, that are different in terms of model independence, interaction schemes for coordination and component relocation to realize the service. The service is realized if one knows the services, needed resources or executing item. Components are the resources used, interaction is defined with request-reply protocol. There are essentially four paradigms:

- **client-server**, common architecture that we have already seen, in which there is a client entity that ask for a resource and the server reply with an answer. Code is executed inside the server.
- **remote evaluation**, involves the transmission of executable software code from a client computer to a server computer for subsequent execution at the server. After the code has finished executing, the results of its execution are sent back to the client.
- **code on demand**, sends executable software code from a server computer to a client computer upon request from the client's software. Some well-known examples of the code on demand paradigm on the web are Java applets.
- **mobile agent**, is a composition of computer software and data which is able to migrate (move) from one computer to another autonomously and continue its execution on the destination computer. In reality, the mobile agent is the code/object on move which travels

in its itinerary within the network of connected nodes. More specifically, a mobile agent is a process that can transport its state from one environment to another, with its data intact, and be capable of performing appropriately in the new environment. Mobile agents decide when and where to move.

Mobile code paradigm brings some issues that must be considered, like: security, detect component to move, why, how and when move it, communication, data space management.
Advantages of mobility code are: maintainability, flexibility of data management and in protocol inclusion, reliability and autonomy. Common applications are: IR, E-Commerce.

17.1.1 Types of code mobility

There are essentially two types of code mobility: **strong mobility** and **weak mobility**.

Strong mobility. involves moving both the code, data and the execution state from one host to another, this is important in cases where the running application needs to maintain its state as it migrates from host to host. Execution is suspended, transmitted to the new environment and there started again.

Weak mobility. Involves moving the code and the data only. Therefore, it may be necessary to restart the execution of the program at the destination host.

17.1.2 Data space management

We can simply say that mobility is a modification of the data space. Management depends on the resource type and by the references. We can have transferable resource or non transferable, and we can ave multiple links with the same resource. Every resource has one identifier, type and value, and they can be used for binding them. **Binding by type** guarantees compatibility, with **binding by values** the value does not change after the migration, instead with **binding by identifier** the name identifies the resource. When we have different strategies to manage the data space:

- **Binding Removal**, remove the binging to resources (invalidation). If a process try to access to invalid resource an exception can occurs.
- **Network Reference**, maintain the reference but make it remote. Problem: how can we distinguish local binding to remote one?
- **Move**, moves directly the resource and it remains local. Move code and data.
- **Copy**, duplicate the resources and code.
- **Re-binding**, instead of applying a copy try to use an object of the same type. Different object but with different type.

17.1.3 Mobility design

Security is a fundamental point for mobility, since communication is not sure and there can be applied different types of attacks like: access to private resources (authentication), spoofing, incorrect resource using, denial of service or block of machines. The main goals of the design of mobile code are:

- **security**, protection by accidental damages or intentional ones of the execution environment.

- **portability**, heterogeneity management of the platforms.
- **performance**, provide an high level of performance.

Code can be interpreted or compiled, both the two strategies bring some advantages and limits. Interpretation increases portability and security run time control, but decreases performance. Compilation improves performance but decreases portability and security. As we know there are also other strategies, called hybrid (like java), that try to get the best of both the two strategies. In general with hybrid solution the source code is compiled in an intermediate language, lower respect to the original one, then it is interpreted.

17.2 Ubiquitous systems

Ubiquitous computing refers to a system in which computing is made to appear anytime and everywhere. In contrast to desktop computing, ubiquitous computing can occur using any device, in any location, and in any format. A user interacts with the computer, which can exist in many different forms, including laptop computers, tablets and terminals in everyday objects such as a refrigerator or a pair of glasses.

18 Operating System Support

In the present section, we will examine the relationship between the middleware layer and the operating system (OS) layer. The task of any operating system is to provide problem-oriented abstractions of the underlying physical resources, it takes over the physical resources on a single node and manages them to present these resource abstractions through the system-call interface.

18.1 Operating system layer

Users will only be satisfied if their middleware-OS combination has good performance. Middleware runs on a variety of OS-hardware combinations (platforms) at the nodes of a distributed system. The OS running at a node provides its own flavor of abstractions of local hardware resources for processing, storage and communication. Middleware utilizes a combination of these local resources to implement its mechanisms for remote invocations between objects or processes at the nodes.
The following image shows how the operating system layer at each of two nodes supports a common middleware layer in providing a distributed infrastructure for applications and services.

Figure 116: System layers.

The below image instead shows the core OS functionality that we shall be concerned with: process and thread management, memory management and communication between processes on the same computer.

Figure 117: Core Os Functionality.

