Cloud Computing and Distributed System

Ca' Foscari University of Venice
Department of Environmental Sciences, Informatics and Statistics

Afternotes
Academic Year 2021 - 2022

Student Zuliani Riccardo 875532

Contents

Chapter 1

Introduction to Distributed System

1.0.1 Distributed System Definition

The definition of Distributed System can be given in two ways:

- A **distributed system** is a system 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.
- A **distributed system** is composed of *more than one autonomous computer system* that interacts 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.

1.0.2 Cloud Computing Definition

Moreover we can define also **Cloud Computing,** it is a specialized form of distributed computing that introduces *utilization models* for remotely provisioning scalable and *measured resources*. Further we can define its five essential characteristics, its three service models ad its four deployment models.

- **Characteristics:**
  - On-demand self service
  - Measured service
  - Broad network access
  - Rapid elasticity
  - Resource pooling
- **Service models:**
  - Software as a service

  - Platform as a service
  - Infrastructure as a service

- **Deployment modules:**
  - Public Clouds
  - Private Clouds
  - Hybrid Clouds
  - Community Clouds

1.1 Advantages

Distributed System has lots of advantages:

- **Resource sharing:** sharing of hardware and software resources
- **Heterogeneity:** it can be composed of different components
- **Reliability:** if there is a crash the system is still alive
- **Extendibility / Scalability**:
  - *Extendibility:* possibility to include another node in the system
  - *Scalability:* possibility of including a new node of the same type

Not necessary that the added node has to be of the same type of the others

- **Performance:** they offer results in an efficient way and optimum values of throughput and response time.
- **Transparency:** is defined as *hiding* the separation of components in a distribution system from the user and the application program. In a way such that the system is *perceived as a whole* rather than as a *collection of independent components*.

1.2 Features

Distributed System has lots of features:

- **Concurrency:** nodes can work in parallel
- **Autonomous and asynchronous:** a node can survive alone and each machine has its own clock
- **Resource sharing:** access to resources can be shared, so a DS implements a strategy to manage them
- **Lack of global time:** when a process need to cooperate they coordinate their actions by exchanging messages. Since each machine is composed of its own clock, they are not able to coordinate synchronously.
- **Faults independent:** all computer system can fail, and it is their responsibility to design a plan for the consequences of possible failures. Possible faults of nodes do not affect other nodes in the network.

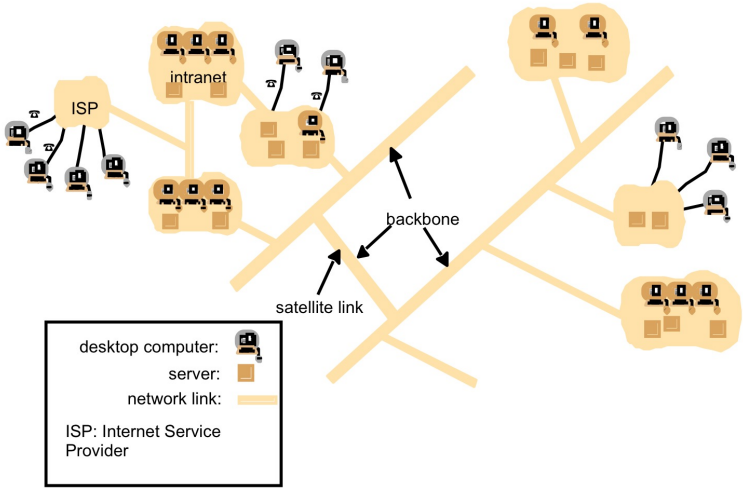


Figure 1.1: Internet as Distributed System

1.3 Goals

- **Economy:** since data can be shared is not necessary to replicate it
- **Software:** the implementation follows particular standards
- **Flexibility:** gives a clear interface to use the system
- **Availability:** the system detects faults and applies recovery operations
- **Performance:** optimal performance in term of:
  - Response time
  - Throughput
  - Parallelism
  - Bottleneck reduction
- **Locality and control distribution, security and efficiency**
- **Transparency**

1.4 Computer System vs Distributed System

- A *Computer System* is characterised by *single hardware*, system software and application software for data or control
- A **Distributed System** is composed of **distributed hardware** that can have or not **distribute software / application** to manage data and control
  - **Distributed hardware:** system composed of *more than two computer systems*, interconnected by a communication network and each *computer is independent* from the others but can *interact with them*

1.4.1 Control

Control is essential to manage physical or logical resource on the system, it can be:

- **Centralized:** there is a unique responsible entity
- **Distributed:** computer system on the net cooperate to reach the solution
- **Hierarchical:** system is more scalable since all the functions are subdivided in different modules

1.5 Open Problems

- **Data Sharing:** understands how and what resources must be shared and it provides a *system to retrieve them* efficiently
- **Heterogeneity:** enables users to access services and run applications over a heterogeneous collection of computers and networks
- **Concurrency**: several clients could attempt to access a *shared resources* at the same time. The process that administrates shared resources could take one client request at a time.
- **Openness:** *the system is not completely defined*, but it is possible to extend it, for instance *including a new machine*. Open distributed system can be constructed from heterogeneous hardware and software, possibly from different vendors
- **Middleware:** is an intermediate layer that provides an unique public interface
- **Mobility:** it refers to *program code* that can be *transferred* from one computer to another and run at the destination
- **Security:** it is necessary to develop a secure system that *does not allow unauthorized* users to read / write data:
  - *Confidentiality:* protection against revelation to unauthorized individuals
  - *Integrity:* protection against alteration or corruption
  - *Availability:* protection against interference
- **Scalability:** if it remains stable even if there is a significant increase in the number of resources and users
- **Quality of Services (QoS):** reliability, security and performance of the entire system
- **Fault management:** faults need o be *transparent to the user*. One of the most common recovery algorithm is "**checkpoint and rollback**" in which we go back the last checkpoint to recover the last consistent state
- **Transparency:** system is perceived as a *set* instead of a *collection of independent components*. Moreover we can analyse different type of transparency:

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

1.6 WWW as Example

World Wide Web 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 distributing its existing functionality
- **Client-server architecture**
- **Transparency:** using DNS it reaches the *location transparency*. With the usage of symbolic server names it is not necessary for the client to know the exact physical address IP of the server

Chapter 2

Distributed System

2.1 Communication

There are two type of communication character:

- **Communication entities:** can answer the question *"What is Communication?"*
- **Communication paradigm:** concerns on *"How do they communicate?"*

Entities can be called as:

- *Nodes or Processes* → **system-oriented entities**
- *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
- **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.** Like:
  - Request-replay protocols
  - Remote procedure calls
  - Remote method invocation
- **Indirect Communication**: allows a strong **degree of decoupling (disaccoppaimento)** between *senders* and *receivers*. Senders don't need to know who they are sending to and both senders and receivers don't need to exist at the same time. Like:
  - Group-communication
  - Publish-subscribe.

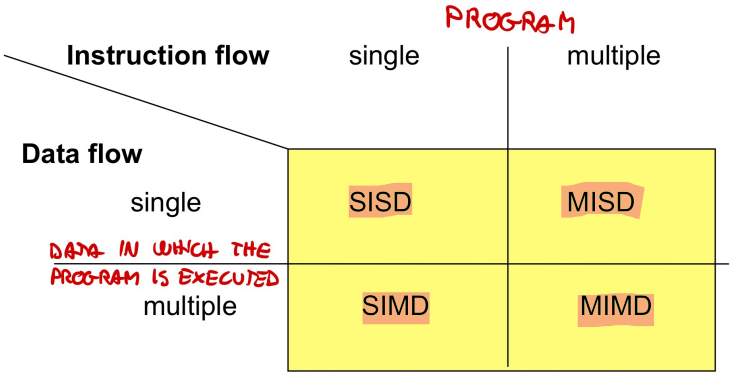


Figure 2.1: Flynn Classification

2.2 Hardware Organization

The main goal of the hardware organization is to **improve the performance while maintaining the cost as low as possible**. There can be developed:

- **Centralized System**: that increase the CPU speed but has some physical constraints
- **Distributed system**: for which there can be different various models divided according to the **Flynn Classification**:

Now we analyse each element of the table

- **SISD**: it corresponds to the Von Neumann model with a *single program executed by a CPU and data stored inside the memory*. In order to improve the performance of the system some techniques are adopted like:
  - **Parallelism**: in which are used many specialized ALU that execute each single operation with high speed
  - **Pipeling**: in which are executed simultaneously more operations for different instructions
  - **Hierarchies of Memory**: like cache levels
  - **Specialized Hardware**

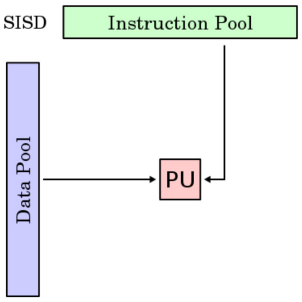


Figure 2.2: SISD

- **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. Also in this case there can be different architectures that determine how to organise the system, like:
  - **Vectorial Architecture:** where instead of one ALU that works on a single variable for each input, there is a vector of ALU working on a vector of input values and producing a vector of output.
  - **Array Processor:** represented as a matrix processor x memory, in which each processor executes the instruction on the available data in the local storage.

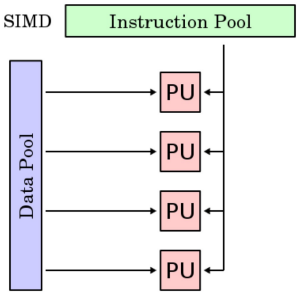


Figure 2.3: SIMD

- **MISD:** Multiple Instruction Single Data

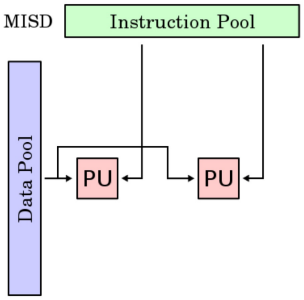


Figure 2.4: MISD

- **MIMD:** typical distributed systems in which *different CPU executes different programs on different data streams*. Each CPU is associated with his Program Counter, memory, data and programs and they cooperate to execute and provide a common service. A MIMD system, implemented by a distributed system, can be organized in different ways, ***Multiprocessor* or *Multicomputer***.

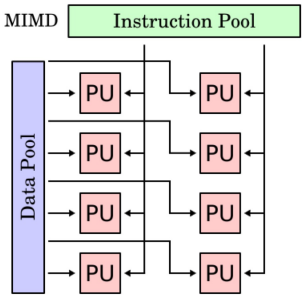


Figure 2.5: MIMD

2.3 Multiprocessor

Multiprocessor is characterised by a *set of CPUs* connected in a *communication network* in which there is a **shared memory.** The advantage of this design, since CPUs are **strictly coupled**, are that:

- It has **short transmission delay**
- And **high transmission speed**

2.3.1 Bus

CPUs are connected in a **bus** and they *share the same memory*. They have to interact to determine who can access the shared memory. We can imagine that it is necessary to **implement a synchronisation system to access the memory.** An improvement of this architecture is based on the **usage of a private memory**,

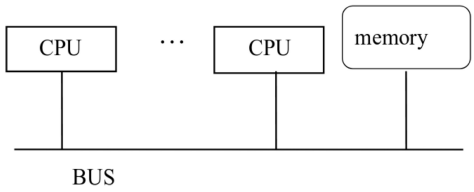


Figure 2.6: Multiprocessor - Bus Architecture

in which each CPU has also a **private cache memory** where the most recently used words. If the required word is stored inside the cache so there is a hit and it is not necessary to access the shared memory. The improvement is needed since there is the problem of **coherence** in which for example:

- At time t CPU A writes value x at the memory address y
- At time t' CPU B writes value x at the memory address y

A possible solution is **write through cache**, each writing in the cache is brought to the memory. Moreover all the CPU checks for possible writing and possibly they eliminate or update the word (**snoopy cache**).

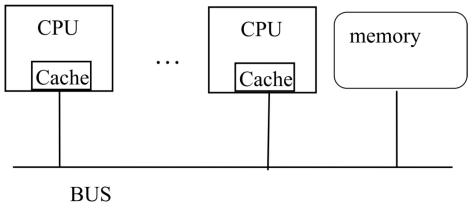


Figure 2.7: Multiprocessor - Bus with Cache

2.3.2 Switch

The **shared memory is subdivided into** n **modules** and a particular element, called **switch**, are used to *decide the right path*. With this implementation different CPUs can access different memory modules at the same time. N memory modules and N CPUs require N^2 switches. If all CPUs want to access the first memory we have the **array-like case**, otherwise we could introduce some sort of parallelisms

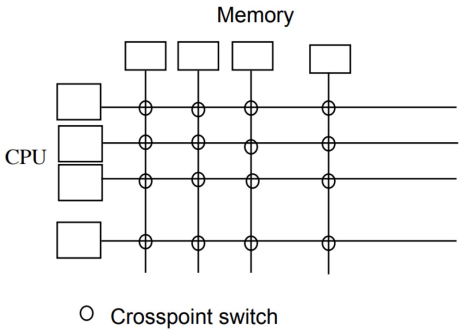


Figure 2.8: Multiprocessor - Switch

2.3.3 Omega Network

Respect to switch architecture, switches are organized as a **binary tree** with direct access from each CPU to each module, so there is a **decreasing number of switches** ($\frac{n}{2}\log n$). But the **drawback** is the *communication delay* of each switch.

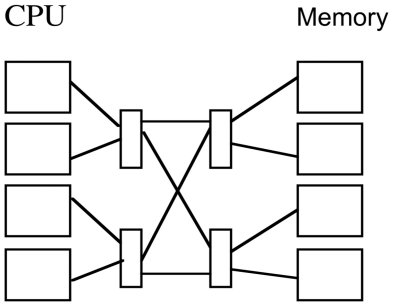


Figure 2.9: Multiprocessor - Omega

2.3.4 NUMA

NUMA (Not Uniform Memory Access), **each CPU has a local memory**, this brings an **increase of performance** since CPUs save most of the data of shared memory inside local memory. It is necessary to develop an allocation algorithm for programs and data.

2.4 Multicomputer

- Each computer system is composed of a **private local memory**
- They can be local at different positions, loosely coupled, so:
  - Longer transmission delay
  - Lower transmission speed
- The computer systems are **autonomous**
- They are **connected by a network** and they interact using it

Also here there can be different architectures:

2.4.1 Bus

Computer systems are connected by a **bus line** but there is **no shared memory**. They can *access remote resources*. It is necessary to implement a system to reduce traffic and some communication network protocols.

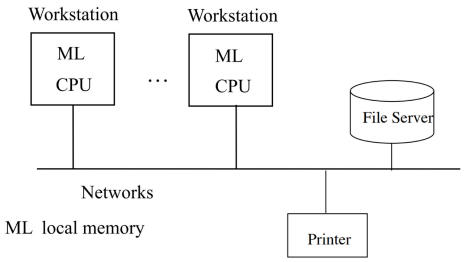


Figure 2.10: Multicomputer - Bus

2.4.2 Switch

Computer systems can be organized as a **grid of switches or hypercubes**, that reduces the number of steps and switches necessary to connect all the nodes.

2.5 Type of Distributed System

- **Systems with massive parallelism**, like supercomputers with thousands of CPUs, which have *low latency, high bandwidth and good reliability*

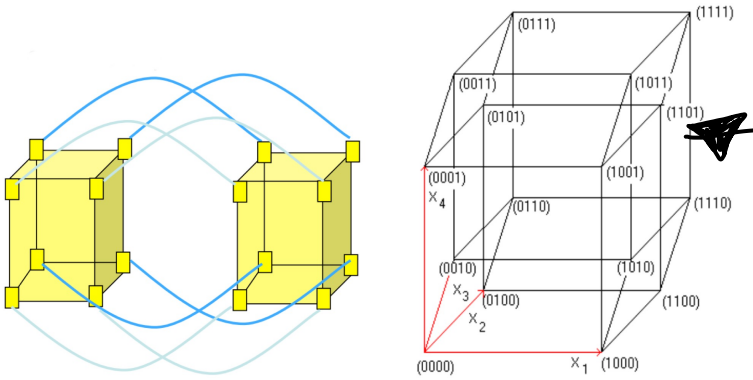


Figure 2.11: Multicomputer - Switch

- Systems with **clusters of workstations (COW)**, computer systems connected with available components. There is no special design to optimise performance and reliability. Like homogeneous systems.
- **Cloud computing systems:** in which there are computers as utility, resource visualisation, on demand feature, pay-per-use model, high scalability and transparency
- **Distributed transaction processing systems:** like database applications
- **Pervasive, ubiquitous, mobile systems**
- **Sensor networks:** energy critical systems

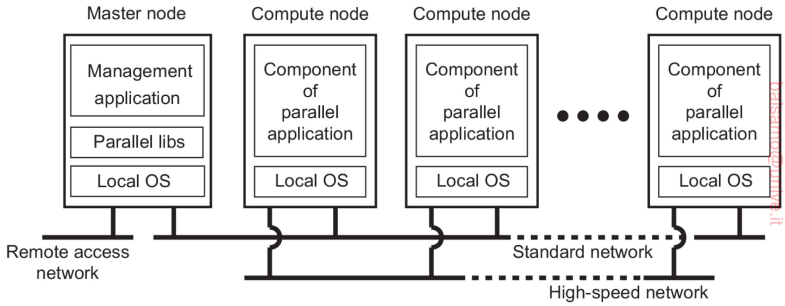


Figure 2.12: Example of a cluster of computer systems. A single program can run in parallel on a set of c.s. Master: process allocation, user interface and Compute nodes: use the middleware

2.6 Software Organization

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

- **Loosely-coupled:** in which machines are **completely independent and they share resources.** Each computer system can be identified and it is able to operate independently of others in such a way that a network fault does not block the operation of a single computer system

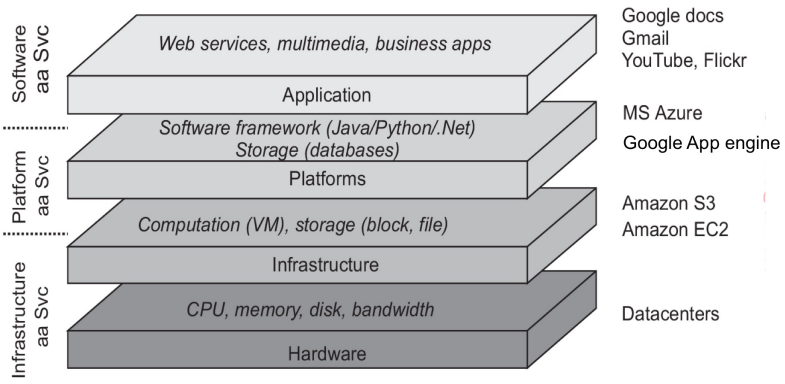


Figure 2.13: Organization of cloud computing - levels of services - Example

- **Tightly-coupled,** machines are **independent** but there is a **controller** that coordinates them. We can say that there exists a hierarchy structure.

