Unit 7. Distributed Systems Basic Concepts

Concurrency and Distributed Systems

Teaching Unit Objectives

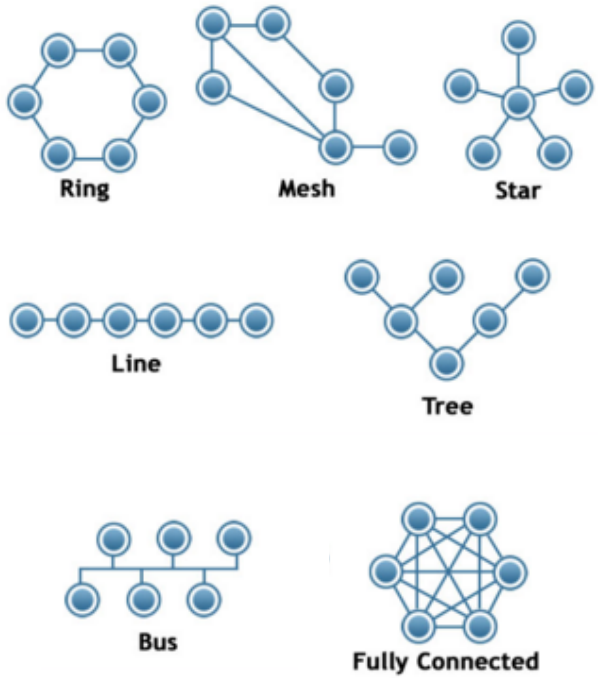
- Characterize the **distributed systems** and detail their main features.
- Identify the main **objective** they pursue as well as the difficulties that must be overcome.
- Identify the usual design **techniques** and **principles** for its construction.

Content

- Concept of a Distributed System
  - Definition of a Distributed System
  - Examples
  - Objectives of Distributed Systems
  - Middleware
- Features of Distributed Systems
  - Transparency
  - Availability
  - Scalability
  - Security

Definition of a Distributed System

- Natural extension of **concurrent systems**, when they run on more than one computer.
- Each computer in the distributed system is called a **node**
  - Nodes are connected to each other through **communication channels**.
  - The result is a **graph**, where the computers are the nodes and the edges of the graph are the communication channels.
  - Depending on the shape of the graph, we will have different **topologies** (totally connected graph, ring graph, etc.)



Definition of a Distributed System

- **Distributed System: set of independent computers that offer their users the image of a single coherent system.**
  - At hardware level: autonomous machines
  - At user level: image of a single system
  - At internal level: distributed algorithms

- At hardware level: **Autonomous machines**
  - Hardware (memory, clock, disk, etc.) is not shared among them.
  - They are machines that run and fail independently of each other.
  - These are computers that could be separated physically, without any hardware preventing it.

Definition of a Distributed System

- At user level: **Image of a single system**
  - Users who access the distributed system observe it as if it were a single computer.
  - We also refer to this property as **distribution transparency**: the fact that the system is distributed among several computers is hidden.
    - This aspect differentiates it from *networked systems*, where users access each of the different computers that make up the network, individually, knowing their names or network addresses.

Definition of a Distributed System

- At internal level: **distributed algorithms**
  - Each **node** executes a part of the algorithm.
  - The execution is concurrent between all the nodes.
  - A distributed system can be viewed as a **collection of distributed algorithms** built with a common goal.
  - In each distributed algorithm, the nodes communicate with each other and synchronize, usually by **sending and receiving messages**.
    - The distributed algorithms have many similarities with the *concurrent algorithms*, where the difference lies in arranging each activity in a different computer, instead of being threads within the same process.

Examples of Distributed Systems (I)

- Cloud storage systems
  - Dropbox, Google Drive, OneDrive, etc.
- Messaging systems
  - Whatsapp, viber, line, etc.
- Search systems
  - Google, Bing, yahoo, etc.
- Resource directory systems
  - DNS, LDAP, etc.
- User authentication systems
  - Kerberos, sesame, etc.
- Portals of information and public or private services
  - Poliformat.upv.es, agenciatributaria.es, valencia.es, etc.