The core OS components and their responsibilities are:

- *Process manager:* creation of operations upon processes. A process is a unit of resource management, including an address space and one or more threads.
- *Thread manager:* thread creation, synchronization and scheduling. Threads are scheduled activities attached to processes.
- *Communication manager:* communication between threads attached to different processes on the same computer.
- *Memory manager:* management of physical and virtual memory.
- *Supervisor:* dispatching of interrupts, system call traps and other exceptions, control of memory management unit and hardware caches.

18.2 Processes and threads

A process consists of an execution environment together with one or more threads. A *thread* is the operating system abstraction of an activity. An **execution environment** is the unit of resource management: a collection of local kernel-managed resources to which its threads have access. An execution environment primarily consists of:

- an address space.
- thread synchronization and communication resources such as semaphores and communication interfaces (sockets).
- higher-level resources such as open files and windows.

Execution environments are normally expensive to create and manage, but several threads can share them. Threads can be created and destroyed dynamically and the goal of having multiple threads of execution is to maximize the degree of concurrent execution between operations, thus enabling the overlap of computation with input and output, and enabling concurrent processing on multiprocessors.

18.2.1 Address space

An address space is a unit of management of a process's virtual memory. It is large and consists of one or more *regions*, separated by inaccessible areas of virtual memory. A region, that can be seen in the below image, is an area of contiguous virtual memory that is accessible by the threads of the owning process.

Figure 118: Address space

Each region is specified by the following properties:

- its dimension (lowest virtual address and size)
- read/write/execute permissions for the process's threads
- whether it can be grown upwards or downwards

A **shared memory region** is one that is backed by the same physical memory as one or more regions belonging to other address spaces. Processes therefore access identical memory contents in the regions that are shared, while their non-shared regions remain protected. The uses of shared regions include the following:

- *Libraries:* library code can be very large and would waste considerable memory if it was loaded separately into every process that used it.
- *Kernel:* often the kernel code and data are mapped into every address space at the same location. When a process makes a system call or an exception occurs, there is no need to switch to a new set of address mappings.
- *Data sharing and communication:* two processes, or a process and the kernel, might need to share data in order to cooperate on some task.

18.2.2 Creation of a new process

For a distributed system, the design of the process-creation mechanism has to take into account the utilization of multiple computers, consequently, the process-support infrastructure is divided into separate system services.
The creation of a new process can be separated into two independent aspects:

- the **choice of a target host**, for example, the host may be chosen from among the nodes in a cluster of computers acting as a compute server.
- the **creation of an execution environment**.

Choice of process host. The choice of the node at which the new process will reside is a matter of policy.
The *transfer policy* determines whether to situate a new process locally or remotely. The *location policy* determines which node should host a new process selected for transfer. Process location policies may be *static* or *adaptive*.

Creation of a new execution environment. Once the host computer has been selected, a new process requires an execution environment consisting of an address space with initialized contents.
There are two approaches to defining and initializing the address space of a newly created process.

1. The first approach is used where the address space is of a statically defined format. In this case, the address space regions are created from a list specifying their extent. Address space regions are initialized from an executable file or filled with zeros as appropriate.
2. In the second approach, the address space can be defined with respect to an existing execution environment (*copy-on-write*).

Figure 119: Copy-on-write

18.2.3 Threads

Another important aspect is to talk about the advantages of enabling client and server processes to possess more than one thread. They allow concurrency with I/O operation and computation in multiprocessor systems. Possible advantages of the multi-thread are:

- Less expensive for creation and management.
- Simpler to make resource sharing.

Possible problems can be, instead, the mechanisms for concurrent programming.

Figure 120: Client and server with threads

The previous image shows one of the possible threading architectures, the **worked pool architecture**. In its simplest form, the server creates a fixed pool of *"worker"* threads to process the requests when it starts up. The module marked *"receipt and queuing"* is typically implemented by an *"I/O"* thread, which receives requests from a collection of sockets or ports and places them on a shared request queue for retrieval by the workers. We may handle varying *request priorities* by introducing multiple queues into the worker pool architecture, so that the worker threads scan the queues in the order of decreasing priority.
A *disadvantage* of this architecture is its *inflexibility*, the number of worker threads in the pool may be too few to deal adequately with the current rate of request arrival. Another disadvantage is the high level of switching between the I/O and worker threads as they manipulate the shared queue.
Other types of architectures are:

- **thread-per-request architecture:** the I/O thread spawns a new worker thread for each request, and that worker destroys itself when it has processed the request against its designated remote object. This architecture has the advantage that the threads do not contend for a shared queue, and throughput is potentially maximized because the I/O thread can create as many workers as there are outstanding requests. Its disadvantage is the overhead of the thread creation and destruction operations.
- **thread-per-connection architecture:** associates a thread with each connection. The server creates a new worker thread when a client makes a connection and destroys the thread when the client closes the connection. In between, the client may make many requests over the connection, targeted at one or more remote objects.
- **thread-per-object architecture:** associates a thread with each remote object. An I/O thread receives requests and queues them for the workers, but this time there is a per-object queue.