We can distinguish several Operating Systems depending on the type of the system we are dealing with:

- **Uniprocessor system:** for a single central system the OS provides the virtualization of the machine and manages the resources and 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, LCS - LCH):** 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

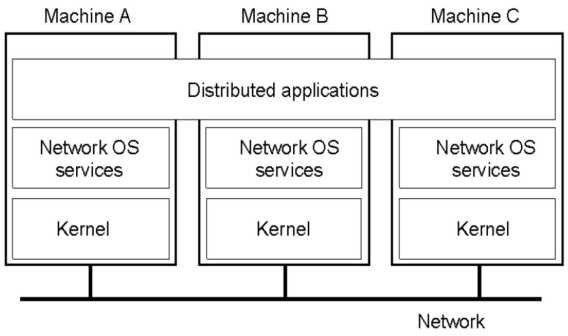


Figure 2.14: NOS Architecture

- **Distributed system (DOS, TCS - LCH):** has a **tightly coupled software** on a **loosely coupled hardware**.
  - Everything is managed by a **global manager, without sharing memory.**

- It is an operating system that produces a single system image for all the resources.
- A DOS is an operating system that runs on **several machines** whose purpose is to provide a useful set of services.
- DOS also has the role in making the collective resources of the machines 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.
- A DOS should provide:
  * Common IPC, provides mechanisms to allow the processes to **manage shared data**
  * **Global protection**, provides protection mechanisms to maintain secure systems and data
  * **Process management**, which provides mechanisms to manage processes
  * **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

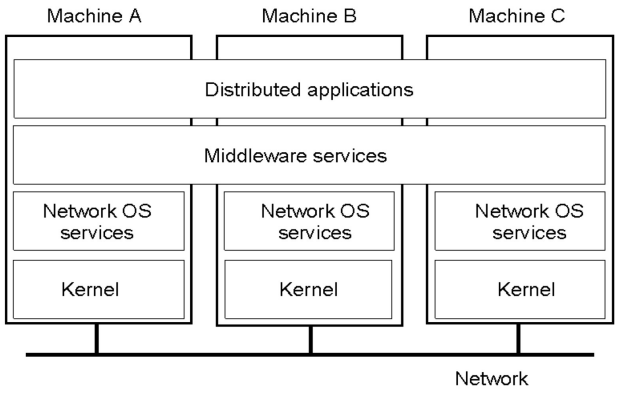


Figure 2.15: DOS Architecture

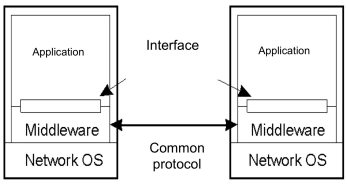


Figure 2.16: In an open distributed system based on **middleware** the used protocol by each middleware level should be the same. Like the offered **interface**s to the application

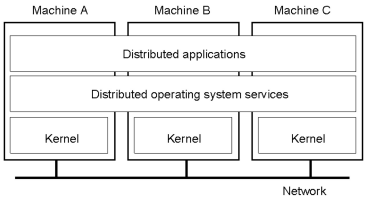
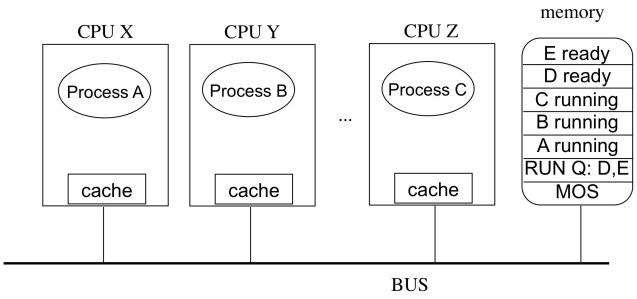


Figure 2.17: General structure of **multicomputer** operating system

- **Multiprocessor system (MOS, TCS - TCH):** has a **tightly coupled software** on a **tightly coupled hardware**.
  - It has a **global shared memory**
  - There is a **unique global execution queue** and several **CPUs** that execute 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**.



2.7 Summary of Distributed System - Software Architectures

System	Description	Main Goal
MOS **DOS**	Tightly-coupled operating system for multi-processors and homogeneous multicomputers	Hide and manage hardware resources
NOS	Loosely-coupled operating system for heterogeneous multicomputers (LAN and WAN)	Offer local services to remote clients
Middleware	Additional layer atop of NOS implementing general-purpose services	Provide distribution transparency

Categories	NOS	DOS Multicomputer	MOS Multiproc.	Middleware
Transparency	NO	Yes (high)	Yes	Yes
The same OS	NO	Yes	Yes	NO
Copies of OS	n	n	1	n
Communication	shared file	messages	Shared memory	Model dependent
Communication protocol	Yes	Yes	NO	Model dependent
Single RUN Q	NO – per node	NO – global distributed	Yes - central	NO – per node
sharing with files with common semantics	not necessarily	Yes	Yes	Yes
Scalability	Yes	Moderately	NO	Model dependent
Openness	Open	Closed	Closed	Open

Chapter 3

Model of System

The overall goal is to ensure that the structure will meet present and likely future demands on it.

3.1 Architecture Model

An architectural model provides the definition of what are components, structures of the system and functions associated with them. There are different types of processes:

- Client
- Server
- Peer

Architectural models can be classified in client-server or peer processes. A **client-server architecture** can be based on layers or tiers.

- A **layer architecture** splits a 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.
- **Tired architectures** are complementary to layering. Whereas layering deals with the vertical organisation of services into layers of abstraction. This technique is most commonly associated with the **organisation of applications and services**. Each tier level is used as a 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 rely on 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*.

This is a single layer architecture since there is a single communication phase in which client send a request to the server, it provice the service and it reply to the client

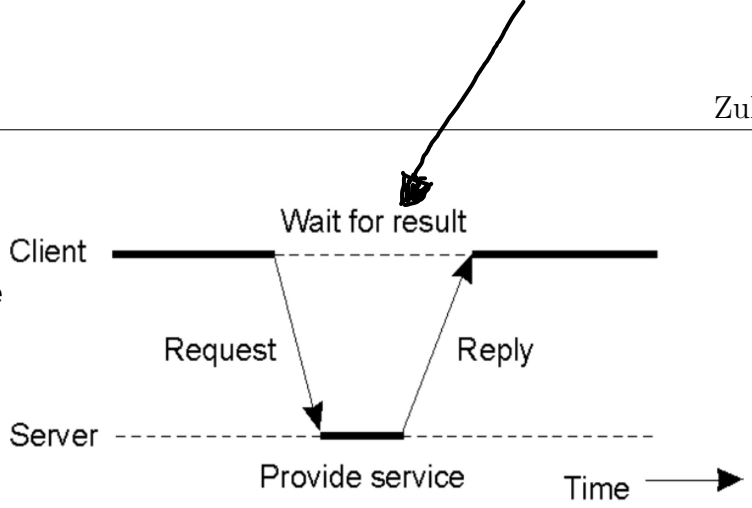


Figure 3.1: Client - Server Single Tier Architecture

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

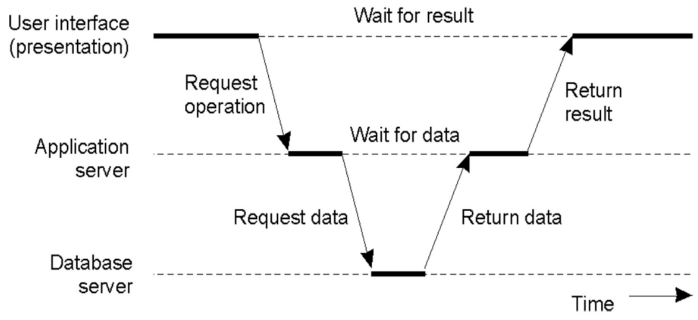


Figure 3.2: Client - Server Multi Tier Architecture

Client-Server architecture can be composed of a **set of servers** that provide the same services and they can interact to reduce the load of each server. Respecting the previous simple model, with only one server, these kinds 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 famous to 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. 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.

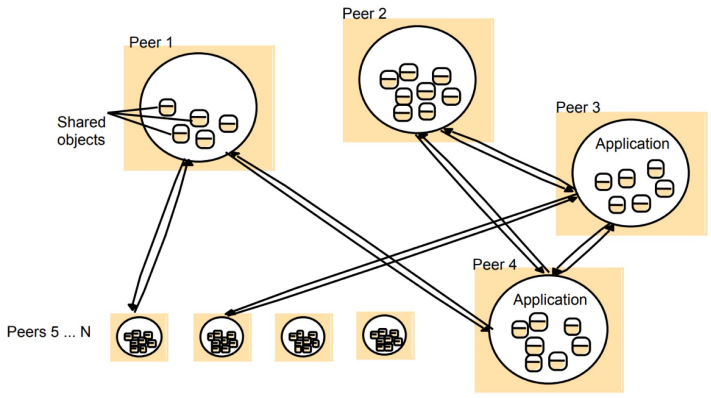


Figure 3.3: Peer-to-Peer Architecture

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

Could be **mobility** of the node, of the code (program can be moved) or data.

- Like **Applets**, which 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 a **good interactive response** since it does not suffer from the delays or variability of bandwidth associated with network communication.

  When a client requests something from 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 that travels from one computer to another in a network carrying out a task on someone's behalf (comportamento), such as collecting information, and eventually returning with the results.

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 process interact*.Like using message passing, moreover we have to consider also the delay time.

- **Faults 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 system 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.
- **Performance Model**

3.2.1 Interaction Models

Messages are transmitted between processes to transfer information 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.

- **Latency:** the delay between time of starting transmission and time starting to receive the message by receiver
- **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. Measurement relevant for streaming and multimedia purpose.
- **Lack of Global Time** There is not a unique clock but each process has its own clock and they usually are not synchronised. 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 systems processes are synchronized and so it is easier to develop communication protocols
  - Instead, asynchronous systems are not bounded, so it is necessary to develop different strategies to provide communication protocols

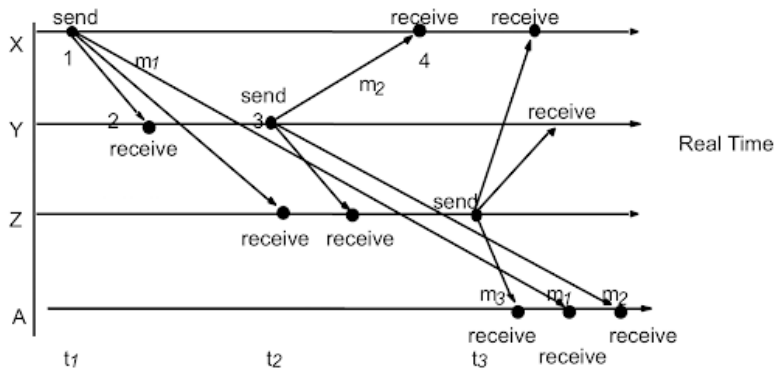


Figure 3.4: Organization of cloud computing - levels of services - Example

3.2.2 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. There are different types of failure:

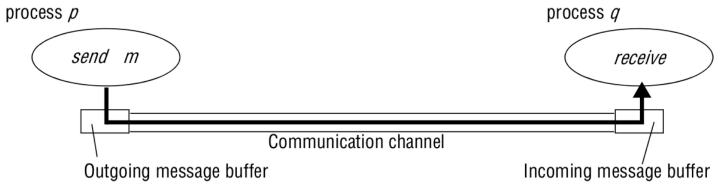


Figure 3.5: Organization of cloud computing - levels of services - Example

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.

3.2.3 Security Models

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

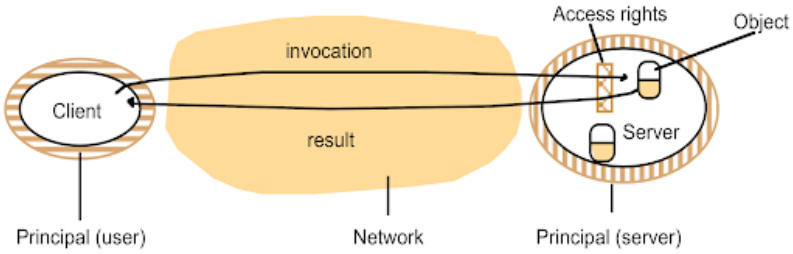


Figure 3.6: User Process with Autority

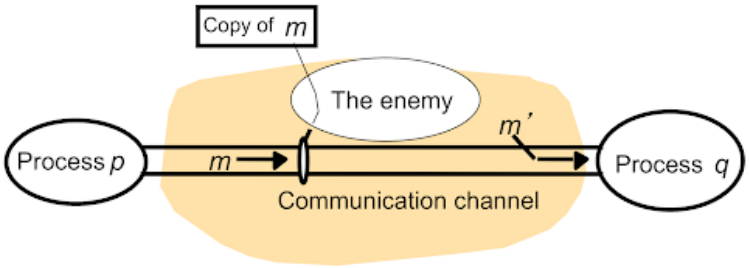


Figure 3.7: Communication Attack

are different type of attack that a enemy could take:

- *Eavesdropping:* obtain private or secret information
- *Masquerading:* assuming the identity of another user
- *Message tampering:* altering the content of messages in transit (ma in the middle attack)
- *Replaying:* storing secure messages and sending them at a later date/time
- *Denial of Services:* flooding a channel or other resource, denying access to others

A secure channel must take into account these security features:

- Cryptography and shared secret information like key

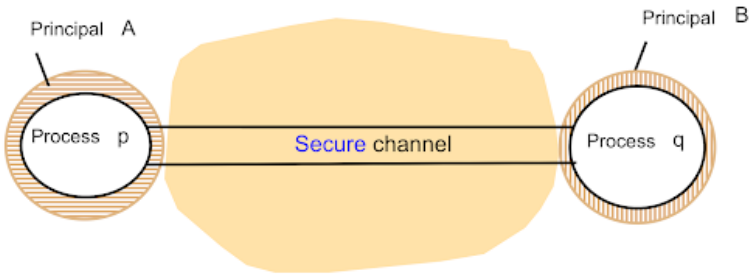


Figure 3.8: Secure Channel

- Authentication based on proof of ownership of secret
- Privacy and integrity
- Use of timestamp to avoid messages from being recorded
- VPN, SSL, SFTP

3.2.4 Performance Models

The most important performance indexes are:

1. Throughout (X)
2. Mean Response Time (R)
3. Utilization (U)

In this section we focus in three main cases:

Performance models with basic queue

This is a simple queried model for a centralised system where we have three general assumptions:

- Independence job
- Exponential times
- Infinite buffer queue

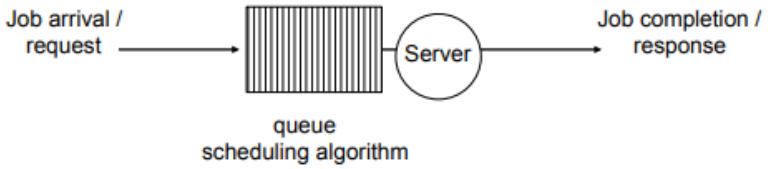


Figure 3.9: Organization of cloud computing - levels of services - Example

Now we introduce two type of measure:

- Job arrival rate: λ request/sec
- Service rate: μ request/sec

Performance models with basic queue and one server

This is a basic queuing (fare la fila) model for a centralized system where we have exponential and independence assumptions. *Proprieties:*

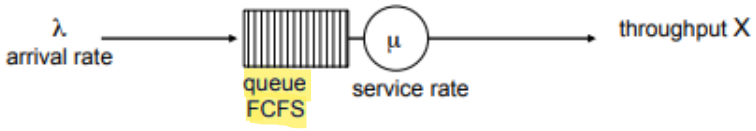


Figure 3.10: Performance models with basic queue and one server

- The model is stable if $\lambda < \mu$
- We define the traffic intensity $p = \lambda/\mu$
- Utilization (U): $U = p$
- Mean Response Time (R): $R = 1/(\mu - \lambda) sec$
- Throughput (X): $X = \lambda \frac{req}{sec}$
- Mean Queue Length (N): $N = p/(1-p)$

Queue Length probability: $\pi_i = p^i \times (1-p) i \geq 0$ probability of i jobs in the systems

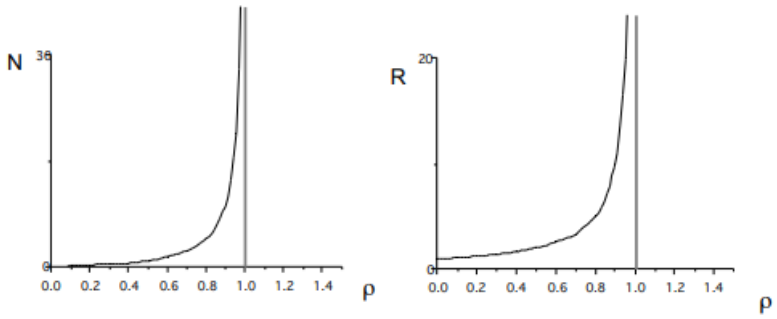


Figure 3.11: Organization of cloud computing - levels of services - Example

Performance model with queue and multiple server

Basic queuing model for a system with multiple servers (processors), where we have the exponential and independent assumptions. Here we have to introduce an additional type of measure, so in summary we have:

- Job arrival rate: λ request/sec
- Service rate: μ request/sec
- Number of servers per processors: (m # servers) / processors

Proprieties:

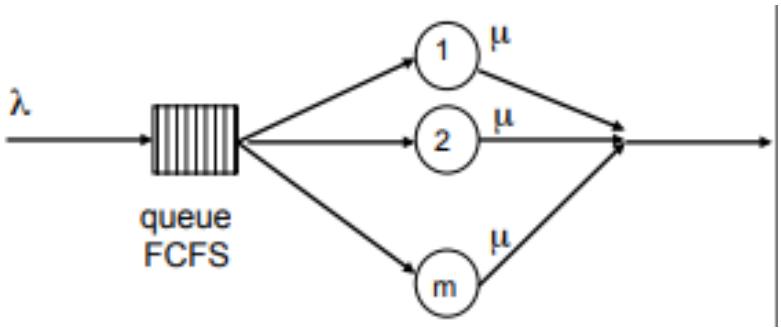


Figure 3.12: Performance model with queue and multiple server

- The model is stable if $\lambda < m \times \mu$
- We define the traffic intensity $p = \lambda/(m \times \mu)$
- Utilization (U): $U = p$
- Mean Response Time (R): $R = N/X$ sec
- Throughput (X): $X = \lambda \frac{req}{sec}$
- Mean Queue Length (N): $N = (m \times p) + (\pi_m \times p)/(1-p)^2$

Queue Length probability:

1. $\pi_i = \pi_0 \times (m \times p)^i / i!$ for $i \leq i \leq m$ probability of i jobs in the systems
2. $\pi_i = \pi_0 \times m^{(m)} \times p^{(i)} / m!$ for $i > m \leq m$

Relations of Performance Models with basic queue

- *Little's Law:* $N = X \times R \rightarrow$ average # of jobs in the system = throughput × mean response time
- *Utilization Law:* $U = X \times S \rightarrow$ system utilization = throughput × average service time
- *Flow Forced Law:* $X_k = X \times V_k \rightarrow$ throughput of a component k = system throughput × average # visit to component k

Chapter 4

Client Server Communication

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

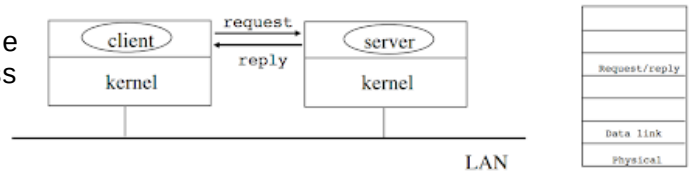
4.1 Addressing

Addressing refers to the ability of a client to **understand where the server is (server localization)**. There can be different solutions to solve these problems and answers to questions like *"Where is the server?"*.