Examples of Distributed Systems (II)

- Social Networks
  - Facebook, twitter, etc
- Media distribution
  - Netflix, youtube, flickr, etc.
- Peer-to-peer (P2P) systems
  - BitTorrent, JXTA, Napster, Gnutella, eMule, etc.
- Grid Systems
  - Globus Toolkit, Seti Project, etc.
- Online games, banking systems, control systems, etc, etc.
- **Nowadays it is difficult to find systems that are not distributed or that do not have or use a distributed component.**

Examples of NOT distributed systems

- Examples of not distributed systems
  - A set of computers in a network, where a user connects with "ssh" to each of them separately, and knows what can be executed in each one of them.
    - However, in this network of computers, distributed systems could coexist, such as LDAP, DNS access, distributed file systems, distributed printing systems, etc.
  - A set of processes within the same computer.
    - However, most distributed systems are initially developed, in a prototype first stage, within a single computer. But its design and vocation lies in a later physical distribution.
  - A concurrent program consisting of several threads, within a single process.

Objectives of Distributed Systems

The main objective of any Distributed System is to provide users with access to remote resources, **easily** and **reliably**.

- **Resource** must be understood broadly
  - Examples: printers, computers, storage devices, other connected users, photographs, videos, bank accounts, etc.
    - Example: cloud storage services (Dropbox, Google Drive, OneDrive, etc.)
    - Example: messaging systems (whatsapp, viber, line, etc)
    - Examples: Netflix, Google, Online banking systems, Facebook, Online games, etc.
- Advantages:
  - Economy of resources: users do not need to have all the resources locally.
  - Sharing of resources between users.