The representation of these different architectures can be seen in the following image.

Figure 121: Alternative server threading architectures.

The thread's lifecycle can be seen in the below image.

Figure 122: Thread's lifecycle.

18.3 Scheduler and Thread

Many kernels provide native support for multi-threaded processes, including Windows, Linux, Solaris, Mach and Mac OS X. These kernels provide thread-creation and -management system calls, and they schedule individual threads. Some other kernels have only a single-threaded process abstraction. Multi- threaded processes must then be implemented in a library of procedures linked to application programs. We can say that threads are implemented at application level. In such cases, the kernel has no knowledge of these user-level threads and therefore cannot schedule them independently. A threads runtime library organizes the scheduling of threads. A thread would block the process, and therefore all threads within it, if it made a blocking system call, so the asynchronous (non-blocking) I/O facilities of the underlying kernel are exploited.
Using thread at application level we have some advantages like:

- less cost to context change (without system call).
- possible to implement scheduling ad hoc (adaptive).
- possible to create and manage larger number of possible threads.

The drawbacks are:

- kernel cannot make thread scheduling
- kernel cannot compare priorities of thread in different processes.

- thread cannot use multiprocessor concurrency.
- a thread error blocks the process and all its threads.

It is possible to combine the advantages of user-level and kernel-level threads implementations. One approach is to enable user-level code to provide scheduling hints to the kernel's thread scheduler it is called **hybrid solution**. Another is a form of **hierarchical scheduling**. Each process creates one or more kernel-level threads. User-level threads are also supported. A user-level scheduler assigns each user-level thread to a kernel-level thread. This scheme can take advantage of multiprocessors, and also benefits because some thread-creation and thread-switching operations take place at user level. The scheme's disadvantage is that it still lacks flexibility: if a thread blocks in the kernel, then all user-level threads assigned to it are also prevented from running, regardless of whether they are eligible to run.

18.4 Communication: invocation and call

Communication is part of invocation and it can be implemented using different approaches like: RPC, RMI, message passing, event notification, system call or group notification. It is important also to consider the transparency level given by this techniques, and a good proxy for providing that is a middleware. An important aspect of communication is the performance provided by the mechanism used. Fundamental aspects that can affect the performance are:

- type of communication that can be synchronous or asynchronous.
- address space to cross, it can be local or remote. Obviously crossing remote address space introduce some possible network delays that can have strong impact.
- scheduling thread that affect context change.

Measures for the evaluation of the performance are *delay time* and the *throughput*, that can be affected by latency.

Figure 123: Typology of address space

Lightweight Remote procedure call The previous image suggests that a cross-address-space invocation is implemented within a computer exactly as it is between computers, except that the underlying message passing happens to be local. Indeed Bershad developed a more efficient invocation mechanism for the case of two processes on the same machine called **lightweight RPC**. The LRPC design is based on optimizations concerning data copying and thread scheduling. Instead of RPC parameters being copied between the kernel and user address spaces involved, the client and server are able to pass arguments and return values directly via an A stack. The same stack is used by the client and server stubs. In LRPC, arguments are copied once: when they are marshalled onto the A stack. There may be several A stacks in a shared region, because several threads in the same client may call the server at the same time.

Figure 124: lightweight RPC

A client thread enters the server's execution environment by first trapping to the kernel and presenting it with a capability. The kernel checks this and only allows a context switch to a valid server procedure; if it is valid, the kernel switches the thread's context to call the procedure in the server's execution environment. When the procedure in the server returns, the thread returns to the kernel, which switches the thread back to the client execution environment. Note that clients and servers employ stub procedures to hide the details just described from application writers. There is little doubt that LRPC is more efficient than RPC for the local case, as long as enough invocations take place to offset the memory management costs.

Asynchronous operations: serialized and concurrent invocation With remote invocation we have some delays that have a strong impact on the final performance. The main goal of asynchronous operation is to try to limit as much as possible them. A solution is performing **concurrent invocation**, respect to **serialized one**. The following picture can give a simple idea of how they work:

Figure 125: Serial and Concurrent invocation

With concurrent invocation client allocate multiple threads to perform blocking invocations concurrently. A clear example could be the browser web. We have seen the potential benefits of interleaving invocations between a client and a single server on a single-processor machine. In the serialized case, the client marshals the arguments, calls the `Send` operation and then waits until the reply from the server arrives – whereupon it `Receives`, unmarshals and then processes the results. After this it can make the second invocation. In the concurrent case, the first client thread marshals the arguments and calls the `Send` operation. The second thread then immediately makes the second invocation. Each thread waits to receive its results.