4.1.1 LAN

The assumption that **client knows the address of the server and server gets also the address of the client** when it receives a request. The problem of this solution is that **there is no location transparency** and in case of relocating to the server it is necessary to apply modification on the client.

since both client and server know each other the respective adderess



So how can we find a solution to obtain the location transparency?

Figure 4.1: Client Server LAN

4.1.2 Broadcast

1. Client sends via broadcast a **special packet** *"locate"* to identify the server and discover where it is located
2. Only the server answers *"I Am Here"* with his physical address
3. The client send the *"request"* to that address
4. The server sends a *"reply"* to the client

5. This solution **offers location transparency.** The **problem** regards the *usage of channels*, since now there is an **increasing amount of traffic** due to broadcast flooding.

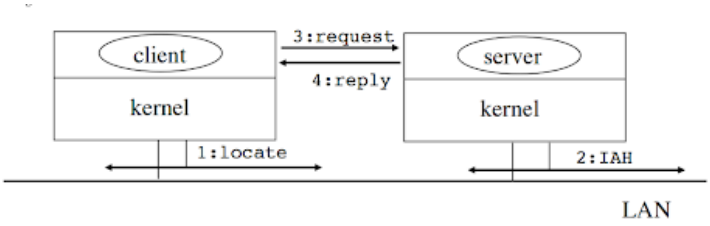


Figure 4.2: Client Server Broadcast

4.1.3 Usage of Name Server

It provides mapping of local names of the servers and physical addresses:

1. The client sends to NS a message *"lookup"* to get the server address
2. The NS answer with a *"replay"* message with the physical address of the server
3. The client sends a *"request"* to the server
4. The server sends a *"reply"* to the client

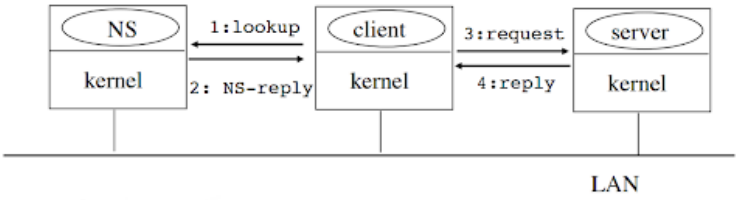


Figure 4.3: Client Server Name Server

- This solution **provides location transparency** for the *server*
- But it does **not provide location transparency** for the *NS* since for each communication the client has to contact the NS, so that could be considered as a bottleneck.
- Possible replication for robustness, since it is hard to keep consistency of the copies

4.1.4 Primitives / Operations

1. **Synchronous:** processes are blocked when they use primitives.
   - *Send:*
     - It specifies the **send buffer** and the **destination address**
     - Process remains **blocked** until the buffer has been sent.
   - *Receive:*
     - The invocation does not return control to the calling process until the message has been received and deposited into a destination buffer specified from the calling

2. **Asynchronous:**
   - *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 it takes to copy the message inside the buffer.

buffer to be sent -> kernel buffer

   - *Receive:*
     - Indicates to the kernel the local base address and the size of the buffer where data is received
     - It returns control to the calling process.

kernel -> loca base address and buffer size

   - *Advantages:* the caller **continues the computation** concurrently to the communication
   - *Disadvantage:* it is difficult to **recognize when the communication is terminated.** How can we migrate this problem?
     - Implement an **alternative** to asynchronous send
       - The **buffer** to be sent is in the **process space** → the caller cannot change it until it is transmitted
       - The **buffer** to be sent is in the **kernel space** → cost of copying
     - **Interrupt** from the core when the buffer has been transmitted
       - Does not require copying
       - The sender can reuse the buffer after the interrupt

3. **Primitives with buffer:** receiver has a buffer in which it can **store messages** sent by sender and it can be introduced in synchronous and asynchronous systems:
   - 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 arrive with the process address and are put in the mailbox
     - If there is a **pending receive** the message is passed to the process, eventually unblocking it, otherwise the message is kept until an invocation does not arrive.

- If the mailbox is full the kernel could discard the message → **message loss event**

4. **Primitives without buffer:** so the communications **must be synchronous**, sender is blocked until it receives an unblock

   - 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"*, the 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

4.2 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 server -> client: information about the communication

ack client -> server distinguish request

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 requests and discard copies of replies.

- **AYA, IAA:** if a client sends a request and the answer **does not come back** in a given time: is the server working properly?
  - The client sends a message AYA to test the server if the answer is a message IAA or REP, ok, otherwise it sends again an AYA after some attempts it can assume that the server is not alive
- **TA:** sometime the server **cannot accept a REQ**, thus:
  - The server does not have buffer space for new message

  - The server is overloaded

- **AU:** mistake in the address → the client has not to try again

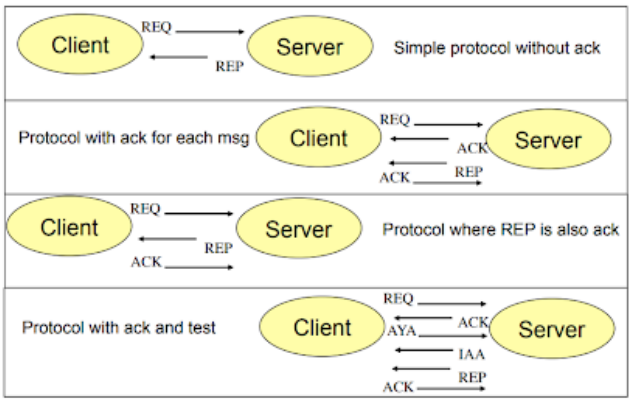


Figure 4.4: Examples of interaction Client-Server

4.3 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.

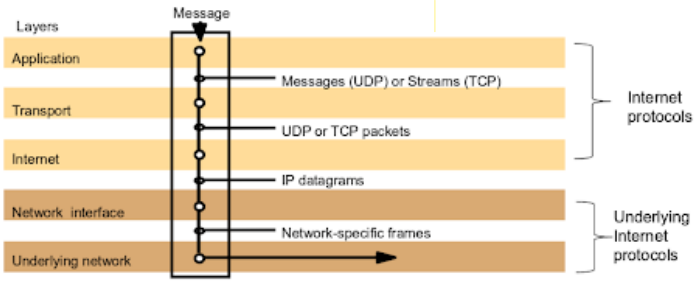


Figure 4.5: Networks / Reliability

Chapter 5

Interprocess Communication

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

- **Remote Procedure Call:** allowing a calling process to call a procedure in a remote node as if it is 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
- **Events:** asynchronous notification to objects of events

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 distinguishing local and remote procedures**. The development of RPC and RMI includes:

- **Marshalling:** technique to codify parameters in order to be sure that the structure given to the procedure is preserved. This problem arises from the possibility of a computer system to represent data in different ways. Marshalling codifies 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**
  - *Static:* decided a priori how to reach the server -¿ faster since there is a direct communication to the server
  - *Dynamic:* discovers where is the server dynamically → higher level of **location transparency**

The **Middleware** provides *transparency*, the procedure calls are used in the 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.**

5.2 Object Oriented Model

RMI is based on the **Object Oriented Model (OOM)**, in which components are called objects. An **object** is an entity that contains a set of data and methods that define its behavior. 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 does not specify how they are implemented.
- **Methods:** are operations that define the *behavior of objects of the same class*

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

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

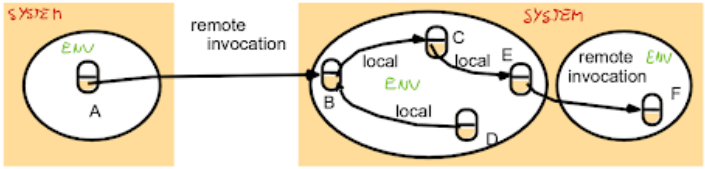


Figure 5.1: Example of usage of remote and local methods

5.3 Invocation Semantics

Remote procedure calls provide 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:

- Re-transmission of the request

- Duplicate filtering in the server in the case of re-transmission
- At arrival of a re-transmitted request: re-execution of resend of the result

There can be **different types of call semantics**, 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 sent 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 5.2: Invocation Semantics

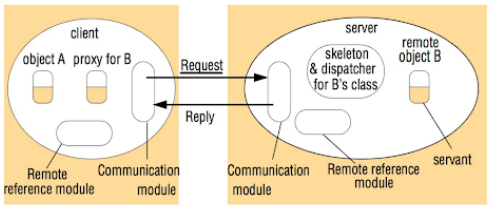
- **Maybe:** the request is **sent only once** independently of the fact that the client receives a reply or not
  - There is **no fault tolerance measure**, there might be loss of the invocation or the result or crashing of the server.
  - The client does not know if the RMI or RPC has been **successful or not.**
- **At least once:** there is **re-transmission until the client does not get a reply**.
  - Possible duplicate executions
  - It can be used only for **idempotent operations** → operations that return the same result at every execution and that do not produce any change of state.
- **At most moce:** there is **re-transmission until the client does not get a reply**, but the server **remembers the operations already executed** and does not reply to 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
    - *Positive answer:* it returns the result without replay the execution
    - *Negative answer:* it compute and return the result

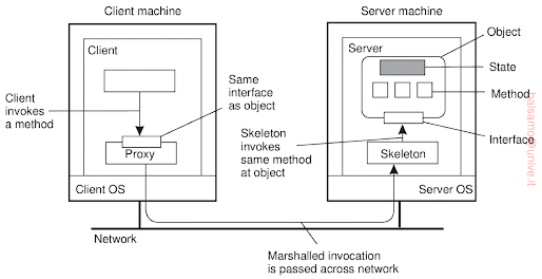
  Operations for this semantic are called **non-idempotent**, since each time they are executed, they produce a change of state inside the server

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

5.4 RMI procedures are characterized by:

- **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
- **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, 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





5.5 RPC procedures are characterized by:

- **Stub:** its main role is to make **transparent the remote call** and **execute marshalling**
  - *Client stub* makes transparent the remote call and execute marshalling
  - *Server stub* applies un-marshalling and execute the service procedure required
- **Dispatcher:** it opens the request message and it selects the procedure corresponding to the request

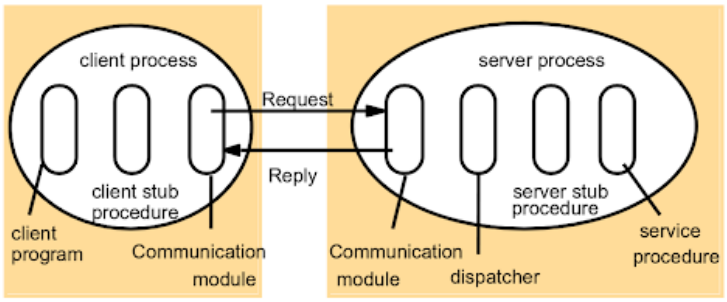


Figure 5.3: RPC

5.6 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 on the performance. Request-Reply protocol is not ideal so a possible solution is to reverse the interaction:

- Server notifies the client when it is 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

Advantages: it avoids repeated calls from the clients that increase the performances

Disadvantages: the server has to keep a list of the interested clients and realize RMI to all the callback in the list

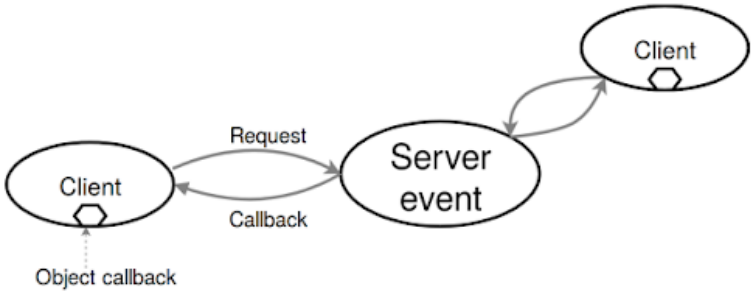


Figure 5.4: Callback

Event Notification

Chapter 6

Interprocess Communication - Implementation & Design

There are many possible kinds of communications, so we will illustrate the three most important

- **Socket:**
  - Creates a **channel** at low level between processes on different hosts
  - In which for each application it is necessary to define a **protocol for data coding and decoding**
  - This solution implements a **lack of transparency**
- **Remote Procedure Call or RPC:** allows client programs to call procedures transparently in server programs running in separate processes. More deeply:
  - There is an **abstraction** of the process communication at a level of procedure call
  - The parameters are **packed and sent** to the destination after the encoding
  - Data **conversion** between different supports introduces a very **high overhead**
  - It is connected to the **process definition** and it cannot be integrated in the Object Oriented code
- **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** is created, that is actually active on a different host
  - All the invocation of the methods to be transmitted are handled by the **Object Remote Broker (ORB)**

Middleware provides transparency to the interprocess communication:

6.1 UDP

UDP represents a **datagram communication without ordering and reliability.**

- Interface to UDP is based on **message passing** to enable sending a single message from sender to receiver
- Communication procedure is based like so:
  - **Client** (sender or receiver):
    - Create socket
    - Link it to the IP address and port Server
  - **Server:**
    - Create socket
    - Link it to a port
    - Let the clients know the port
  - There can be many type of **failure like:**
    - **Omission:** lost of message (channel fault, checksum, limited space)
    - **Arbitrary:** unordered delivery

Moreover the communication procedure in JAVA UDP is formed by two clases:

- **DatagramPacket:** which for the actual transmitted packet formed as so:

Figure 6.1: DatagramPacket components

- **DatagramSocket:** which provides the number of the port to a process that asks for it allowing the system to choose a free port

For sending and receiving there are three methods to extract the information:

- getPort
- getData
- getAddress

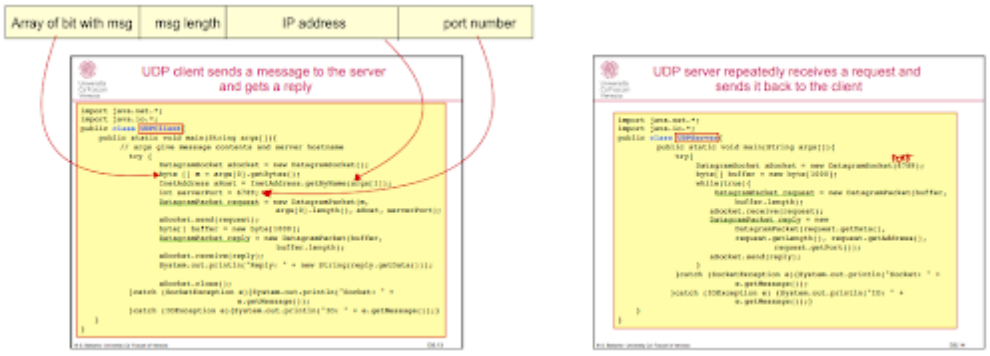


Figure 6.2: Java Datagram Application

6.2 TCP

TCP opens a **stream of communication**

- Interfaces to TCP are based on **bidirectional stream** and to enable **sending a stream of data** between sender and receiver
- It is based on **producer-consumer communication**
- In TCP the destination is the couple (IP address, port number), so the **server** has to let the client **know** this **couple of data**
  - This means there is a **lack of transparency**
    - It could be solved by referencing to the service by name and using a **name server**
    - Moreover the server OS provides the name so that there is **location independence = location transparency + migration transparency**
- The communication in TCP is based like so:
  - **Client:**
    - Create stream socket
    - Link it to any port
    - Execute a connect
  - **Server:**
    - Create a listen socket
    - Link to a port
    - Wait for a connect request → queue
    - Execute an **accept** by creating a **stream socket** for that client and keep the other port free to listen to other

- There can be different **failure** also in TCP:
  - Possible block due to the buffer of the destination socket, control of the correctness, duplication and ordering, control of the message loss
  - The **Failure Model** to ensure these types of error is
    - Control of the correctness, duplication, ordering and control of the message loss

Chapter 7

Request Replay 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**.

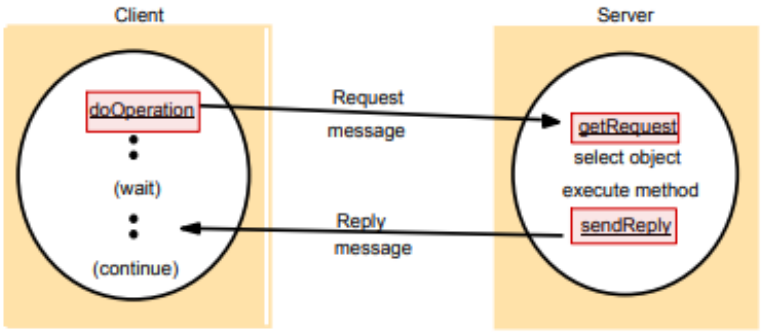


Figure 7.1: Request Reply Communication

7.1 Primitives

The request/reply protocol is composed of **three primitives**, the way in which these 3 functions are **implemented** correspond to the **choice** of a space semantic provided by the communication system.