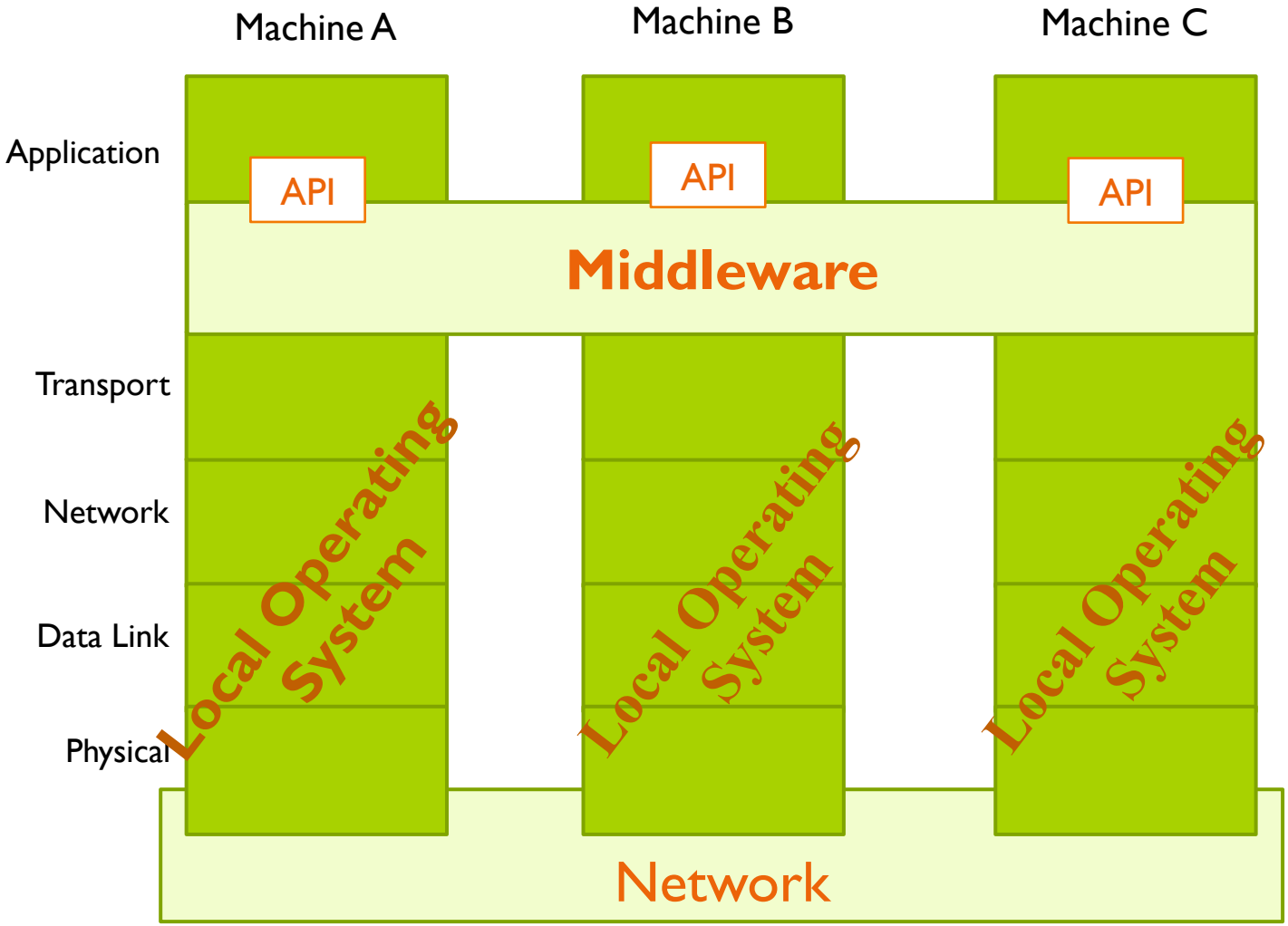
Middleware

It is defined as a software layer above the operating system, but under the application program that provides a common programming abstraction throughout a distributed system

Bakken, D. 2001

- From the point of view of the application programmer, it is a simple **API** with which to access distributed services.
- The use of the API triggers a distributed work between the different nodes of the system, which will collaborate to offer the distributed service
- The use of middleware services allows the development of **distributed applications**, preventing the developer from developing part of its complexity: transparency, availability, scalability or security.
- Likewise, when developing a distributed system, it will be convenient to offer the services in the form of middleware, ideally through interfaces that use **open standards**.