18.5 Organization and distributed operating system architecture

In this section, we examine the architecture of a kernel suitable for a distributed system. An open distributed system should make it possible to:

- run only that system software at each computer that is necessary for it to carry out its particular role in the system architecture – system software requirements can vary between, for example, mobile phones and server computers, and loading redundant modules wastes memory resources;
- allow the software (and the computer) implementing any particular service to be changed independently of other facilities;
- allow for alternatives of the same service to be provided, when this is required to suit different users or applications;
- introduce new services without harming the integrity of existing ones.

The separation of fixed resource management mechanisms from resource management policies, which vary from application to application and service to service, is the main principle in operating system design.
The organization of the operating system can be: **monolithic** or **microkernel**. These designs differ primarily in the decision as to what functionality belongs in the kernel and what is to be left to server processes that can be dynamically loaded to run on top of it.

Monolithic it performs all basic operating system functions and takes up in the order of megabytes of code and data, it is coded in a non-modular way. In other words it is an operating system architecture where the entire set of functionalities is working in kernel space. The result is that to a large extent it is intractable: altering any individual software component to adapt it to changing requirements is difficult. The advantage is that it is relatively efficient.

Microkernel the kernel provides only the most basic abstractions, principally address spaces, threads and local interprocess communication; all other system services are provided by servers that are dynamically loaded at precisely those computers in the distributed system that require them. Clients access these system services using the kernel's message-based invocation mechanism. The advantages are extensibility of the system, given by its modular organization, and it is simple to debug.

18.6 Scheduling: process allocation

Allocation of processes in centralized system has the advantage of using a unique process queue, in some sense we have the global knowledge. So the scheduler manages that queue and applies scheduling algorithm to decide which is the next process that must be executed. **Scheduling algorithms** are designed for system performance optimization, that essentially try to minimize the response time and maximize the system resources utilization (throughput, CPU utilization). There can be different properties of this scheduling algorithms like: fairness, efficiency, fault-tolerance etc.
Under certain condition the theory provides an optimal algorithm for process scheduling. In **multiprocess systems** also we have a unique process queue, but now the system is composed by multiple processors. In that case caching can improve the performance, so a good scheduler should be consider also the past history for process assignment to the CPU already used.
In distributed system process allocation is more complicated since there is not a unique process queue, processes must be allocated to different machines and migration of them can occur. There is not guarantee about process type, they can be identical or heterogeneous. Scheduling in distributed system introduce a new target to optimize which is **load balancing**. Load balancing consists on developing a solution/strategy that move processes between machines in order to assign the same load to the different machines. The algorithm can use state information, that could be local or coming from sub-system, it can use migration or some heuristics.

Process allocation could be classified in this way:

- **static/dynamic**, if allocation decision are prefixed or not. Dynamic algorithms are based on the state information.
- **deterministic/non-deterministic**, if next process requests are known
- **centralized/distributed/hierarchical**
- **optimal/ approximated**
- **with/without migration**, if a running process can be moved.

18.7 Scheduling algorithms for distributed systems

We can have different scheduling algorithms:

- **probes algorithm**
- **deterministic algorithm**

- **centralized algorithm**
- **bidding algorithm**
- **hierarchical algorithm**
- **coscheduling**

Probes algorithm When the processor executing the process becomes overloaded it takes the decision to transfer the process, **probes** another processor to decide whether to transfer the process. If the contacted processor is **underloaded** the transfer takes place, otherwise it selects randomly another processor to repeat the attempt up to a maximum number of times. The problem here is to decide the threshold for accepting processes and the maximum number of repetition.

Deterministic algorithm Here we have n processes to distribute on k identical processors ($n > k$). There is an assumption: the **traffic** between each process couple is known. The algorithm distributes the load to minimize the total traffic among the processors. The drawbacks of this strategy is the knowledge of the traffic distribution, traffic information and deal with change of variation of it.

Centralized algorithm it is a centralized algorithm to distribute the load between workstations for load balancing. Each workstation has a **score** and if it sends to other processes (resource consuming) the score increase, if it servers others (it is running processes of other workstation) decreases the score. A processors in the free state receives the process generated by the workstation with the lowest score. The algorithm uses a table of processor utilization. A possible variation consists on the fact that the score can be increased/decreased by a constant for each unit of CPU time requested/offered.

Bidding algorithm It is an extension of the centralized algorithm. It is based on the metaphor of an economic system, where CPU time is a associated to a price. The price of the resources if fixed by the marker lows (scores). Each processor computes the **offered price** according to the demands and quality of service. The algorithm evaluates the most convenient choice for each process to be executed.