- **doOperation:** used by the **client** to invoke remote operations and receiving the reply, specifying:
  - The remote server
  - The operation to be invoked
  - The arguments of that operation

  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

- **getRequest:** used by the server process to receive requests
- **sendReply:** method used by the **server** to reply to the client when it has finished to invoke the requested operation. Then the **client** which receives the reply from the server is unblocked

The information transmitted in a request or reply message is shown below:

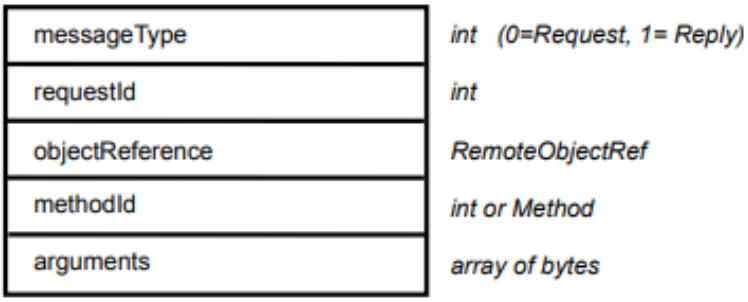


Figure 7.2: Transmitted Infrmation

- **messageType:** indicates the type of the message, if it is a *request* or a *reply* message
- **requestID:** is the message identifier. doOperation generates requestId for each message and the server remembers these identifiers
- **objectReference:** is a remote object
- **methodId:** is an identifier for the operation to be invoked

Notice that in **request-reply** communication each message has 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

Now we face the faults in the request-reply communication, in particular if the **primitives** are implemented over **UDP protocol**, they suffer from:

- Order of delivering not guaranteed
- Loss of message
- 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. The server has to have a **duplicate filtering strategy** thus 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
- The **history of transmitted reply messages** can be maintained by a table, but this could fill the memory
  - 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 needs 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

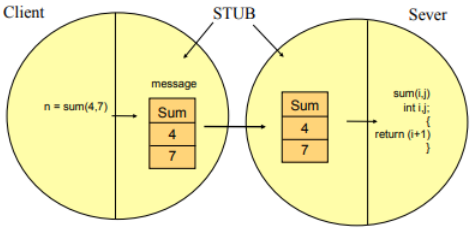
7.3 Remote Procedure Call

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. The following point list describe the the remote invocation mechanism:

1. The **Client calls the STUB** in normal state
2. Client STUB **builds** the **request** message, parameter marshalling, calling the OS to make the call
3. **OS sends** the message to the server
4. **Server** node **receives** it, and passes to the Server STUB
5. **Server STUB unmarshalling** of the parameters, **building the reply** message and passing th the OS for the reply
6. The message is **sent** to the client
7. The **Client STUB** receives the message
8. Client STUB **unmarshalling, result passing** to the client

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 RPC design issues

The RPC has three main design issues:

- **Parameter passing:**
  - *Call by value:* the parameter value is copied in the file, and if this is modified, this has no effect on the original variable
  - *Call by reference:* the reference to the variable is copied, the called procedure HAS access to the variable and the modifications directly affect the calling program
  - *Call by value-result:* the value of the variable is copied n the file as in the call by value, then it is copied back to it after the call, rewriting the original result
- **Binding:**
  1. *Static:* write the client program for a specific server on a specific machine
  2. *Dynamic:* locate the server at **run time**
- **Fault managements:**
  - *Communication:* the working processes can be not able to temporary communicate
  - *Node crash:* possible recover

7.3.2 Semantic in RPC

Before introducing the semantic we have to clarify that:

- **Normal Termination (NT):** when the client receives the message from the server
- **Abnormal termination (AT):** when the client **DOES NOT receive the reply**, so either it has not received any message or has received an unknown address (AU)

Semantics:

- **Maybe / exactly one:** 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
  - NT: the call has been executed once
  - AT: the call has **NOT** been executed once

  Difficult to implement
- **At least once:** 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
  - NT: the call has been executed once or more than once
  - AT: the call has **NOT** been executed once
- **At most once semantics:** it is achieved by retries masking any omission failures of the requests
  - NT: the call has been executed once
  - AT: the call has **NOT** been executed once or it has been **partially** executed or it has been executed **more than once**

  Relatively easy to implement

7.3.3 RPC fault management and semantics

The faults can be classified into two main categories:

- **Communication:** working processes can not be able to communicate temporarily
- **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 **several situation** in which the client *sends* the request and 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 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* the execution
- *Before* the execution
- *After* the execution

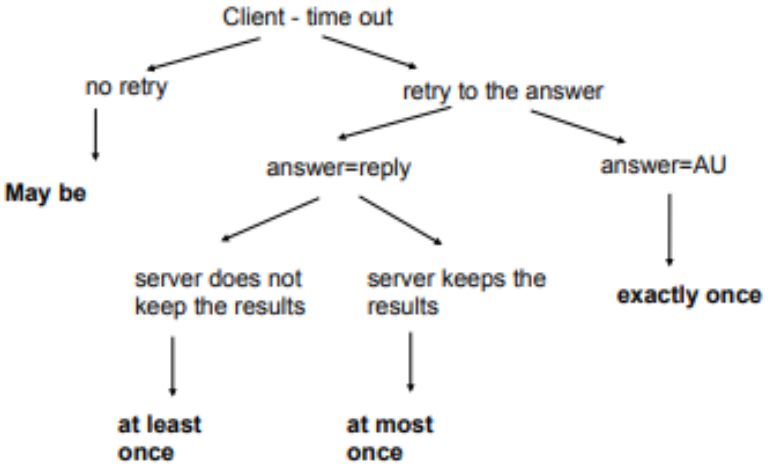


Figure 7.3: Fault Management and Semantics

7.3.4 RPC semantic and server crash

Crashing of servers 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 behaviour **"all or nothing"**:

1. 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 the presence of server crash provided 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 semantics. Summary of the two semantics:

- **At least one RPC:**
  - *Client:*
    - Send 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 one RPC:**
  - *Client:*
    - Send 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 it 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 a crash of the server this state is lost

7.3.5 RPC and client crash

- If a client is subject to a crash after sending a request that has been executed by the server, the execution is **called orphans**
- They consume resources (execution, memory, etc)
- The orphans create consistency problems and they interfere with the normal execution

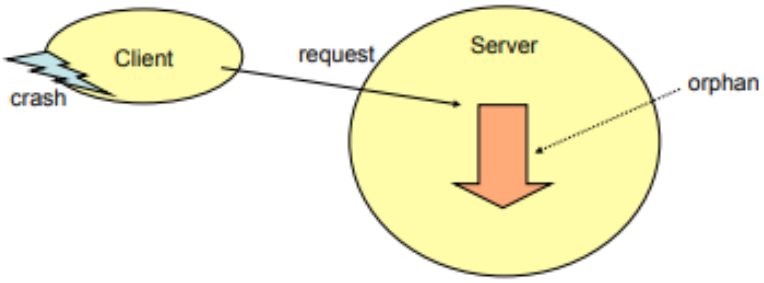


Figure 7.4: Client Crash

We can manage the **orphans** in two way:

- **Duplicate check:**
  - The **server** as soon as receives a new request **checks** whether it is already in execution from the same node a previous request from the same client
  - If so it should **terminate** or abort the **previous call** before executing the current one
- **Check of client vitality:**
  - A **server** should **periodically checks** whether the **clients** are *"alive"*
  - If one suspects a fault of the client, then one should **terminate** or abort the orphans

Chapter 8

Multicast Communication

Communication can be defined not only one-to-one, in fact there are other two possible cases:

- **Broadcast** → *one-to-all communication*: a process send the ,estate to all the other processes
- **Multicast** → *one-to-many communication*: a process send the message to a specific group of processes

Multicast communication is very frequent and there are different motivations:

- **Fault tolerance (server 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
- **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

Multicast brings with it some issues for which it is needed to develop a solving strategy, like the fact that if the sender sends a request to a group of servers, how does it has to deal with the answer? There can be different strategy, like:

- No wait
- Wait only for some answer, but how many
- Wait for all the answer

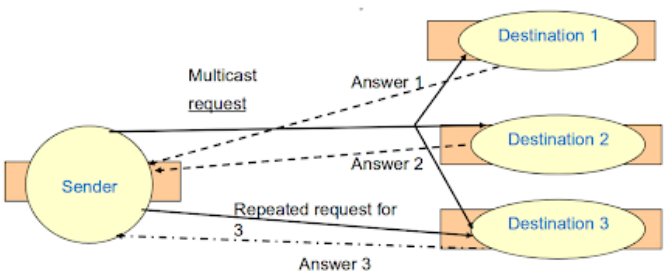


Figure 8.1: Example of Multicast Communication

8.1 Atomicity and Reliability Definitions

Multicast communication can also provide two mainly features:

- **Atomicity:** which guarantee that all the component inside the group will receive the message
- **Reliability:** which guaranteed the delivery:
  - *Validity:* messages are delivered
  - *Integrity:* messages that are delivered at most once are correct
  - *Agreement:* messages are delivered to all the destination processes

An **unreliable multicast** communication system sends only once the message.

8.2 Ordering

Another important issue from multicast protocol is **ordering**, messages are transmitted to a group in multicast and arrive to each component of the group according to the sending order. Taking this example:

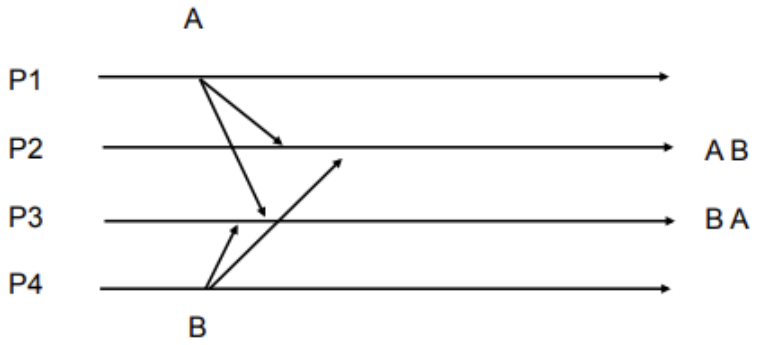


Figure 8.2: Ordering Problem in Multicast

A and B are two events sent by two different senders and A is sent before B. Ordering problem consists of developing a strategy that ensures that receivers receive the two messages with the same order. **Single Atomic Multicast** keeps 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 does not ensure the order, so we have to considered three solving methods:

- **FIFO ordering:** where it preserves the order from the perspective of a sender process. (the order is the same of the sender process)
- **Casual ordering:** where it considers a logical ordering, called also casual, and the message is delivered considering that constraint, so if an event A precedes an event B then the messages are delivered according to that order
- **Totally ordered multicast:** if it is atomic, it delivers the message to each member of the group in the same order

8.3 Implementations

Multicast can be supported by different types of implementations of the two primitives: *send* and *receive*. The way the two are implemented define the multicast system. Next we 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 8.3: Example of Multicast Implementation

For a reliable multicast is necessary to check the following assets:

- Possible message loss
- Crashing of sender
- Wait the answer
- Possible retransmission in case of time out

The implementation of the fault management system have to take into account the occurrence of faults like:

- Message omission
- Sender crash

A possible solution is to use a **monitor** with the following rules:

- Check of 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 executed

8.4 Reliability and Atomicity in practise

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 is 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 waits 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 keeps 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.

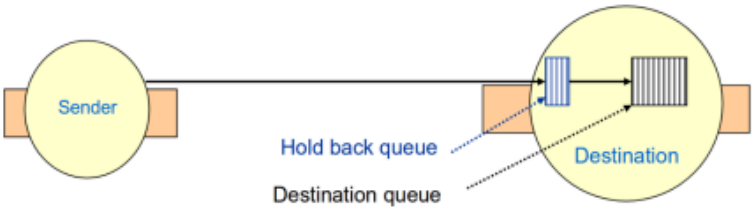


Figure 8.4: Hold'back Queue

Another solution consists to the usage of **negative ack:**

- The messages are numbered according to the order from the server
- 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**
- The sender has a **copy** (history) of the **sent messages**

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 messages using piggybacking technique.

8.5 Atomic and totally order 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 identifiers cannot arrive.

This solution is easy to implement if sender and receivers agree with **a shared identifier sequence** used to assign identifiers 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:

- **Centraized:** unique manager that makes the ordering. But the drawback is that it can become a bottleneck since all the request should pass from him

- **Distributed:** coordination of the receivers that makes an agreement on the ordering

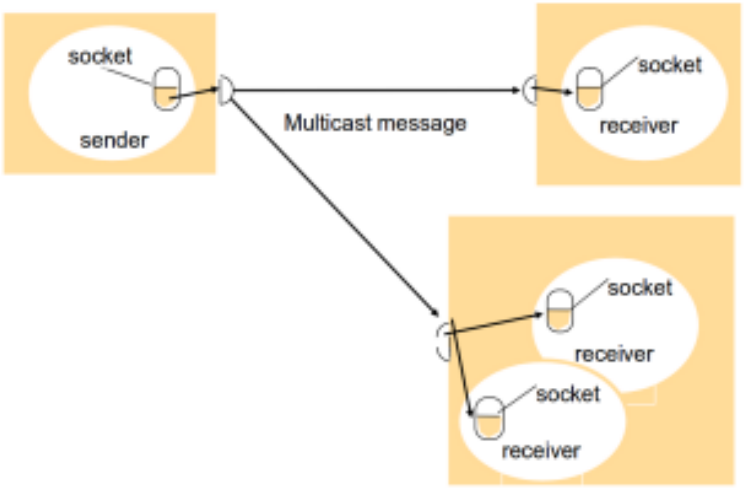


Figure 8.5: Atomic and Totally ordered Multicast

Chapter 9

Types of RMI system

- **Common Object Broker Architecture (CORBA):**
  - Allows the communication among applications written in **various languages** through the use of a model of neutral object
  - Allows the communication among existing applications developed in **heterogeneous environments** and for which one cannot make a conversion in a common language
- **Java RMI:**
  - Allows the communication among **applications written in Java**
  - **Environment heterogeneity** as a consequence of the Java language portability. It leads to a simpler code

9.1 Distributed object model features

- The clients of remote objects interact through remote interface and must manage the exception for possible failure of RMI
- The arguments and the results of the remote methods are passed by value and not by reference, by using the concept of serialisation
- A remote object is always passed by reference
- There are some security mechanisms introduced to check the behaviour of the classes and the references

9.2 Garbage collection of remote objects

- Every JVM updates a series of counters, each of them associated to a given object
- Each counter represents the reference number to a given object that is currently active on a JVM
- Every time a reference to a remote object is created, the corresponding counter is incremented

- When no client has reference to an object, the runtime of RMI uses a "weak reference" to address it
- The weak reference is used by the garbage collector to cancel the object as soon as also the local references are not present

9.3 Serialisation

- Mechanism used for the data transmission between client and server in RMI
- It is an automatic transformation of objects and structures of objects in simple byte sequences, to be sent in a data stream
- There can be serialisation only for instances of serializable object
- It is not transferred the real object, but only the information that characterise the instance of it
- At de-serialization time a copy will be created of the transmitted instance by using the .class

Chapter 10

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.

10.1 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.** The OS running at a node provides its own flavour of abstractions of local hardware resources for processing, storage and communication.

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.

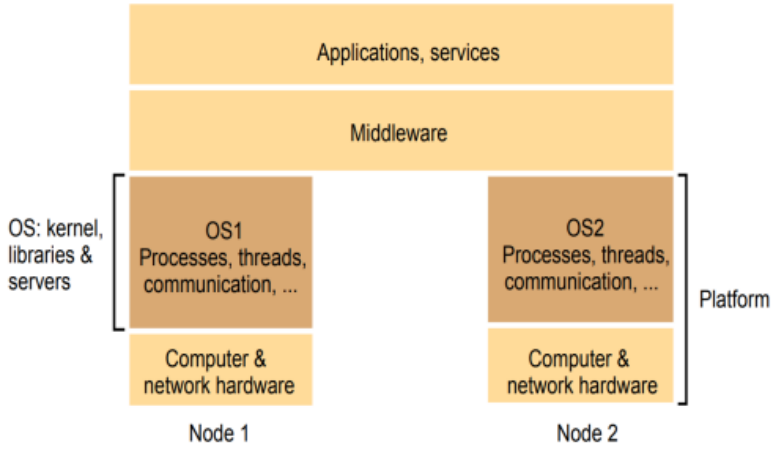


Figure 10.1: Operating System Layer

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.

The core OS components and their responsibilities are:

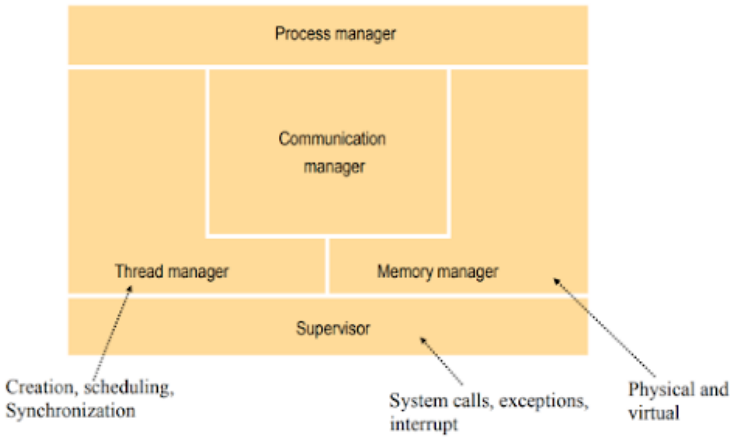


Figure 10.2: Core OS Functionality

- **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, synchronisation 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

10.2 Process and Threads

- A **preocess** consists of an execution enviroment 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 **maximise the degree of concurrent execution between operations**, thus enabling the overlap of computation with input and output, and enabling concurrent processing on multiprocessors.

10.2.1 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.**

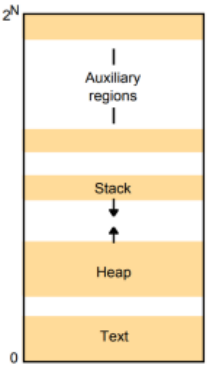


Figure 10.3: Address Space

Each region is specified by the following properties:

- Its dimension: lowest virtual address and size
- Permissions for the process's threads: read/write/execute
- Direction of 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
- **Data sharing and communication:** two processes, or a process and the kernel, might need to share data in order to cooperate on some task

10.2.2 Creation of a New Process

For a distributed system, the design of the process-creation mechanism has to take into account the **utilisation 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 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

- The **creation of an execution environment:** once the host computer has been selected, a new process requires an execution environment consisting of an address space with initialised contents. There are **two approaches:**

  1. The first 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
  2. In the second approach, the address space can be defined with respect to an existing execution environment

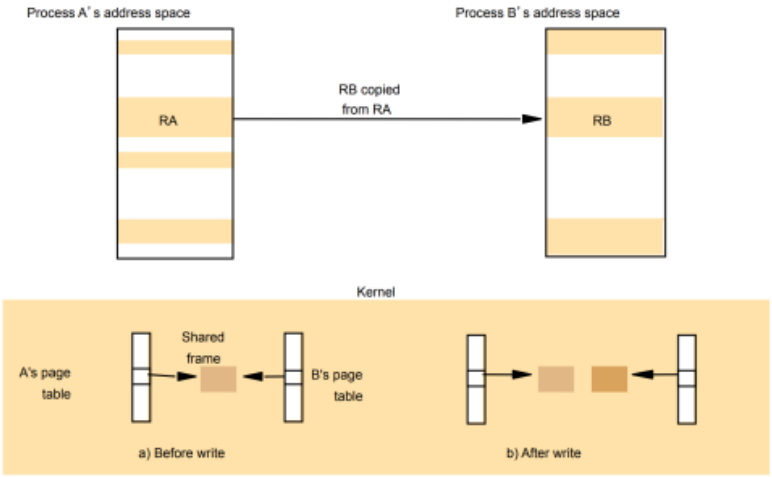


Figure 10.4: Creation of an Execution Environment

10.2.3 Threads

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.

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

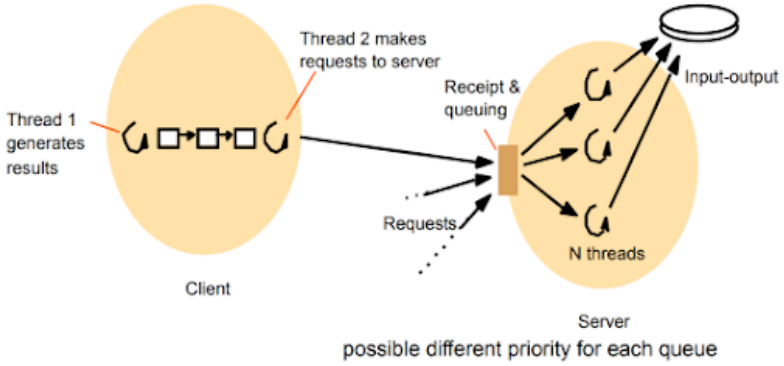


Figure 10.5: Thread Architecture

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.