Middleware

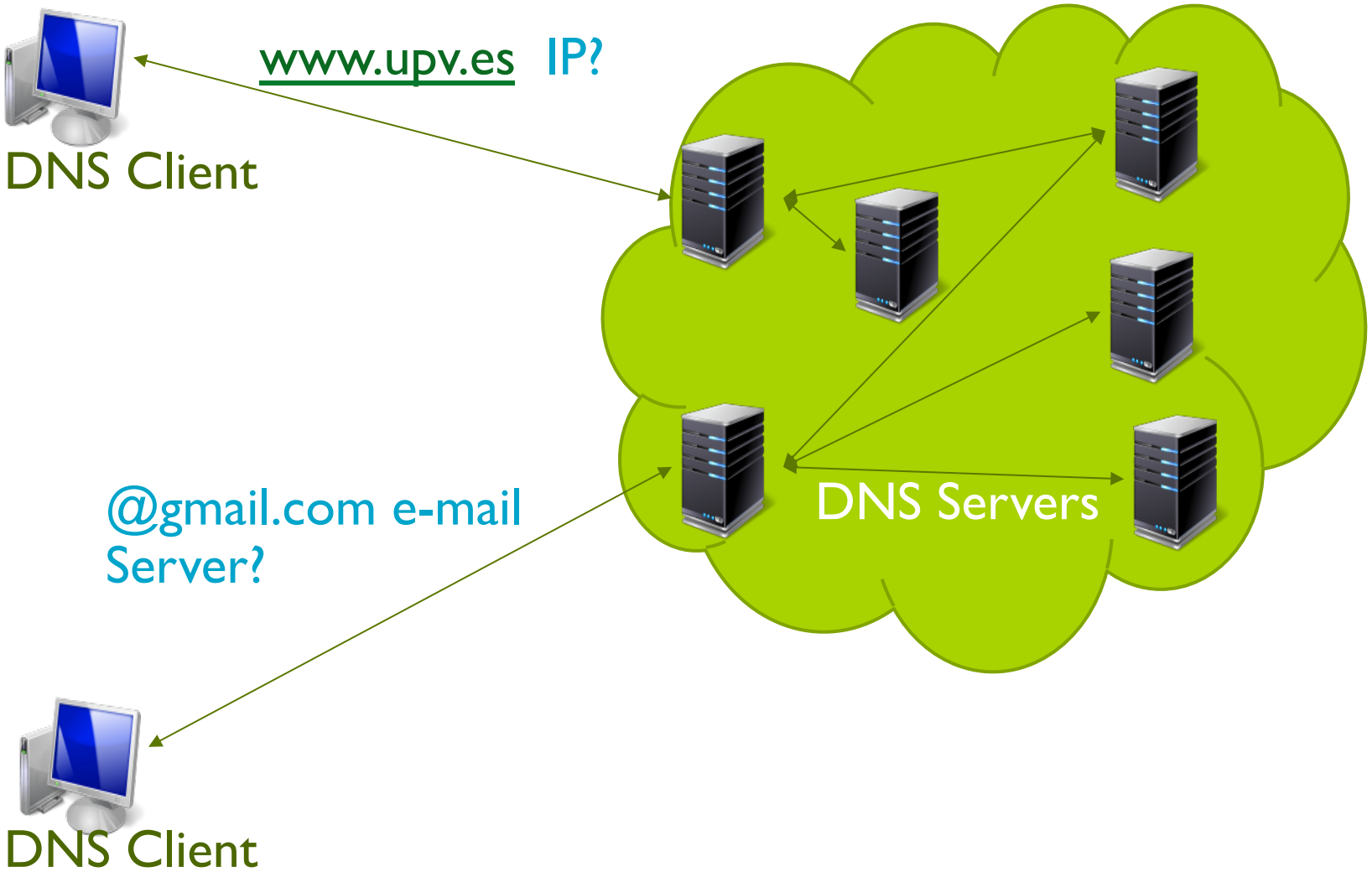


It helps the development of distributed applications, simplifying the programmer much of its complexity.

Middleware

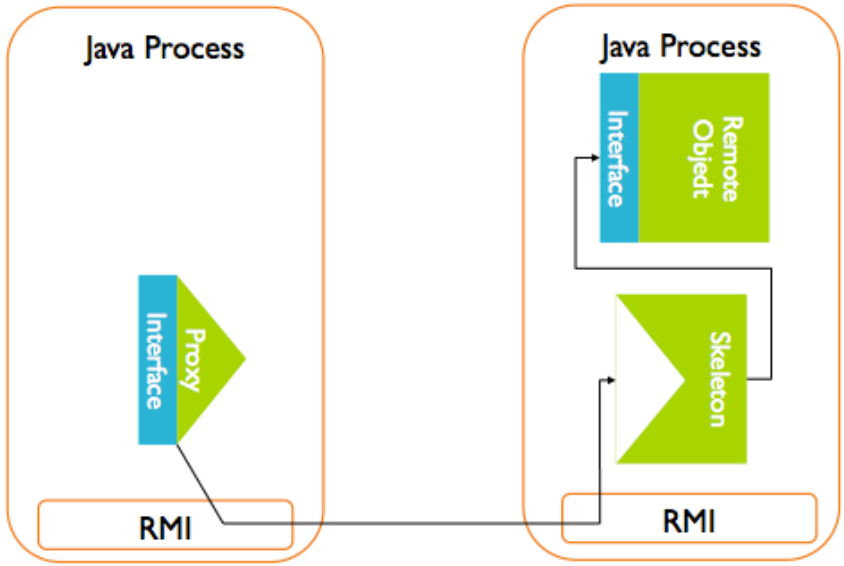
- Examples of Middleware:
  - Java Message Service (JMS)
  - Java Remote Method Invocation (RMI)
  - Domain Name System (DNS)
  - Open Database Conectivity (ODBC)
  - Lightweight Directory Access Protocol (LDAP)