Hierarchical algorithm The hierarchical algorithm improves scalability of the scheduler. Processes are organized based on an **hierarchical organization** in which each K processors has a **supervisor** that keeps the information about the single K workload and the number of available processors. We can have different levels of organization, and it is more fault tolerance since in case of crash of the supervisor **substitution mechanisms** are used to substitute it. Every supervisor that receives a request of J processors: if it has $K < J$ processors then forwards the request to a higher level of the hierarchy, otherwise it partition the request according to the free CPU of the lower levels forwarding them to the lower levels to the processors.

Coscheduling An algorithm can consider the communication among processes to allocate strongly connected processes on the same node. It is convenient to keep together communicating processes, in order to reduce communication delays. A common representation used for this algorithm is based on the usage of a matrix in which rows represent time units and columns represent processors. The location (i, j) contains the id of the a process to be executed at time i on processor j. The algorithm at time i runs the assigned processes indicated in row i.

19 Cloud computing

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of **five essential characteristics**:

- On-demand self service.
- Measured service.
- Broad network access.
- Rapid elasticity.
- Resource pooling.

It is composed of **three service models**:

- Software as a service ($SAAS$)
- Platform as a service ($PAAS$)
- Infrastructure as a service ($IAAS$)

and finally it is composed by **four deployment models**:

- Public cloud
- Private cloud
- Hybrid cloud
- Community cloud

In the following pages, all these concepts will be explained in a deepl way. In a more concise definition, we can so say that Cloud computing is a specialized form of **distributed computing** that introduces utilization model for remotely provisioning scalable and measured resources.
One possible error that can be made is to confuse *cloud computing* and *data center* concepts. The first one in fact has the goal of providing services, instead the second one can be considered only as a collection of a large amount of data stores and its main purpose is to provide data.

19.1 Terminology

The actual technologies is the result of an evolution of several pre-existing structures considered to be the primary influences on cloud computing.

Cluster of workstation. A cluster is a group of independent IT resources that are interconnected and work as a single system. A general prerequisite of hardware clustering is that its component systems have reasonably identical hardware and operating systems to provide similar performance levels. Component devices that form a cluster are kept in synchronization through dedicated, high-speed communication links.

Grid computing. A computing grid provides a platform in which computing resources are organized into one or more logical pools. Grid computing differs from clustering in that grid systems are much more loosely coupled and distributed. As a result, grid computing systems can involve computing resources that are heterogeneous and geographically dispersed, which is generally not possible with cluster computing-based systems.

Grid computing is based on a middleware layer that is deployed on computing resources. These IT resources participate in a grid pool that implements a series of workload distribution and coordination functions. This middle tier can contain load balancing logic, failover controls, and autonomic configuration management, each having previously inspired similar cloud computing technologies.

Virtualization. Virtualization represents a technology platform used for the creation of virtual instances of IT resources. A layer of virtualization software allows physical IT resources to provide multiple virtual images of themselves so that their underlying processing capabilities can be shared by multiple users. As cloud computing evolved, a generation of modern virtualization technologies emerged to overcome the performance, reliability, and scalability limitations of traditional virtualization platforms.

Enabling cloud computing technologies. Other areas of technology continue to contribute on the modern cloud-based platforms, these are distinguished as *cloud-enabling technologies*:

- Broadband Networks and Internet Architecture.
- Data Center Technology.
- Virtualization Technology.
- Web Technology.
- Service Technology.

19.2 Motivations

The motivations that lead to the introduction of cloud computing are basically:

- **Scalability needs**, there's the need of passing from a single PC to a data center because of the exponential growing of data and users. There's a massive data requirement by recent applications.
- **Computation needs**, there's the need of extend the computing power adding many servers because of the increase of web pages, images, users and queries on the net.

19.3 Definition

In this section, we will highlight the basic elements that represent the fundamental concepts belonging to the notion of cloud computing.

19.3.1 Cloud

A *cloud* refers to a distinct IT environment that is designed for the purpose of remotely provisioning scalable and measured IT resources. As a specific environment, a cloud has a finite *boundary* and there are many individual clouds that are accessible via the Internet. A cloud is typically privately owned and offers access to IT resources that is metered. IT resources provided by cloud environments are dedicated to supplying back-end processing capabilities and user-based access to these capabilities. A cloud can be based on the use of any protocols that allow for the remote access to its IT resources.

19.3.2 IT Resource

An *IT resource* is a physical or virtual IT-related artifact that can be either software-based, such as a virtual server or a custom software program, or hardware-based, such as a physical server or a network device.

Figure 126: Examples of common IT resources and their corresponding symbols.

The following image illustrates how the cloud symbol can be used to define a boundary for a cloud-based environment that hosts and provisions a set of IT resources. The displayed IT resources are consequently considered to be cloud-based IT resources.

Figure 127: A cloud is hosting eight IT resources: three virtual servers, two cloud services, and three storage devices.