Disadvantages:

- Its **inflexibility**, the number of worker threads in the pool may be too few to deal adequately with the current rate of request arrival
- 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.
- **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

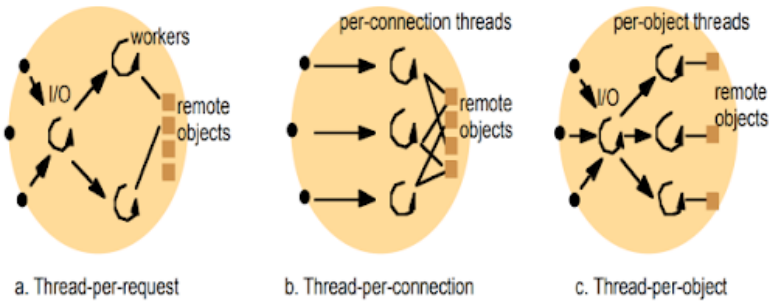


Figure 10.6: Types of Thread Architecture

The JAVA thread's life cycle can be seen in the below image

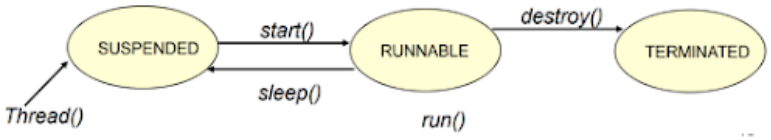


Figure 10.7: Java Thread's Lifecycle

10.3 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 Using thread at application level we have some advantages like:

- Less cost to context change
- Possible to implement scheduling ad hoc
- 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. Possible approaches:

- Enable user-level code to provide scheduling hints to the kernel's thread scheduler it is called **hybrid solution**
- Form of **hierarchical scheduling**

10.4 Communication: invocation and call

Communication is part of invocation and it can be implemented using different approches like:

1. RPC
2. RMI
3. Message Passing
4. Event Notification
5. System call
6. Group Notification

An important aspect of communication is the **performance provided by the mechanism used**. Fundamental aspects that can affect the performance are:

1. **Type of communication** that can be *synchronous* or **asynchronous**
2. **Address space to cross**, it can be *local* or *remote*
3. **Scheduling thread** that affect context change

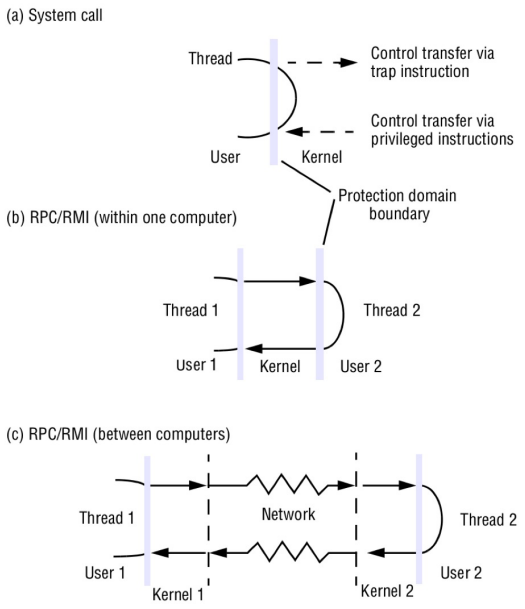


Figure 10.8: Topology of Address Space

10.4.1 Lightweight Remote Procedure Call

The previous image suggest that a *cross-address-space* invocation is implemented within a computer exactly as it is between computers. Indeed Bershad developed a more efficient invocation mechanism for the case of two processes on the same machine called **lightweight RPC, LRPC**.

The LRPC design is based on optimization 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 LRPP, the **arguments are copied once:** when they are marshalled onto the A stack. There my be several A stacks in a shared region, because several threads in the same client may call the server at the same time.

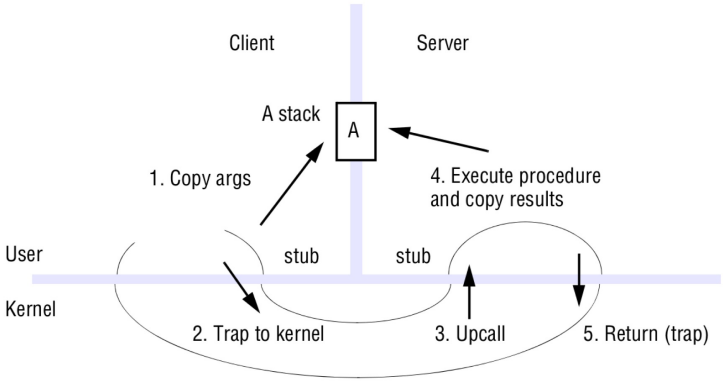


Figure 10.9: LRPC

- A client thread enters the server's execution environment by first tapping to the kernel and presenting it with a capability
- The kernel check 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.

10.4.2 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 explain who they work.

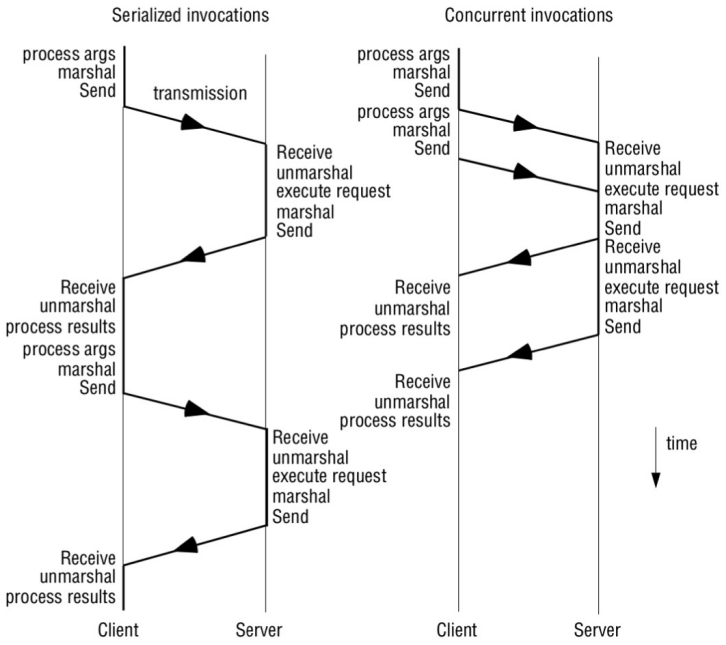


Figure 10.10: Serial and Concurrent Invocation

With **concurrent invocation** client allocate multiple threads to perform blocking invocations concurrently, an example is browser web. In serialized case, the client marshal the arguments, call the *Send* operation and the waits the results. After this it can make the second invocation. In the concurrent case, the first client thread marsh the arguments and calls the *Send* operation. The second thread then immediately makes the second invocation. Each thread waits to receive its results.

10.5 Organization and distributed operating system architecture

Now we analyse 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
- 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
- Introduce new services without harming the integrity of existing ones

The organization of the operating system can be:

- **Monolithic**
- **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.

10.5.1 Monolithic

It perform all basic operating system functions. It is an operating system architecture where the entire set of functionalities is working in kernel space. The advantage is that it is relatively efficient.

10.5.2 Microkernel

The kernel provides only the most basic abstractions, principally address spaces, threads and local interprocess communication. 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.

10.6 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 maximise the system resources utilization.

Under certain condition the theory provides an optimal algorithms for process schedulting. In **multiprocess system** also we have a unique process queue, but now the system is composed by multiple processors.

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.

Scheduling in distributed system introduce a new target to optimize which is **load balancing**. Load balancing consist on developing a solution / strategy that move processes between machines in order to assign the same load to the different machines.

Process allocation could be classified in this way:

- **Static / Dynamic** if allocation decision are prefixed or not
- **Deterministic / Non-deterministic** if next process requests are known
- **Centralized / Distributed / Hierarchical**
- **Optimal / Approximated**
- **With / Without Migration**

10.7 Scheduling algorithms for distributed systems

We can have different scheduling algorithms:

10.7.1 Probes algorithm

When the processor executing the process becomes overloaded it takes the decision to transfer the process, **probes** another processor the decide whether to transfer the process. If the contacted processor is **underloaded** the transfer takes places, otherwise it selects randomly another processor to repeat the attempt up to a maximum number of times.

10.7.2 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, the traffic information and deal with change of variation of it.

10.7.3 Centralized algorithm

It is centralized algorithm to **distribute the load** between workstations for load balancing. Each workstation has a **score** and if it sends information to other process the score increase, if it serves others the score decreases.

10.7.4 Bidding algorithm

It is an extension of centralized algorithm. It is based on the **metaphor of an economic system**, where CPU time is associated to a price. 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.

10.7.5 Hierarchical algorithm

The hierarchical algorithm **improves the 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.

10.7.6 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 process to be executed at time i on processor j.

Chapter 11

Synchronization and coordination in distributed systems

One of the main goal of DS is that for a set of process to synchronize and to coordinate their actions or to agree on one or more values.

Before introducing the algorithms we have to make an important distinction on synchronization:

- **Internal:** agreement of a set of processes
- **External:** forced by an external agent

In distributed system there are several problem to analyse:

- Clock Synchronization
- Mutual exclusion
- Election algorithm
- Atomic transaction
- Deadlock management

11.1 Mutual Exclusion

It 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 characterised 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
- **Ordering:** entering a critical section occurs some accord to the casual ordering

11.2 Central Server Algorithm

It has a simple implementation, **it simulates the primitives offered by the centralized server** through a coordinator process. Its behaviour is structured as follow:

1. A process asks the coordinator to enter and it is possibly blocked
2. The answer is an access **token**
3. At receiving time of the answer, it enters the **critical section**
4. At the **exit time** from the critical section it communicates to the coordinator

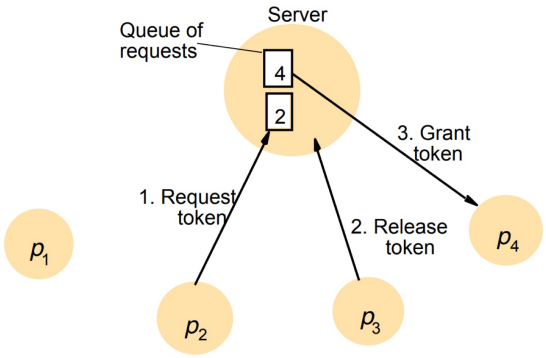


Figure 11.1: Server Managing a mutual exclusion token for a set of processes: Centralized Algorithm

11.3 Distributed Algorithm

DA works under the assumption that there exists a **global ordering of events**, normally each algorithm is based like so:

- Protocol to **enter / exit to the critical section**:

that one which want to acces to the critical section

  - A process that wants to **enter a critical section** sends a message to all the other process, 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 critical section and it does not want to enter** → sends **OK to the sender**
  - **Be in the critical section** → it does not answer and put **message in a local queue**

– **Wants to enter in the critical section** → compares the **timestamp** and the **oldest has higher priority**, if it is the oldest process, it sends the **OK**, otherwise, if it is itself, it put **the message in the local queue**

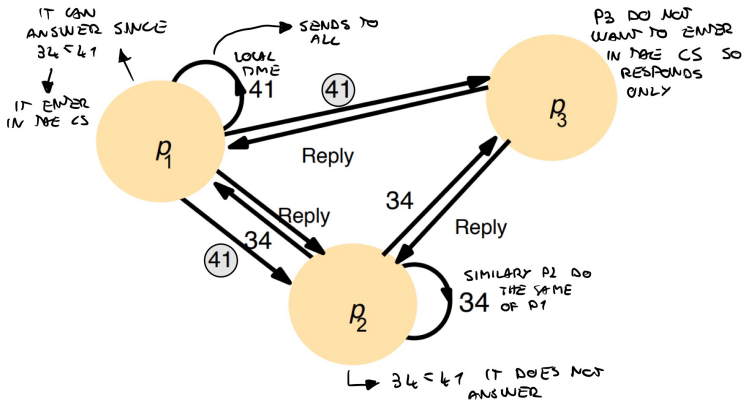


Figure 11.2: Multicast Synchronization

This algorithm satisfies all the three mutual exclusion criteria, but it has some **drawbacks**:

- **High traffic** generated by N processes that require $2(N-1)$ messages for the multicast request and the answer
- **Fault of the system** if one of the processes crashed
- **Possible bottleneck** introduced by processes

11.4 Ring Algorithm

Processes share a **token** using a **local ring** structure.

1. The first process has a **token** that uses and then forwards to the next one.
2. The process that **has the token** is enable to **access to the critical section**.

This algorithm satisfies the *Safety* and *liveness* conditions but not the *ordering* one. It has the following features:

- **Costs:** $[1, N-1]$ messages to obtain the token. 1 message to exit fro the critical section and $[1, N-1]$ message for synchronization
- **Reliability:**
  - If a process faults it is necessary to **rebuild the logical ring**
  - If a process having the token faults it is necessary to make an **election** of the next process that will have the token
  - Loss of the token also for hardware/software faults
- **Performance:** the algorithm always **requires bandwidth to transmit the token** even if none ask for the critical section

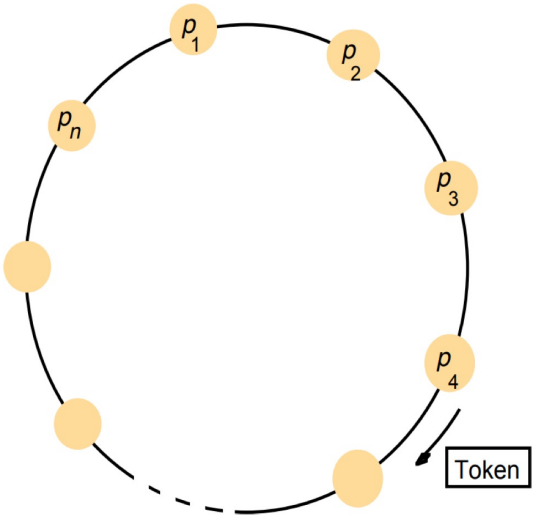


Figure 11.3: A ring of process transferring a mutual exclusion token

11.5 Voting Algorithm

In order to enter in the critical section, it is necessary to **synchronize only a subset of the interested processes.** So the processes make a vote (*election*) to decide which process can access to the critical section:

- Vote set V_i is a subset of $p_1, ..., p_N$, associated to each process p_i
- A process that wants to **enter in the critical section** sends a message to all the other processes in V_i
- It **waits** for all the **replay**
- Once all the **OK** have been received it enters the critical section
- At the **exit time** from the critical section, it sends a message **release** to all the others members of V_i
- A process p_j in V_i that receives the request $\rightarrow$ if its states is **HELD** or it **already answered** after having received the last message **release**, it does not 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 **replay**

11.6 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

11.7 Election Algorithm

The main goal of this algorithm is to let know all the processes on a specific **elected process** that act as **coordinator** in the distributed system. The following algorithm make three assumptions:

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

Normally, a process calls an election and at the end of the algorithm, all the process agrees **on the same elected process** The general structure of these algorithms is the following:

- A process calls an election;
- Each process can be participant or not to the election;
- The elected process is unique even if more than one process called election.

11.7.1 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. It works as follow:

- 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 the election message ID is **the same**, then is a **coordinator**, it becomes **non participant**, and it sends an **elected** message
- Every other processes that receives an **elected** message are marked as non participant

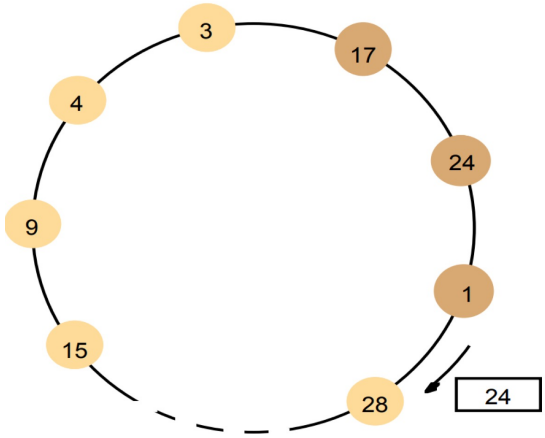


Figure 11.4: A Ring-Based Election in process

11.8 The Bully Algorithm

In this algorithm each process communicate with process having **higher ID**. The message that are shared a are: *election*, *answer* and *coordinator*. It works as follow:

- A process starts the election by sending an **election message** to all the process with **higher ID**, and waits for the replies
- If the **reply** does not arrive **within a timeout** a coordinator is nominated and communicate the news to the other processes
- 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 number and consider that process as **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.

A process **starts an election** in the following cases:

- When a process is reactivated after a faults
- 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 cost performance:

- **Worst case:** $O(n^2)$ messages
- **Best case:** $O(n-1)$ messages

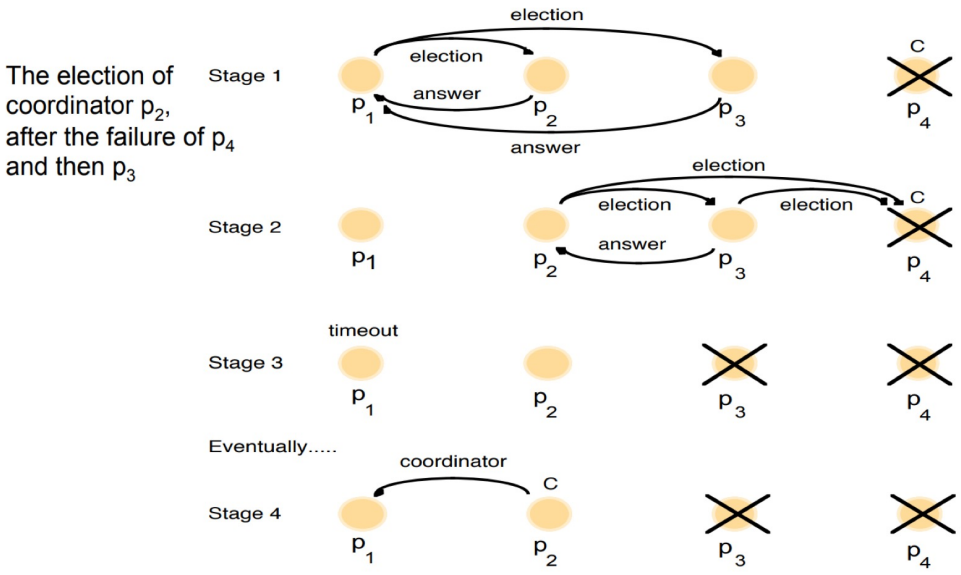


Figure 11.5: Example of execution of Bully Algorithm

11.9 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
- **Circular waiting:** there exists a closed path in the allocation graph

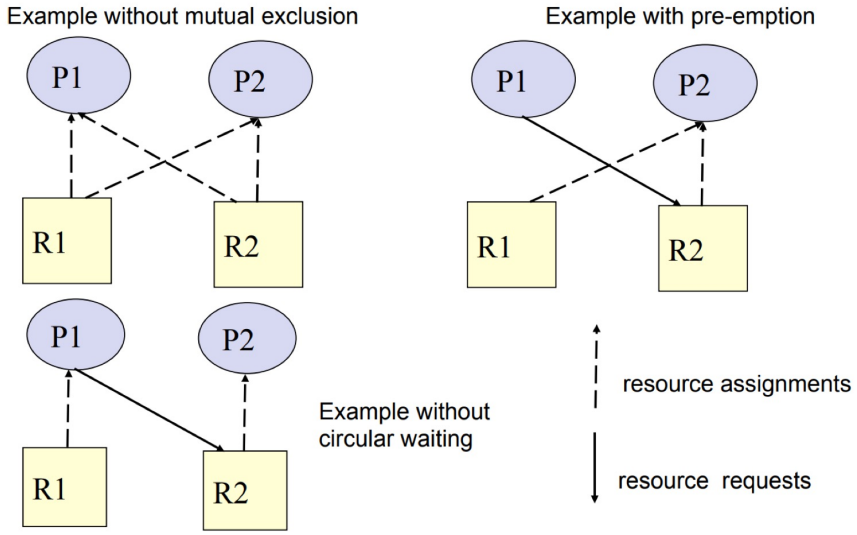


Figure 11.6: Deadlock Conditions

There are **different types** of deadlock:

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

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

- **Prevention** applied by:
  - Allowing only one resource allocation
  - Pre-allocate resources
  - Ordering
  - Age rule
- **Avoidance:** it determines the stable states on the basis of the process needs
- **Ignoring:** applied to terminate process in cyclic waiting
- **Detection:** by applying *Chandy-Misra-Haas* algorithm:
  1. A blocked process P_1 starts a test (structure composed of
     $$ID_{blockedprocess}, ID_{sendingtestprocess}, ID_{receivingtestprocess}$$
  2. P_2 receives the test and if it does not need other resources it stops the test. Otherwise if it is blocked by P_3 it forward the test
  3. If a process receives a test with $ID_{blockedprocess} = ID_{receivingtestprocess}$ it detects the deadlock

Chapter 12

Coordination and synchronization: clock

Time is an important and interesting issue in distributed systems, for several reasons.