Example of middleware: DNS



Example of Middleware: Java RMI

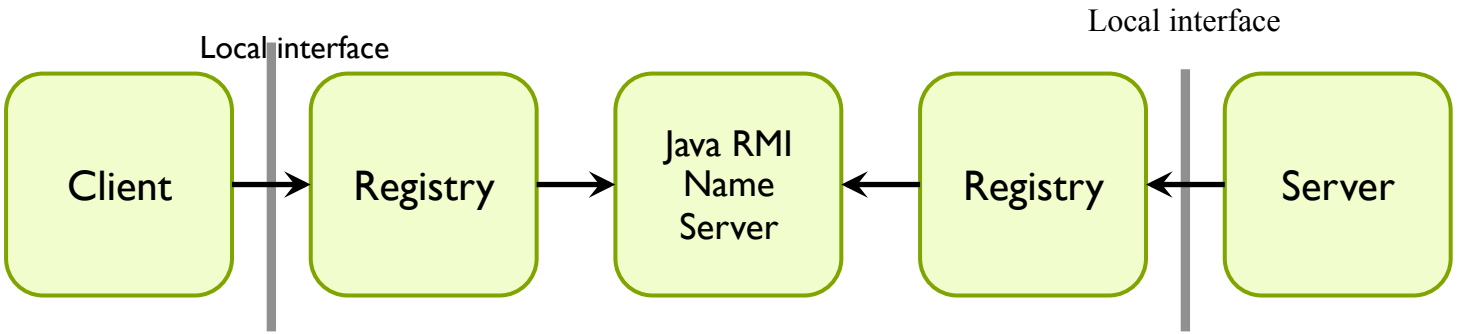
- **Java RMI** (*Remote Method Invocation*) is an **object-oriented communication middleware**
  - It allows **invoking** Java object methods of another JVM, and **passing Java objects** as arguments when invoking these methods.



  - For each remote objects, RMI dynamically creates an object called a **skeleton**
  - To invoke a remote object from another process, we use a **Proxy**
    - Allows locating the *skeleton* of the remote object

Java RMI Name Server

- The **name server** allows registering remote objects, so they can be located.
  - The *name server* can be hold in any node. It can be accessed by both client and server using a local interface named *Registry*
- In Java Oracle distributions, the name server is launched using the **rmiregistry** order.