19.3.3 Cloud Consumers and Cloud Providers

The party that provides cloud-based IT resources is the cloud provider. The party that uses cloud-based IT resources is the cloud consumer. These terms represent roles usually assumed by organizations in relation to clouds and corresponding cloud provisioning contracts.

19.3.4 Scaling

Scaling represents the ability of the IT resource to handle increased or decreased usage demands. It is possible to have two types of scaling:

- **Horizontal Scaling**, scaling out and scaling in.

- **Vertical Scaling**, scaling up and scaling down.

Horizontal Scaling. It is referred to the allocating or releasing of IT resources that are of the same type. The horizontal allocation of resources is referred to as *scaling out* and the horizontal releasing of resources is referred to as *scaling in*. Horizontal scaling is a common form of scaling within cloud environments. An example of Horizontal scaling can be seen in the following image where an IT resource (Virtual Server A) is scaled out by adding more of the same IT resources (Virtual Servers B and C).

Figure 128: Example of Horizontal scaling.

Vertical Scaling. When an existing IT resource is replaced by another with higher or lower capacity, *vertical scaling* is considered to have occurred. Specifically, the replacing of an IT resource with another that has a higher capacity is referred to as *scaling up* and the replacing an IT resource with another that has a lower capacity is considered *scaling down*. Vertical scaling is less common in cloud environments due to the downtime required while the replacement is taking place. An example of Vertical scaling can be seen in the following image, where an IT resource (a virtual server with two CPUs) is scaled up by replacing it with a more powerful IT resource with increased capacity for data storage (a physical server with four CPUs).

Figure 129: Example of Vertical scaling.

Comparison. A possible comparison can be found in the following table that provides a brief overview of common pros and cons associated with horizontal and vertical scaling.

Horizontal Scaling	**Vertical Scaling**
Less expensive (through commodity hardware components)	More expensive (specialized servers)
IT resources instantly available	IT resources normally instantly available
Resource replication and automated scaling	Additional setup is normally needed
Additional IT resources needed	No additional IT resources needed
Not limited by hardware capacity	Limited by maximum hardware capacity

Table 7: A comparison of horizontal and vertical scaling.

19.4 Cloud Service

Although a cloud is a remotely accessible environment, not all IT resources residing within a cloud can be made available for remote access. A *cloud service* is any IT resource that is made remotely accessible via a cloud. In the following image, it can be seen that a cloud service with a published technical interface is being accessed by a consumer outside of the cloud (left). A cloud service that exists as a virtual server is also being accessed from outside of the cloud's boundary (right). The cloud service on the left is likely being invoked by a consumer program that was designed to access the cloud service's published technical interface. The cloud service on the right may be accessed by a human user that has remotely logged on to the virtual server.

Figure 130: Example of Cloud Service.

The *cloud service consumer* is a temporary runtime role assumed by a software program when it accesses a cloud service. As shown in the below image, common types of cloud service consumers can include software programs and services capable of remotely accessing cloud services with published service contracts.

Figure 131: Examples of cloud service consumers.

19.5 Cloud Computing Advantages

Different benefits are associated with the usage of cloud computing, it is possible to highlight so, these benefits:

- **Performance**.
- **Cost reduction**, the number of investments is reduced and there's an access to powerful infrastructures without purchasing them (*proportional costs*). In relation to cloud consumers these other benefits can be included:
  - On-demand access to pay-as-you-go computing resources, such as processors by the hour, and the ability of release these computing resources when they are no longer needed.
  - The perception of having unlimited computing resources that are available on demand.
  - The ability to add or remove IT resources at a fine-grained level, such as modifying available storage disk space by single gigabyte increments.
- **Scalability**, clouds can instantly and dynamically allocate IT resources to cloud consumers, on-demand or via the cloud consumer's direct configuration. Similarly, cloud-based IT resources can be released as processing demands decrease. A simple example of usage demand fluctuations throughout a 24 hour period is provided by the following image.

Figure 132: Examples of demand fluctuations.

- **Availability** and **reliability**, cloud environment provides an extensive support for increasing the availability of a cloud-based IT resource to minimize or even eliminate outages, and for increasing its reliability so as to minimize the impact of runtime failure conditions.
- Maximize resource utilization.

19.6 Risks and Limits

Cloud computing can introduce distinct risks and limits that should be considered when we want to use it. In particular a cloud computing system:

- Requires high and available connection between nodes, which is not always available.

- Introduce security problems like authentication on the usage of shared resources and private resources.
- Lack of industry standard, public cloud are usually proprietary.
- Limited portability between cloud providers. In case of migration of the could provider from Cloud A to Cloud B it can happen that B does not support the same security technologies as Cloud A, this problem reduce portability between services.