- 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

In distributed systems there is 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.

12.1 Model definition

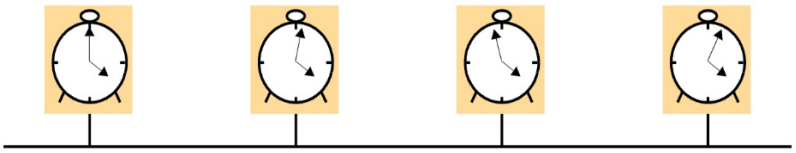


Figure 12.1: Clock

There are N processes $p_1, p_2, ..., p_N$ connected to a network. Each process has a **local clock**, there is **no shared memory** and they can communicate only using **message passing**.

The **ordering** is defined as follow:

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

$$h =< e_i^0, e_1^1, ... > \quad \textbf{state history of process i}$$

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

12.2 Physical clock

Synchronization algorithms based on **physical clocks** operate using local clock of each process.

C_i 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.

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

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

  - $S(t)$ is the real time (provided by UTC)
  - D is the precision
  - I is the interval of synchronization

- **Internal:** if there is an agreement among a set of processes.

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

- A physical clock H is correct if the drift (difference between the two clock) is within a given threshold $p > 0$. Given t, t' if $t' > t$ then

$$(1-p)(t'-t) \leq H(t') - H(t) \leq (1+p)(t'-t)$$

- A clock C is defined **monotone** if it only goes on forward

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

$$C_i(t) = aH_i(t) + b \quad a, bconstants$$

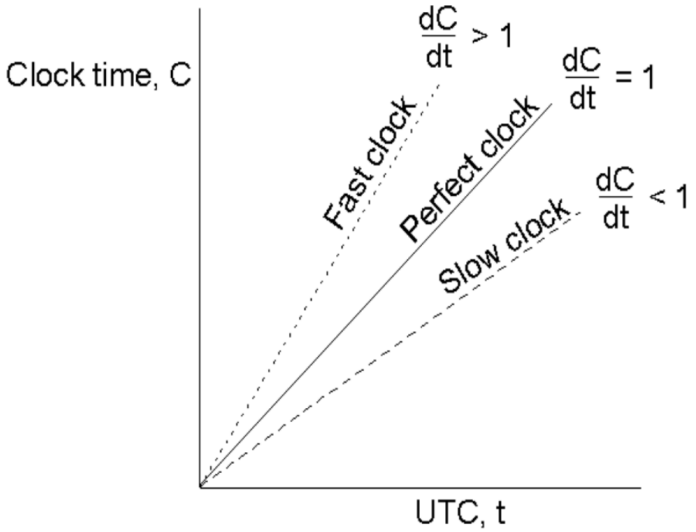


Figure 12.2: Clock Deviance from real time

12.2.1 Distributed synchronous system

We begin by considering the simplest possible case: **synchronization between processes in a synchronous system**. 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.

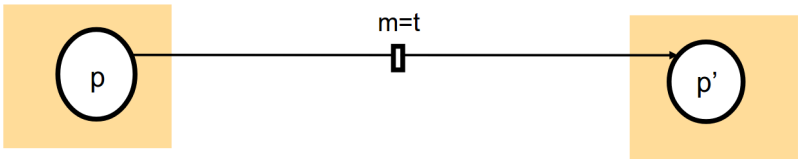


Figure 12.3: Clock synchronization: Sync. System

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_0 receives the message and sets its own local clock to $t + T_{com}$ (message transmission time).

- There is always a minimum transmission time, T_{min}.
- In a synchronous system, by definition, there is also an upper bound max T_{max}

From these results some possible choice can be discussed to determine the clock of p':

- If p' sets the clock to $t+T_{min} or t+T_{max}$ the error would be $\leq \delta = (T_{max} - T_{min})$
- If p' sets the clock to $t + (T_{min} + T_{max})/2$ the error would be $\leq \delta/2$
- The optimum bound that can be achieved on clock skew when synchronizing N clocks is $\delta(1 - 1/N)$

12.2.2 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, which suggested the usage of a **time server**, connected to a device that receives signals from a source of **UTC**, to synchronize computers externally.

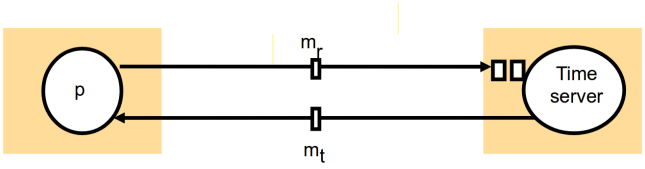


Figure 12.4: Cristian Algorithm

Machine point of view: 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},
- It sets the clock to $t + T_{round}/2$

Time server point of view: 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 $\pm(T_{round}/2 - T_{min})$

- **Advantage:** works well when synchronization delay is close to zero
- **Drawbacks:** The usage of a central server reduces and limits reliability and performance
- **Improvements:** using multicast instead of single and central server,, or introduce authentication

12.2.3 Berkeley

Berkeley uses a coordinator computer as the master, also called **active time server**.

- Unlike in Cristian's protocol, this server periodically questions the other computers whose clocks are to be synchronized, called **slaves**.
- The slaves send back their clock values to it.
- he master estimates their local clock times by observing the round-trip times (similarly to Cristian's technique), and it averages the values obtained

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 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**, thus the master takes a **fault tolerant average.** 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.

12.2.4 Network Time Protocol (NTP)

The last two cited algorithm are based on the usage of *centralized system*, now we see an approach based on **distributed algorithms.**

The **Network Time Protocol** 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
- **Reliable service** that is fault tolerant to connection loss. **Fault tolerance** is given by redundancy of server and path
- **Scalability** allows frequent synchronization
- To provide protection against interference with the time service, whether malicious or accidental

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
- **Secondary servers** are synchronized with primary servers
- The servers are connected in a logical hierarchy called a **synchronization subnet**, whose levels are called **strata**

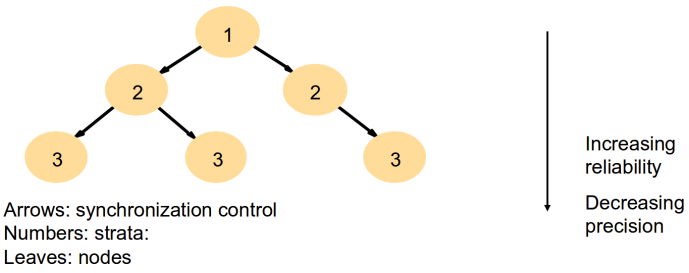


Figure 12.5: NTP subnet

The greater is the number of servers adopted the greater is the reliability provided. **Server synchronization** can be done by:

- **Multicast:** common with *high speed LAN*
- **Procedure call:** the server is passive and waits for requests like Cristian's algorithm
- **Symmetrical:** servers exchange messages with timestamp

12.3 Logical Clock

In distribute systems since we cannot synchronize clocks perfectly across a distributed system. A different solution is given by **logical clock**, but first is necessary to give some notions of **casual ordering**.

12.3.1 Casual Ordering

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)*:

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

Casual ordering defines order between events and it has some proprieties:

- If $\exists p_i : e \rightarrow_i e'$ then $e \rightarrow e'$
- $\forall$ message m then $send(m) \rightarrow receive(m)$
- If e, e', e'' are events:
$$e \rightarrow e', e' \rightarrow e'' \quad e \rightarrow e''$$
- Even a and b can be **concurrent** and we denote this propriety as $a||b$. Meaning that they are **not related**

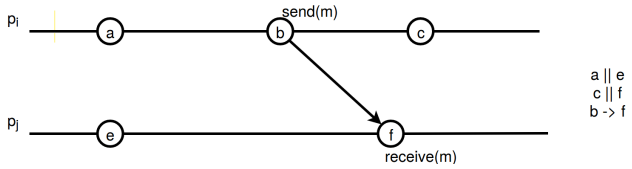


Figure 12.6: Casual Ordering Example

12.3.2 Lamport timestamp

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 the so called **Lamport timestamp** to events. Moreover 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.
- **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)*.

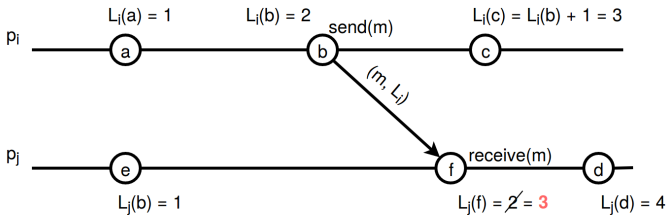


Figure 12.7: Logical clock algorithm example

Logical clock has the monotonicity property, meaning that:

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

12.3.3 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

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

A different ordering that overcomes the iff limitation of the Lamport definition, $L(e) < L(e')$note $e < e'$, 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 12.8: Vector Clock Algorithm

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

Then:

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

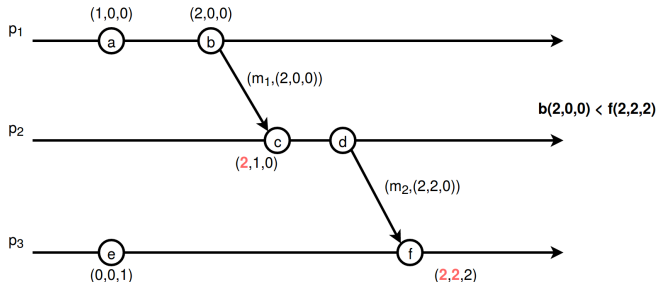


Figure 12.9: Vector Clock algorithm example

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

12.4 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.

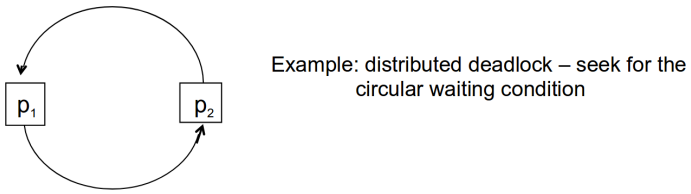


Figure 12.10: Global state deadlock

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. 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: $h_i =< e_i^0, e_i^1, ...e_i^k >$
- 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 set of processes $\{p_1, p_2, ..., p_N\}$

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

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

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**.

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

12.4.1 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*:

$$\forall e \in C \quad e' \rightarrow e \quad \rightarrow \quad e' \in C$$

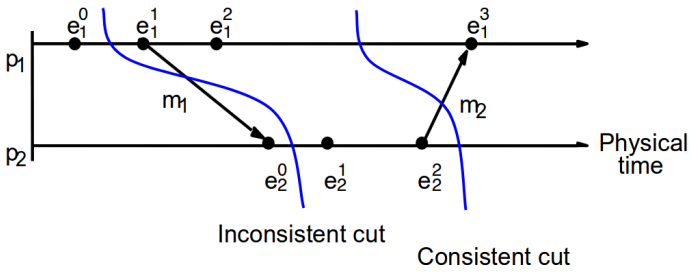


Figure 12.11: Consistent and Inconsistent state

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 $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow ....$ A **consistent run** is an ordering of the events in a global history that is consistent with this happened-before relation $o \rightarrow \text{on} H$.

12.4.2 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 $p_i (i = 1, 2, ..., N)$. The algorithm records state locally at processes. An obvious method for gathering the state is for all processes to send the state they recorded to a designated collector process.

The algorithm assumes that:

- Reliable channel and communication.
- Channel is unidirectional and provide FCFS-ordered message delivery.
- The graph of processes and channels is strongly connected
- 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.

Chapter 13

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** it is an operation sequence on the server, involving a set of processes and/or shared resources, that are guaranteed to be **atomic** in presence of **concurrency** and **faults**. The three methods for **concurrency control** that we will analyse are:
  - Lock
  - Optimistic
  - Timestamp
- The **goal** 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

Propriety of the transaction:

- **Isolation**, there is no interference with the other transactions, partial effects have not to be visible
- **All or nothing**, all the operations are executed successfully or if just one is not completed, one has to reconstruct the initial state

And moreover also **ACID**:

- **Atomicity**: a transaction must be *all or nothing*
- **Consistency**: a transaction takes the system from one *consistent state* to another *consistent state*
- **Isolation**: each transaction must be performed *without interference* from other transaction
- **Durability**: after a transaction has *completed successfully*, all its effects are saved in *permanent storage*

In general transactions can be part of the middleware.

13.1 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**, thus 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.

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, so more technically **if they are serially equivalent** or **serializable**.
- In other words **transactions are serializable if the serial and concurrent run are equivalent.**
- So it is necessary to introduce a new entity called **coordinator** that manages concurrency.

The coordinator gives each **transaction an identifier** or **TID**. The possible callable methods are:

- *Client* invokes ***openTransaction*** of the coordinator to start a new transaction (a transaction identifier TID is allocated and returned).
- *Client* invokes ***closeTransaction*** to indicate the end (all of the recoverable objects accessed by the transaction should be saved)
- *Client* invokes ***abortTransaction*** if for some reason it wants to abort it

A transaction is achieved by cooperation between a client program, some recoverable objects and a coordinator.

- A transaction is achieved by cooperation between a client program, some recoverable objects and a coordinator
- 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.
- **Inconsistent Retrieval:** data values are not consistent for all the copies.

13.2 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 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.

13.3 Conflict of Operations

Serial equivalence is not satisfied in presence of **conflict operations**. The operations are in **conflict** if the result of the **combined execution** depends on the **execution order**

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 13.1: Conflict Rules

*Two transactions are serially equivalent **if and only if** 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 13.2: Non-serially equivalence interleaving operations

We can consider a lot of different type of fault:

- **Dirty reads from previous abort:** 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).
- **Premature write:** in which some wrong effects to the results are given by the aborting operation of a transaction or by multiple write operations to the same object
- **Domino effect:** where the execution of a transaction with abort can cause other abort

And how can we solve all this stuff?

- Imposing that the **read** of objects happens only if the previous **write** on that object have been executed by **completed transactions**
- Delaying the execution of **read**

The resolution of this problem results fundamental in order to be able of recovering the previous state. A good implementation consists on recovering the previous state using of **before images** for all the write operations in each transaction. Thus a solution is:

- 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**
- This last one is called the **strict execution**

13.4 Nested transaction

More complicated transactions, *top-level*, can be composed by other different transactions, *sub-transactions*. **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.

The **advantages** of splitting a transaction in multiple one are: *increasing of performance* and *fault-tolerance*

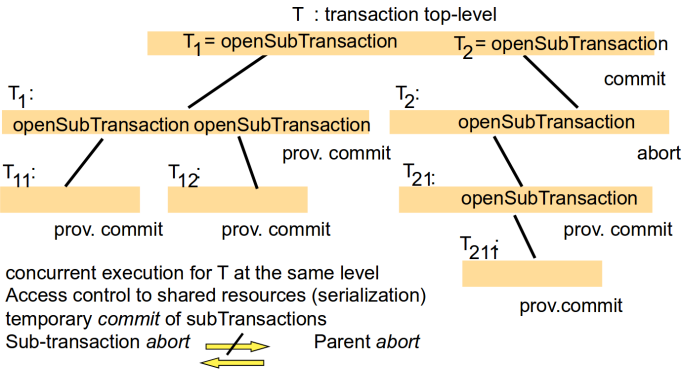


Figure 13.3: Nested transactions

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**.

13.5 Concurrency control: Lock

The most common solution for serializing access to the objects from a server consist on **locking** the access to the shared objects. The idea of **lock** is to to provide access with **mutual exclusion**. All the pairs of operations of two transactions in conflict have to be executed with the same order. 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 13.4: 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**.

13.5.1 Strict two-phase locking

Algorithm that provides concurrency control using **lock strategy**.

When an operation accesses an object within a transaction:

- Object **not locked** → it is **locked** and the operation proceeds
- Object has **conflicting lock** → transaction waits until it is **unlocked**
- Object has a **non-conflicting lock** in the same transaction → the **lock is shared** and the operation proceeds
- Object has **already been locked** in the same transaction → the **lock promoted** and the operation proceeds. Where promotion is prevented by a conflicting lock rule 2 is applied.

For **nested transaction** it is necessary that:

- Each set of nested T **does not see** partial results of the **other sets** of nested T
- Each T **in the set** of nested T **must not see** partial results of **other** T **in the set itself**

At the higher level a T inherits all the acquired lock in the nesting sub tree so avoiding partial view not completed. 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:

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

13.5.2 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.

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 13.5: Lock deadlock

A **wait-for graph** can be used to represent the *waiting relationships* between current transactions. In a wait-for graph:

- Nodes → transactions
- Edges → wait-for relationship 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.**

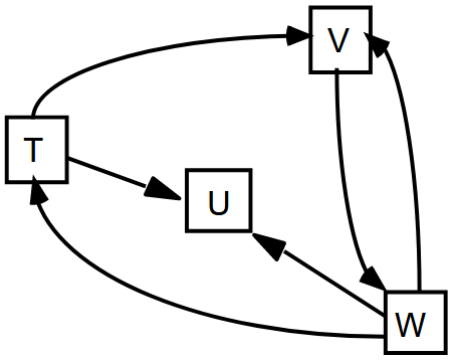


Figure 13.6: Conflict Rules

Now we present two possible **solutions** to overcome the problem of **deadlock**:

- **Lock timeouts:** each lock is given a *limited period* of time 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. The transaction whose lock has been broken is normally aborted. **Drawbacks:**
  - Overhead due to waste of time
  - Transaction abort also without actual deadlock

- **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.