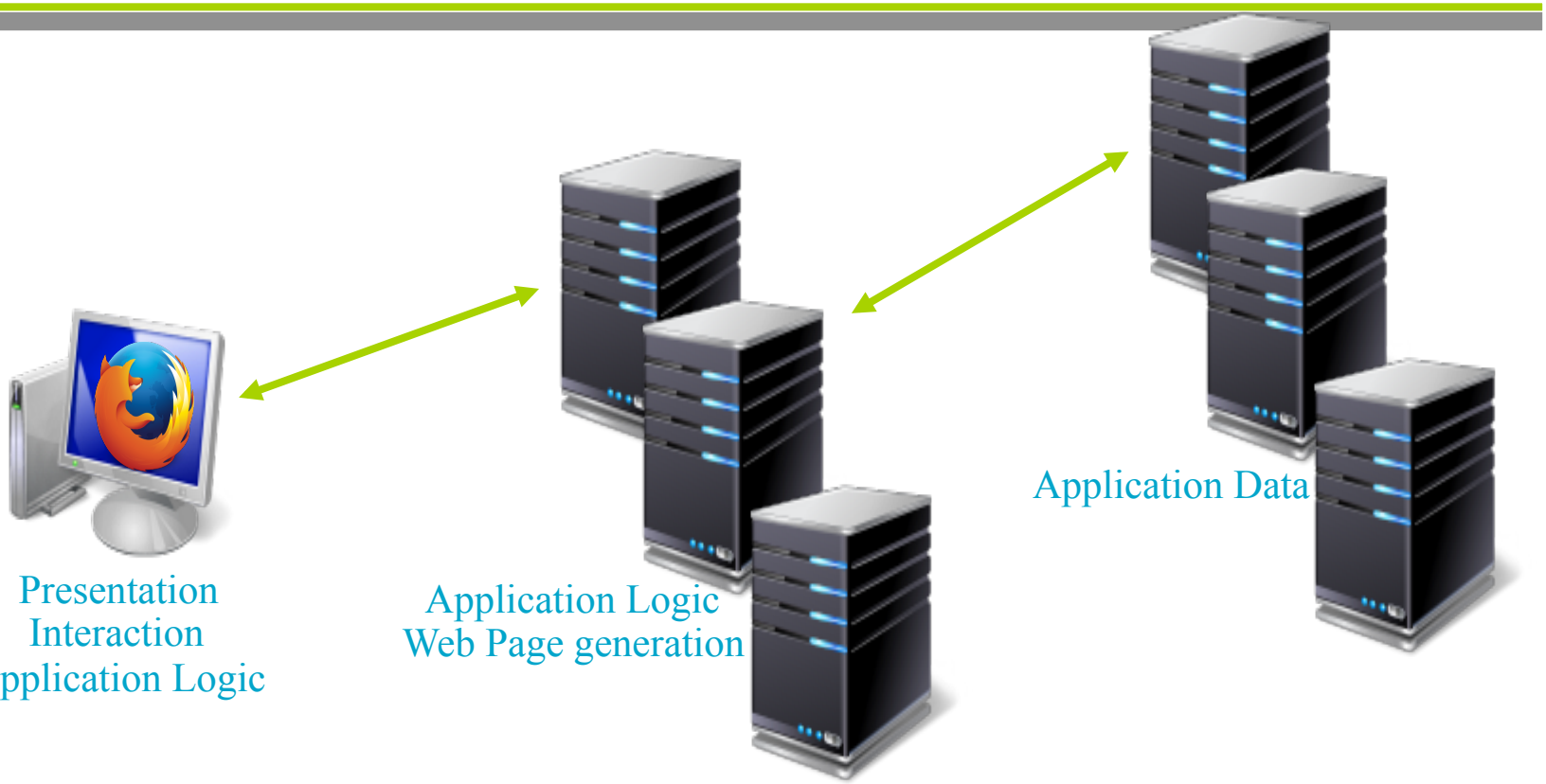
Content

- Concept of a Distributed System
  - Definition of a Distributed System
  - Examples
  - Objectives of Distributed Systems
  - Middleware
- Features of Distributed Systems
  - Transparency
  - Availability
  - Scalability
  - Security

Characteristics of Distributed Systems

- The difficulties inherent in the construction of distributed systems become challenges to be overcome and finally in intrinsic characteristics that can be found in any distributed system, to a greater or lesser degree:
  - **Transparency**: the fact itself of the distribution is hidden, the differences between the different machines and the complexity of the communication mechanisms.
  - **Availability**: the services they offer should always be available. These are intrinsically fault-tolerant systems and where maintenance tasks should not interrupt the service.
  - **Scalability**: they must be relatively simple to expand, both in users, resources and number of nodes.
  - **Security**: resources and users must coexist respecting the restrictions and access rules and security policies that will depend on each system.

Example of Distributed System: PoliformaT



Content

- Concept of a Distributed System
  - Definition of a Distributed System
  - Examples
  - Objectives of Distributed Systems
  - Middleware
- Features of Distributed Systems
  - Transparency
  - Availability
  - Scalability
  - Security

Transparency

- **Distribution transparency** (image of a single system): hide from the user the fact that processes and resources are physically distributed over different computers.
  - Achieving transparency, **simple services** are offered to the user
    - It hides the complexity of the algorithms that are executed, the failures that may happen, the number of computers involved, their location and all those features that are irrelevant to the end user.
  - To achieve transparency of distribution, different specific aspects must be hidden, called **Axes of distribution transparency**

Axes of distribution transparency
Location Transparency
Failure Transparency
Replication Transparency
Other axes of transparency (persistence, concurrency, migration, transaction, etc.)