19.7 Cloud Delivery Models

A cloud delivery model represents a specific, pre-packaged combination of IT resources offered by a cloud provider. It refers to the types of services offered by cloud computing. Three common cloud delivery models have become widely established and formalized:

Infrastructure as a Service (IaaS). The cloud provider offers access to fully functioning software. The service provider owns the equipment and is responsible for housing, running and maintaining it and the client typically pays on a per-use basis. The general purpose of an IaaS environment is to provide cloud consumers with a high level of control and responsibility over its configuration and utilization. The IT resources provided by IaaS are generally not pre-configured, placing the administrative responsibility directly upon the cloud consumer. This model is therefore used by cloud consumers that require a high level of control over the cloud-based environment they intend to create.
Available functionality: Full access to virtualized infrastructure related resources, and possibly to underlying physical resources.
Examples: Amazon Web Services, Google Cloud Storage, DigitalOcean.

Figure 133: Infrastructure as a Service structure

Platform as a Service (PaaS). The cloud provider give the way to rent hardware, operating systems, storage and network capacity over a network. The service delivery model allows the customer to rent virtualized servers and associated services for running existing applications or developing and testing new ones. The service provider offers a complete platform solution that the user runs software on. The platform usually consists of an operating system and an execution environment, often for a specific programming language. In this model, customers don't want to think about the server or its internals, they want to point to a virtual machine, tell their code or container to go live there, and let their application take over from there. It is a '**ready to use**' environment with IT resources already deployed and configured.

Available functionality: Some administrative control over resources relevant to cloud consumer's usage of platform.
Examples: Windows Azure, Google App Engine, Heroku.

Figure 134: Platform as a Service structure

Software as a Service (SaaS). The cloud provided rents applications software, that are hosted by the provider in the cloud and it is available to customers over a network. The user rents access to a physical or virtual server that can either run a predefined operating system or, in some offerings, a customised operating system. The user then has full control and responsibility over the running operating system and can use it to run any software. It is a software program as a cloud service available as a 'product' or generic utility to many consumers. Cloud consumer has a very limited administrative control.
Available functionality: Access to front-end user interface.
Examples: Google Apps, Yahoo!Mail, CRM software.

Figure 135: Software as a Service structure

19.8 Cloud Deployment Models

Cloud deployment model defines a specific type of cloud environment primarily distinguished by ownership, size and access.

- **Public Cloud**: the owner is the cloud provider and services are accessible for everyone and much used for the consumer segment. Examples of public services are Facebook, Google and LinkedIn.

- **Community Cloud**: the owner is a community of cloud consumers and it customers outside the community are not granted to access and use services and resources. A community cloud is similar to a public cloud except that its access is limited to a specific community of cloud consumers.
- **Private Cloud**: the owner is a single organization and only its customers can access to services and resources. Private clouds enable an organization to use cloud computing technology as a means of centralizing access to IT resources by different parts, locations, or departments of the organization.
- **Hybrid Cloud**: as the name suggests, a hybrid cloud deployment model consists of two or more cloud environments, most commonly private and public cloud. For example, a company may choose to deploy their sensitive and secure processing to a private cloud, while their less sensitive data is processed through a small public cloud. In other words, a cloud consumer may choose to deploy cloud services processing sensitive data to a private cloud and other, less sensitive cloud services to a public cloud

19.9 Potentialities and concerns

Why use a cloud computing system? We have different reason and we have analyzed them, but just to remember some of them we have: cost, scalability and performance.
In the contrary there are some cases in which a cloud computing system cannot be adopted like: legislative frameworks, medical records and data privacy.

20 Questions & Answers

20.1 Introduction

What is a distributed system? What does it mean transparency in distributed system? Define and specify the various types of transparency and give two examples.

A distributed system is a system composed by a set of processes that interacts each others in order to provide and satisfy some services. These processed are connected by a network and they communicate using a particular implementation of interprocess communication. Processed are autonomous since they can work alone. One of the main goal of a distributed system is to provide transparency to the client. Transparency hides some implementation to the client that can ignore, if it is possible, how the system work. There are different levels of transparency:

- location (DNS)
- access (URL in www does not provide access)
- mobility
- failure
- replication (proxy for multi-server)
- concurrency
- performance
- scaling

Define the different types of operating systems implemented for distributed systems. Discuss their advantages and limits.
In general are implemented 4 types of distributed systems:

- Uniprocessor system, implemented for single central systems which have a global view of the entire system.
- Network Operating System (NOS),is characterized by loosely-coupled HW and loosely-coupled SW, meaning that independent computer systems run independent applications. It is designed primarily to support more computer systems connected by a network. It manages resources provided by different computers and offers services to remote clients.
- Distributed Operating System, has tightly coupled software on a loosely coupled hardware, meaning that it is composed by independent computer systems with independent software that are coordinated by a controller. It is an operating system that runs on several machines whose purpose is to provide a useful set of services, generally to make the collection of machines behave more like a single machine. It provides: common IPC, global protection, process management, file system and high level of transparency.
- Multiprocessor Operating System, it has tightly coupled software on tightly coupled hardware with shared memory. There is a unique global execution queue and several CPUs that executes ready processes. In order to have an advantage from the presence of the cache it is important that the scheduler considers also in which CPU have already run processes ready for the execution.