13.6 Lock compatibility (read, write and commit locks)

Even when locking rules are based on the conflicts between read and write operations and the level of details 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:** allows one transaction to **write** tentative versions of objects while other transactions **read** from the committed versions of the same objects. This scheme allows more concurrency than read-write locks. **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 13.7: Compatibility table for two-version locking

  - 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:** implement the usage of a hierarchy of locks. At each level, the setting of a parent lock has the same effect as setting all the equivalent child locks. In Gray's scheme, each node in the hierarchy **can be locked**, giving the owner of the lock access to the node and to its children.

13.7 Concurrency control: Optimistic

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

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 13.8: compatibility table for hierarchic locks

- 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.

A different solution is propose by **optimistic concurrency control**, which assume that there are no conflicts for operations. So the 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. During commit time server performs a check and whether there have been interferences and in that case it removes the transactions.

Transactions pass from these phases:

- **Work phase** each transaction has a copy 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. In this p
  - **Validation successful:** transaction can commit
  - **Validation fails:** some form of conflict resolution must be used, and the current transaction or those with which it conflicts will need to be aborted
- **Update phase:** if a transaction is validated, all of the changes recorded in its tentative versions are made permanent.

13.7.1 Serializability

In the validation phase only a transaction per time can enter, moreover the transactions that enter are **numbered in increasing order.** In order to provide validation of the transaction serializability the following rules are considered:

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.

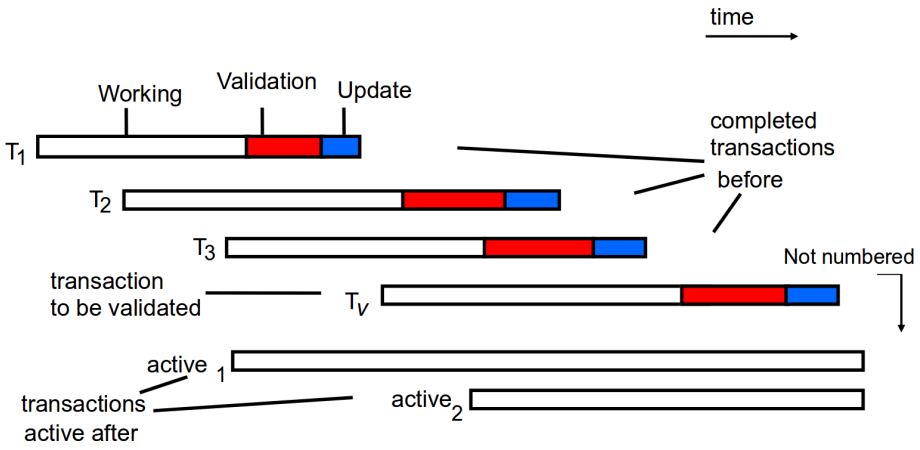


Figure 13.9: Optimistic serializability validation

There two possible **strategy to check overlapping**:

- **Backward**: *checks* with the *previous* transactions already evaluated.
- **Forward**: *compare* with *future* transactions not already evaluated.

13.8 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 **couple of timestamp**.
- The timestamp defines its position in the time sequence of transactions.
- Requests from transactions can be totally ordered according to their timestamps.
- The couple of timestamp is composed by **read timestamp** and **write timestamp** that registers the time of the last transaction that accessed it.
- The object **authorizes** the operation only to transactions with **timestamp greater** of the object timestamp.
- 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.

Chapter 14

Distributed Transactions

A transaction becomes **distributed** if it invokes operations in *several different servers*. 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.

We can identify two different type of transactions:

- **Flat transactions:** are composed by a set of operations that are executed sequentially
- **Nested transactions:** are composed by subtransactions, which can run concurrently if they are at the same level

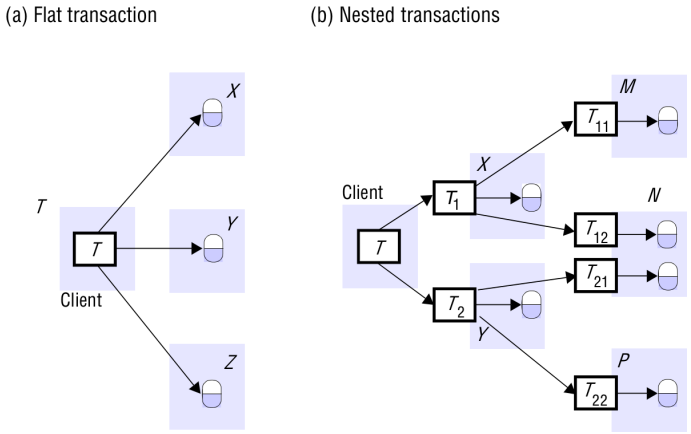


Figure 14.1: Distributed transactions

- In the flat transaction it invokes operations sequentially on objects in servers X, Y and Z
- In the nested transaction, there are other two subtransactions T_1 and T_2 that open further transactions T_{11}, T_{12}, T_{21} and T_{22} which access objects at servers M, N and P. The subtransactions can run concurrently so, T_1 and T_2 are concurrent like T_{11}, T_{12}, T_{21} and T_{22}.

14.0.1 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**. It returns the resulting **transaction identifier** (TID), for example the pair (server ID, # of local transaction). Coordinator at the end is responsible for committing or aborting it, moreover 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 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.

14.1 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**.

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
  - It repeats the operation up to completion
  - But in this way it can happen that **atomicity is not satisfied**
  - 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**, is the most common and used algorithm, here all the participants can do abort. It is composed by two distinct 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 replyes with **Yes** or **No**
     - Before voting *Yes* it prepares to commit by saving objects in permanent storage.

- Otherwise it responds *No*, the participant aborts

2. **Decision phase:**
   - 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.

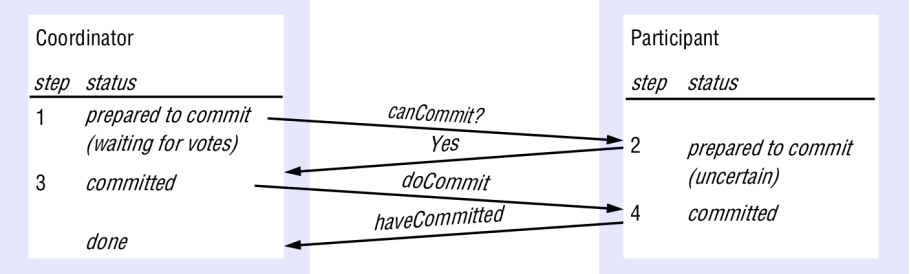


Figure 14.2: Communication in two phase commit protocol

14.2 Fault and performance management

There are various stages in the protocol at which the coordinator or a participant **cannot progress** 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 example if 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.**

14.3 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*.

After all subtransactions have completed, the provisionally committed ones participate in a two-phase commit protocol. In other words provisional commit means

that coordinator and transactional servers are ready to commit subtransaction but they have not yet done so.

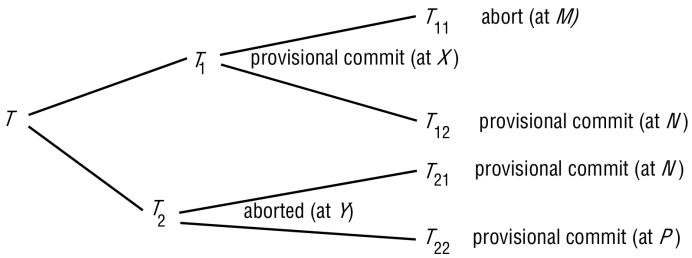


Figure 14.3: Distributed nested transaction

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.

14.4 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 the previous transactions in the **provisional commit list**. Partecipants that receive the message replies with it vote Yes or No if they can commit or not.
- **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.

14.5 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.

14.5.1 Locking

- In a distributed transaction, the locks on an object are held locally
- The **local lock manager** can decide whether to grant a lock or to make the requesting transaction wait.
- 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 14.4: Locking in distributed transactions

14.5.2 Timestamp

In distributed transactions, we require that each coordinator provides **globally unique timestamps**.

- It 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

14.5.3 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

14.6 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 be constructed by 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.

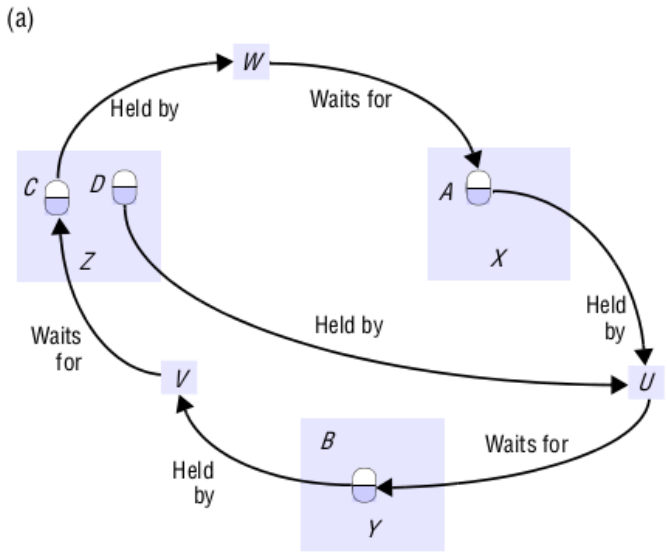


Figure 14.5: Example of global *wait-for graph*

We have three strategy to ensure distributed deadlock:

- **Centralized deadlock detection:** in which one server takes on the role of **global deadlock detector**. When it finds a circle it decide what to do and what transaction to abort. This method is not a good idea since it **depends on the server** to carry it out.
- **Phantom deadlock:** which is a deadlock that is "detected" but it is not really a deadlock. In other word 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:** in which 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**. When a cycle is detected, a transaction in the cycle is aborted to break the deadlock.

14.7 Transaction recovery

This propriety tell us that all the effects of committed transactions and non of the aborted one are reflected in the object they refer. This definition could be expanded by other two aspects:

- **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**.

This two propriety can be merged into a single mechanism called **recovery manager**. Its **tasks** are:

- Save objects in permanent storage for committed transactions.
- Restore the server's objects after a crash
- Reorganize the recovery file to improve the performance
- Manage recovery file

Any server that provides transactions needs to keep track of the objects accessed by client's 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*.

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

14.7.1 Logging

In the logging technique, the *recovery file* represents a log containing the **history** of all the transactions performed by a server. This *history* contains 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

14.8 Shadow versions

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.

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.

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.

14.8.1 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. Thus the recovery managers use two new status value 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.

So the **two phases** are:

- **First phase**, when the *coordinator* is prepared to **commit**, its recovery manager adds a coordinator entry to its recovery file.
  - **It votes Yes:** its recovery manager records a *participant entry* and adds an **uncertain transaction** status to its recovery file as a *forced write*.
  - **It votes No:** it adds an **abort transaction** status to its recovery file.
- **Second phase:** the recovery manager of the coordinator adds either a *committed* or an *aborted* transaction status to its recovery file, according to the decision

Chapter 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. 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.
  - *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.

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**. **Consistency models** can be considered as special agreements between processes and data store.

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**.

15.1.1 Strict consistency

Each **read** of a data x **returns** the value corresponding to the result of the **last write**.

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

Strict consistency is the **strongest consistency model**, in fact a write to a variable by any processor needs to be **seen instantaneously** by all processors. However 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*.

- 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*.

15.2 Casual 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.

The first one is an example of causal consistency and not sequential consistency. 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*.

For instance, If there is an operation or event A that *causes* another operation B, then **causal consistency** provides an assurance (assicurazione) that each other process of the system *observes* operation A before *observing* operation B

Moreover, if one *write* operation **influences** another *write* operation, then these two operations are **causally related**, so one relies on the other.

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. In the FIFO consistency write can be seen by all, also by whom is not interested in.

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
2. Operations on synchronized variables are not allowed until all the previous *write* are completed
3. *Read* or *write* operations are not allowed until all the previous synchronization operations are completed.

15.2.3 Relaxed Consistency

Introduced because data store can't recognize synchronized requests for *write* or for *read*. In order to **solve** this, different synchronization operations will be used:

- **Acquire:** to *enter* in 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

Updates can be done in two different ways:

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

15.2.4 Entry Consistency

Every shared data is associated to a synchronized variable and we access it with an acquire-release. So there is 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 the most recent information

15.2.5 Comparison of Consistency models

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.

Figure 15.1: 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.

Figure 15.2: Consistency models with synchronized operations

15.2.6 Eventual Consistency

Eventual consistency is **client centered**, it guarantees the correct access to the data store from the viewpoint of each client. It 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
- **Read your write:** the effects of a *write* on x by a process are always visible by successive *read* on x by the same process.
- **Write 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 remember that each logical object is implemented by a collection of physical copies cal. 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.

Replicas can be stored in memory in different ways:

- **Permanent:** they are static copies
- **Server-initiated:** they are created dynamically
- **Client-initiated:** they are based on the client cache

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. We shall always require that a replica manager applies operations to its replicas recoverably, in such a way that these operation can be **recovered**.

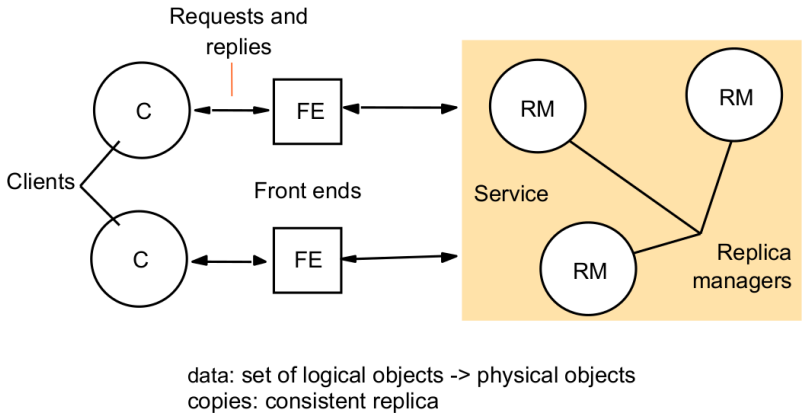


Figure 15.3: A basic architectural model for the management of replicated data

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, replicated by the managers
- Each client requests a series of operations
- Requested operations that involve **no updates** are called *read-only requests*
- Requested operations that **update** an object are called *update requests*

Each client's requests are first handled by a component called a **front end**.

- Its role is to communicate by message passing
- Front end introduces replication, location and access transparency
- 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 multicast 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
   - *FIFO ordering:* if a front end issues request r and then request r'
   - *Casual ordering:* if the issue of request r happened-before the issue of request r'
   - *Total ordering:* if a correct replica manager handles r before request r'
3. **Execution:** The replica managers execute the request, perhaps *tentatively*, in order to undo its effects later
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.

Chapter 16

The Role of Croup Communication

The membership of groups can be created in different ways, in replication circumstances, there is a strong requirement for dynamic membership. A group membership service implements several features:

- Provides an interface to manage group
- Implement a fault detector
- 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. 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 $p \neq q$

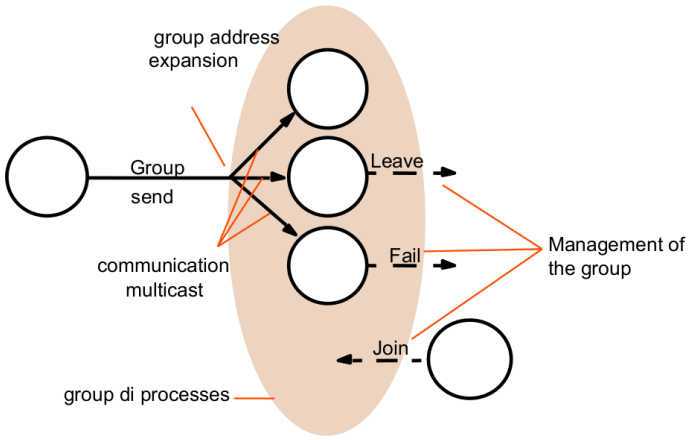


Figure 16.1: Services for a group of processes

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. These guarantees are:

- **Agreement:** correct processes deliver the same sequence of views and the same set of messages in any given view
- **Integrity:** if a correct process p delivers message m, then it will not deliver m again
- **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

Considering the following image, a group with three processes, p, q and r

- **Picture (a):** 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
- **Picture (b):** 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)
- **Picture (c):** it is not allowed for q and r to deliver first the view (q, r) and then m
- **Picture (d):** 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

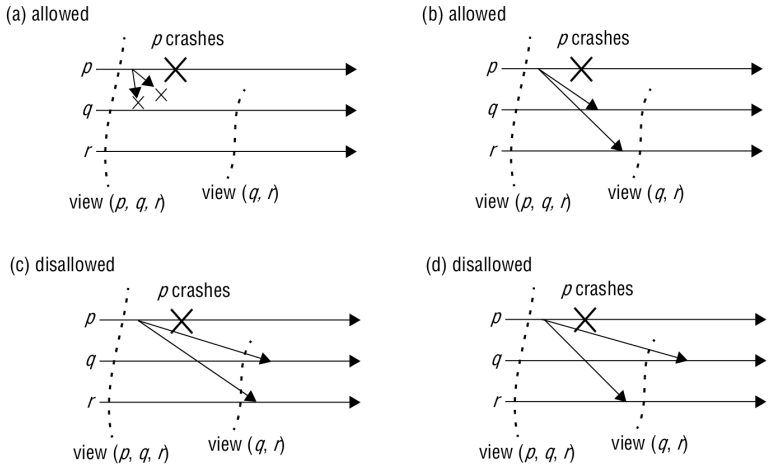


Figure 16.2: Delivery of consistent messages

16.1 Fault-tolerant 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.

16.1.1 Passive replication

In this model at any time there is a **primary replica** manager and one or more **secondary** replica managers. So *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:

- **Request:** the front end issues the request, containing a unique identifier, to the primary replica manager
- **Coordination:** the primary takes each request atomically, in the order in which it receives it
- **Execution:** the primary executes the request and stores the response.
- **Agreement:** if the request is an update, then the primary sends the updated state
- **Response:** the primary responds to the front end, which hands the response back to the client

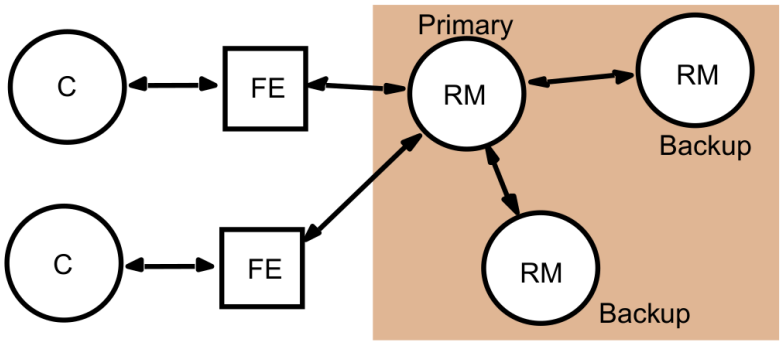


Figure 16.3: The passive model

This system obviously implements linearizability if the primary is correct, since the primary sequences all the operations upon the shared objects. 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
- The replica managers that survive agree on which operations had been performed

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.