20.2 Interprocess communication

Describe the inter process communication based on RPC. Define the software architecture and components. What are the main design goals, and the implementation

problems and choices?
Remote Procedure Call is a paradigm for interprocess communication. Client and Server are the two entities that want to communicate, interact. RPC consists on the usage of the client of procedures provided by the server, which provides an interface to use them. The main goal of RPC is to provide transparency such that remote procedure call are seen as local one. The two entities are composed by essentially these components:

- communication modules, present inside the client and the server. They implement and manage the communication between the two entities using request-reply protocol.
- stub, it make transparent the remote procedure call and execute marshalling and unmarshalling. Server stub applies unmarshalling and execute the service procedure required by the client. Client stub applies marshalling and make transparent the rpc to the client.
- dispatcher, select the right procedure that is required by the client request.

However, due to their different failure and performance characteristics and to the possibility of concurrent access to servers, it is not necessarily a good idea to make remote procedure calls appear to be exactly the same as local calls. **Design issues**: parameter passing (by values, by reference, by value), binding (static, dynamic), fault management.

Define the possible semantics associated to RPC and the relation with the fault management. Give an example for each case.
Invocation semantic defines the semantic provided to the client. The types of semantic consider possible retransmission of the request, filtering of duplicates and re-execution of multiple request. There are essentially 4 types of semantics that can be defined:

- exactly once, there is only one transmission and the request is executed by the server only once or completely not in case of fault.
- maybe, there is no-transmission of the message and there is no guarantee that it will be received by the server.
- at least once, when the sender send the request it set a timeout. If the client does not receive the answer before the timeout it will send again the request. The server can be execute one or more times the same request.
- at most once, there is possible re-transmission of the request but the server does not re-execute the same request. The server maintains an history table, recording the client request and its result.

Considering fault management the last 3 semantics the sender has the guarantee that the request is not executed or it is partially executed. Partial execution wastes time and resources inside the server but it is difficult to solve in distributed system, since only exactly one semantic have no partial execution. Exactly once is easy to implement in local system in which there is the possibility to know the state of the entire state.

Possible faults in RPC
Possible fault events during an RPC

- client cannot locate the server process
- loss of the client request message
- loss of the reply of the server
- server crash during the call

- client crash during the call

Primitives and possible faults in Request-Reply protocol
Request-Reply protocol is based on three fundamental primitives: doOperation, receive and reply.

- loss of messages by client or server
- network errors
- process fault, crashing of server or client

Describe communication based on RMI and its architectural components.
Remote Method Invocation is a communication paradigm based on invocation of methods provided by local or remote objects. The client requires the execution of a method provided by an object that can be on the same process or in a remote one. RMI provides a high level of transparency and it is very similar with the idea of RPC but has the advantage of use object oriented expression and powerful. RMI architectures is composed by:

- communication modules, implemented by server and receiver. They define request-reply protocol between the two entities client and server.
- remote reference module, contains a table reference to local or remote objects.
- proxy, provides location transparency to the client. it gives the idea the objects are local defined.
- skeleton, implements the methods provided by the server
- Dispatcher, selects the right method required by the client.

Define publish-subscribe system
Publish-subscribe systems are also referred as distributed event-based systems. They are the most widely used of all the indirect communication techniques.More specifically, whereas many systems naturally map onto a request-reply or a remote invocation pattern of interaction, many do not and are more naturally modeled by the more decoupled and reactive style of programming offered by events. A publish-subscribe system is a system where publishers publish structured events to an event service and subscribers express interest in particular events through subscriptions which can be arbitrary patterns over the structured events. A given event will be delivered to potentially many subscribers, and hence publish-subscribe is fundamentally a one-to-many communications paradigm. The main characteristics of publish-subscribe systems are: heterogeneity and asynchronicity.The programming model in publish-subscribe systems is based on a small set of operations; Publishers disseminate an event e through a publish(e) operation and subscribers express an interest in a set of events through subscriptions. In particular, they achieve this through a subscribe(f) operation where f refers to a filter –that is, a pattern defined over the set of all possible events. The expressiveness of filters (and hence of subscriptions) is determined by the subscription model. Subscribers can later revoke this interest through a corresponding unsubscribe(f) operation. When events arrive at a subscriber, the events are delivered using a notify(e) operation.