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:

- **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
- **Coordination:** the group communication system delivers the request to every correct replica
- **Execution:** every replica manager executes the request
- **Agreement:** no agreement phase is needed, because of the multicast delivery semantics.
- **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

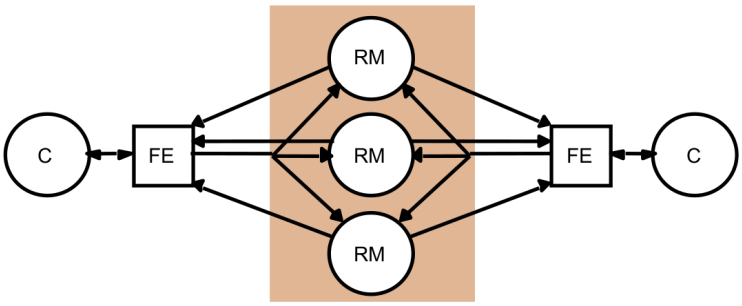


Figure 16.4: The active replication model

16.3 High available services

Now 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.

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 *transport the updates* they have each received from clients.

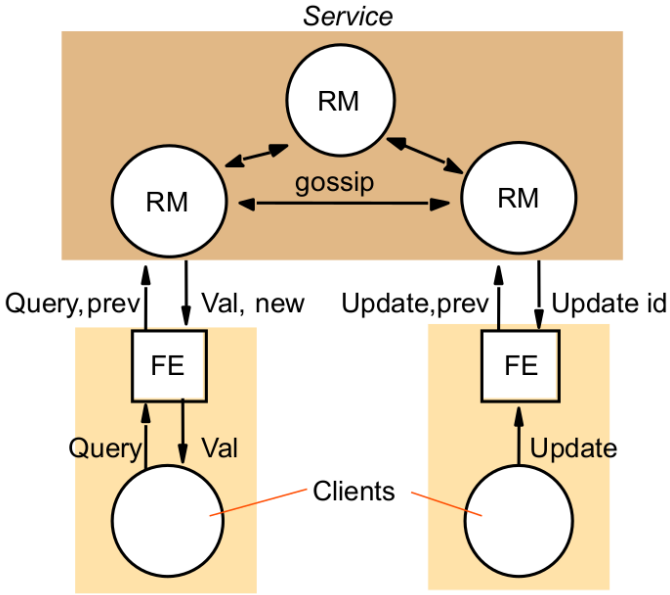


Figure 16.5: Query and update operations in a gossip service

A gossip service provides two basic types of operations:

1. *Queries*: read-only operations
2. *Updates*: modify but not read

The system makes two guarantees:

- *Each client obtains a consistent service over time*
- **Relaxed consistency between replicas**

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.

- It 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, since the replicas may have been updated since the last operation
- An update operation returns a vector timestamp 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.

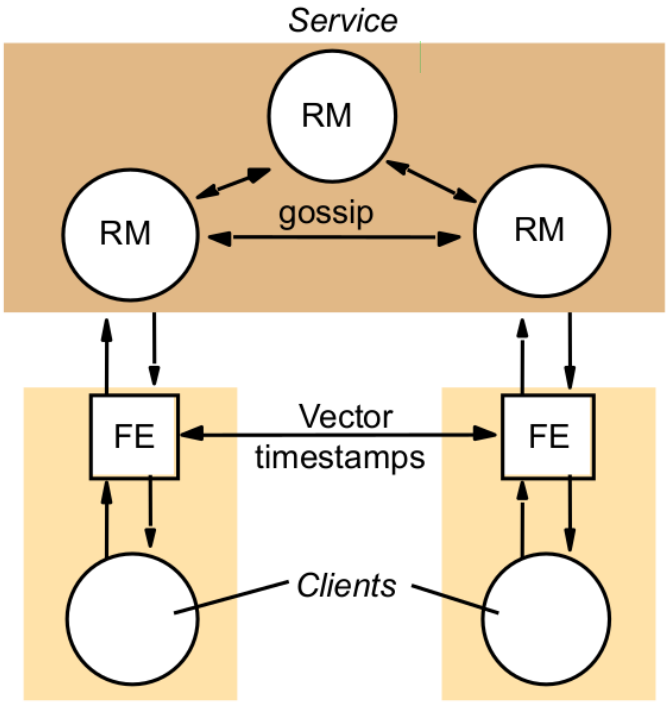


Figure 16.6: Front ends propagate their timestamps whenever clients communicate directly

Replica manager state. Regardless of the application, if contains the following steps:

- **Value:** this is the value of the application state as maintained by the replica manager

- **Value timestamp:** this is the vector timestamp that represents the updates that are reflected in 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
- **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
- **Timestamp table:** this table contains a vector timestamp for each other replica manager, filled with timestamps that arrive from them in gossip messages

16.4 Transactions with replicated data

A **transaction** on replicated objects should **appear the same** as one with *non-replicated* objects, and the **effect** should be the the *same* as if they had been performed one at a time on a *single set of objects*. This propriety is called **one-copy serializability**.

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. 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. 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.

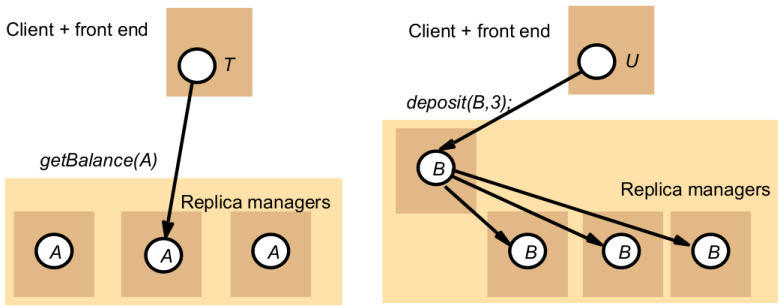


Figure 16.7: Transactions on replicated data

16.4.2 Available copies replication

Simple *read-one/write-all* replication is not a realistic scheme. The **available copies** scheme is designed to allow for some replica managers being temporarily unavailable. *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 the bellow image:

- *getbalance* operation of transaction T is performed by X
- Whereas its *deposit* is performed by M, N and P.
- Concurrency control at each replica manager affects the operations performed locally
- At X, transaction T has read A and therefore transaction U is not allowed to update A with *deposit* operation until transaction T has completed

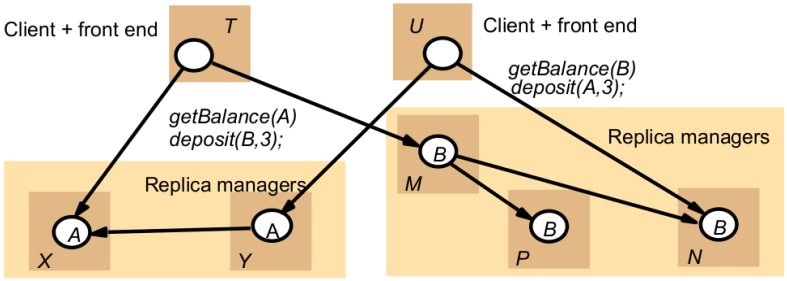


Figure 16.8: Available copies

16.5 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.

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.

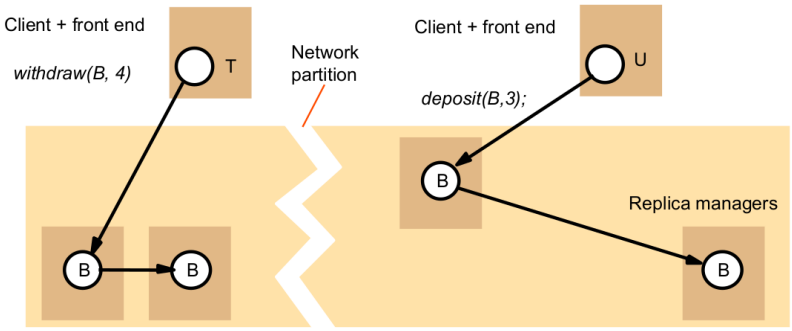


Figure 16.9: Network partition

The optimistic schemes do not limit availability during a partition, whereas pessimistic schemes do.

- The **optimistic approach** allows updates in all partitions
- The **pessimistic approach** limits availability even when there are no partitions, but it prevents any inconsistencies occurring during partitions. In other words it is based on the idea of **quorum**, only the partition that satisfy the quorum is available to the client.
- The **Combined approach** is based on algorithm of **virtual partition**

16.5.1 Virtual Partition

A **virtual partition** is an abstraction of a real partition and contains a set of replica managers. Note that:

- *"network partition"* refers to the **barrier** that divides replica managers into several parts
- *"virtual partition"* refers to the parts themselves

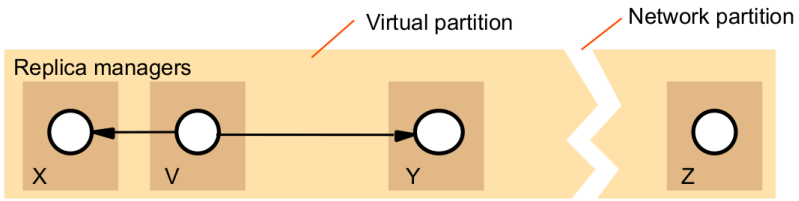


Figure 16.10: Virtual partition

16.5.2 Creating a virtual partition

1. Phase
   - 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**
2. Phase
   - 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 summary:

- 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

Chapter 17

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 should be comparable to that for files stored on local disk. **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.

17.1 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*.

- 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

17.2 Distributed File System Requirements

- **Transparency:** the design must balance the flexibility and scalability that derive from transparency against software complexity and performance.
  - *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
- **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
- **Fault Tolerance:** the central role of the file service in distributed systems makes it essential that the service continues to operate
- **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
- **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.

17.3 Case Study

17.3.1 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**.

- The **flat file service** refer to 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 **directory service** provides a mapping between text names for files and their UFIDs.
- The **client module** also holds information about the network locations of the flat file server and directory server processes

17.3.2 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

- The **NFS Server module** is stored in the kernel on each computer that acts as an NFS server
- 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

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

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.

17.3.3 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.

AFS is implemented as two software components that exist as UNIX processes called

- **Vice:** is a server that works at upper level that runs as a user-level UNIX process in each server computer
- **Venus:** is a client working at upper level 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 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

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**

Chapter 18

Mobility Computing

- **Mobile computing** is concerned with exploiting the connection 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.

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. There are two possible levels of mobility:

- **Code Mobility:** it is the capability to dynamically relocate at run time the distributed application components
- **Mobile Agent:** program in execution can migrate among machines in an heterogeneous networks

Next we have two important aspects of mobility:

- **Migration:** with mobile agent there is a sort of migration, thus we could have **migration of processes** or **migration of objects**. And the main goal is to implement a load balancing, which a system that distributes load uniformly between machines on the network.
- **Mobile Code:** defines different paradigms. 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:** here is a client entity that ask for a resource and the server reply with an answer
  - **Remote evaluation:** involves the transmission of executable software code from a client computer to a server computer for subsequent execution at the server

  - **Code on demand:** sends executable software code from a server computer to a client computer upon request from the client's software.
  - **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. 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.

Issues of Mobility are:

- Security
- Communication
- Data space management
- Mobility specification (how?)
- Mobility management (when?)
- Choose of components to migrate (who?)

And the **advantages** of mobility code are:

- Maintainability
- Flexibility of data management and in protocol
- Inclusion
- Reliability
- Autonomy.

18.1 Types of code mobility

- **Strong Mobility:** involves moving both the code, data and the execution state from one hos 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

18.2 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
- we have different strategies to manage the data space:
  - **Binding Removal:** remove the binging to resources
  - **Network Reference:** maintain the reference but make it remote
  - **Move:** moves directly the resource and it remains local.
  - **Copy:** duplicate the resources and code
  - **Re-Binding:** instead of applying a copy try to use an object of the same type

18.3 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, 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

18.4 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.

Chapter 19

Cloud Computing

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources 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

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

- **Cloud Computing** has the goal of providing services
- **Data Center** can be considered only as a collection of a large amount of data stores to provide data

19.1 Terminology

- **Cluster of workstation:** a cluster is a group of independent IT resources that are interconnected and work as a single system.
- **Grid computing:** a computing grid provides a platform in which computing resources are organized into one or more logical pools. 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. This middle tier can contain load balancing logic, fail over 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. These resources can be shared by a set of users. Modern virtualization improves performance reliability and scalability.
- **Distributed computing:** tightly coupled, homogeneous and single administration
- **Cloud Computing:** On demand, service guarantee, virtualization
- **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

- **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
- **Computational 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

The **fundamental concepts** belonging to the notion of cloud computing are the following one.

19.3.1 Cloud

A cloud refers to a distinct environment that is designed for the purpose of remotely provisioning scalable and measured resources.

- As a specific environment, a cloud has a finite *boundary*
- Resources provided by cloud environments are dedicated to supplying back-end processing capabilities and user-based access to these capabilities

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.

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*

19.3.4 Scaling

Scaling represents the **ability of the IT resource to handle increased or decreased usage** demands. And there are two types:

- **Horizontal Scaling:** 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**
  - The horizontal *releasing* of resources is referred to as **scaling in**

  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).

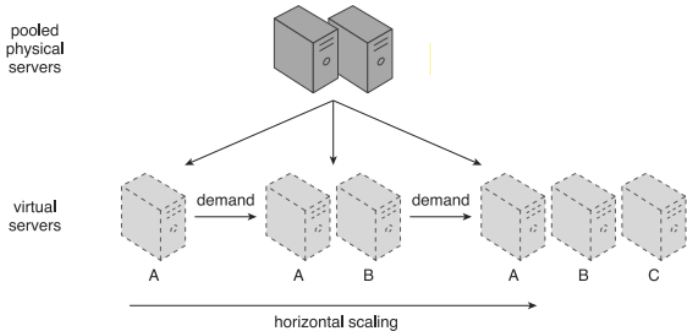


Figure 19.1: Example of Horizontal scaling

- **Vertical Scaling:** when an existing IT resource is *replaced by another* with higher or lower capacity, vertical scaling is considered to have occurred.

- The replacing of an IT resource with another that has a *higher capacity* is referred to as **scaling up**
- The replacing an IT resource with another that has a *lower capacity* is considered **scaling down**

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)

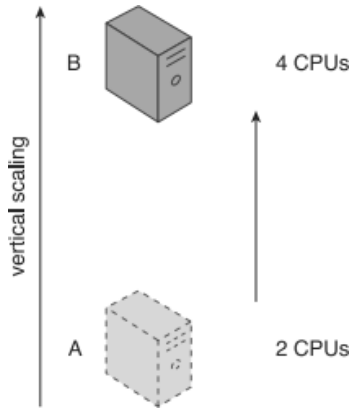


Figure 19.2: Example of Vertical scaling

19.4 Cloud Service

A **cloud service** is any IT resource that is made *remotely accessible* via a cloud. A Cloud Service could be:

- A service with a published technical interface accessed by a consumer outside of the cloud which is likely being invoked by a consumer program
- A service that exists as a virtual server is also being accessed from outside of the cloud's boundary, like by a human user that has remotely logged on to the virtual server

The **cloud service consumer** is a temporary runtime role assumed by a software program when it accesses a cloud service.

19.5 Cloud Computing Advantages

- **Performance**
- **Cost reduction:** the number of investments is reduced and there's an access to powerful infrastructures without purchasing them. More benefits like:
  - On-demand access to pay-as-you-go computing resources
  - 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

- **Scalability:** clouds can instantly and dynamically allocate IT resources to cloud consumers, on-demand or via the cloud consumer's direct configuration
- **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 Risk and Limits

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

- Requires high and available connection between nodes
- 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

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:

19.7.1 Infrastructure as a Service (IaaS)

- **Cloud Provider** offers access to fully functioning software
- **Service Provider** owns the equipment and is responsible for housing, running and maintaining it
- **Client** typically pays on a per-use basis
- The main **goal** of IaaS environment is to provide cloud consumers with a high level of control and responsibility over its configuration and utilization
- The resources provided by IaaS are not preconfigured therefore this model is used by cloud consumers that require a high level of control over the cloud enviroment
- **Examples:** Amazon Web Services, Google Cloud Storage, DigitalOcean.

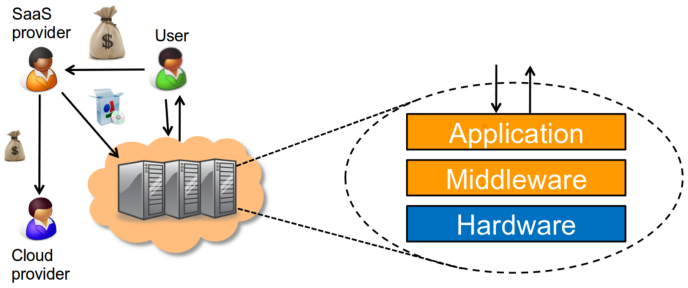


Figure 19.3: Infrastructure as a Service structure

19.7.2 Platform as a Service (PaaS)

- The **cloud provider** give the way to rent hardware, operating systems, storage
- 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 thatthe user runs software on
- The **platform** usually consists of an operating system and an execution environment
- In this model, customers **don't want to think about the server** or its internals, they want to point to a virtual machine
- It is a **'ready to use'** environment with IT resources already deployed and configured
- **Examples:** Windows Azure, Google App Engine, Heroku

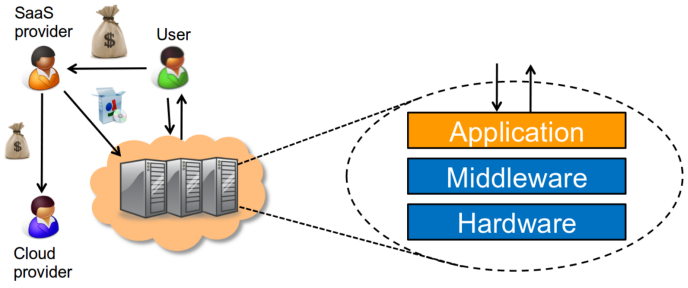


Figure 19.4: Platform as a Service structure

19.7.3 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

- The **user** has full control and responsibility over the running operating system and can use it to run any software
- **Examples:** Google Apps, Yahoo!Mail, CRM software.

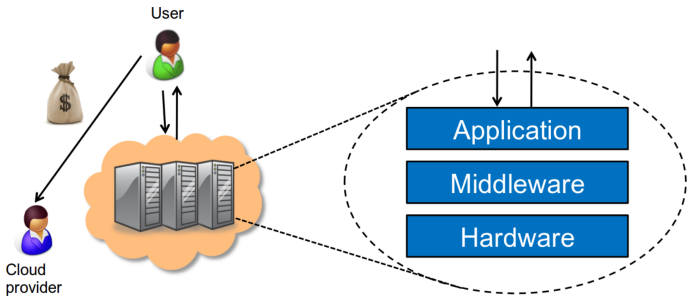


Figure 19.5: Software as a Service structure

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
- **Community Cloud:** the *owner* is a community of consumers and it customers outside of it 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. It enables an organization to use cloud computing technology as a means of centralizing access to IT resources by different locations
- **Hybrid Cloud:** formed most commonly by *private* and *public* cloud. Usually cloud consumer choose to deploy cloud services processing **sensitive data** to a *private cloud* and **less sensitive** cloud services to a *public cloud*

19.9 Potentialities and concerns

- **Why use a cloud computing system?** For its advantages: cost, scalability and performance
- **Cases where cloud computing system cannot be adopted:** legislative frameworks, medical records and data privacy