Axis of Distribution Transparency

- **Location transparency**: the location of the resources is hidden from the user.
- **Examples**

  - Dropbox users do not need to know on which computer the copies of their files are.
  - Google map service users do not need to know which computer the maps are on.
  - Web page users rarely want to know where the resources they access are. It is enough to be able to click on links to obtain more information or specific resources.
- **Mechanisms**
  - To achieve location transparency, resources are usually identified with **unique symbolic names.**
    - When users want to access resources, they will use the symbolic names.
    - The system will translate the symbolic name to its true location, "transparently" to the user.
  - **Name services**, location and resource search services, directory services, etc.

Axis of Distribution Transparency

- **Failure transparency:** it is hidden the fact that the components of a distributed system fail.
  - The more computers are part of a distributed system, the higher the probability that one of them will fail. The user is not interested in knowing when a computer fails.

  - **Examples**
    - It is estimated (Wikipedia, 2017) that the Google search service has 2,000,000 computers. Assuming a conservative failure rate, a failure in each computer every 5 years, means that about 1,000 computers fail each day. About 1 failure per minute!
    - The user does not perceive these failures.
  - **Mechanisms:**
    - **Fault detector** → the nodes are continuously monitored
    - **Replication** → all resources are replicated in more than one node. If the failure of one node is detected, another node will be responsible for continuing to provide the same service on the copy of the resource.
    - **Fault Tolerant Algorithms** → All algorithms running in the system must react appropriately to the failure of a node, offering the same service before and after the failure.

Axis of Distribution Transparency

- **Replication transparency**: it is hidden the fact that resources are replicated in more than one node.
  - Systems are replicated to achieve available systems (fault tolerance) and to increase scalability (more users accessing the system simultaneously, more resources, more nodes).

  - **Examples**
    - When you upload a picture to a Facebook profile, that picture is stored on several computers. In this way, in case of failures, another computer can serve it. Additionally, several different computers serve copies of the same photograph to different users who visit that profile.
    - The user is not interested in knowing how many computers have a copy of the photograph, nor which replica is the one that offers the service, nor in knowing that such replication exists.
  - **Mechanisms:**
    - Mapping of the symbolic names of resources to the location of different replicas.
    - Load balancing algorithms, to choose which replica serves each request on each resource.
    - Replication algorithms that offer **consistency between the replicas**

Axis of Distribution Transparency

- **Other axis of distribution transparency**
  - There are multiple types of transparency that we can find in the literature. All of them with the vocation to collaborate in the common goal of achieving distribution transparency and providing simple services to be used.
  - They are present to a greater or lesser extent in certain systems and will be important depending on the type of distributed system that we are studying.
  - **Persistence Transparency**
    - It is hidden the fact that resources are stored on disk
  - **Concurrence Transparency**
    - It is hidden the fact that the system is being used by multiple users at the same time
  - **Migration Transparency**
    - It is hidden the fact that resources can move.
  - **Access Transparency, Transaction Transparency, Relocation Transparency, etc.**

Considerations about transparency

- Providing transparency has a **high cost**
  - Cost in a greater number of computers, in computers and networks of higher performances, in a higher cost of development and maintenance of the systems, and in higher algorithmic cost (spatial, temporal, number of messages).
  - The greater the transparency → the higher the cost and quality observed by users
- Sometimes total transparency in any of the axes does not interest. The convenience depends on each particular system.
  - For example: ".es" domains (you should know that they are pages from Spain)
- Sometimes total transparency is impossible to achieve or would have too much cost for a given system.
  - For example: "poliformaT" is located on the campus of the university. A general failure that affects the networks or supply of the entire university, would cause a violation of the failure transparency → the cost of offering higher failure transparency may be unapproachable at the moment.

Content

- Concept of a Distributed System
  - Definition of a Distributed System
  - Examples
  - Objectives of Distributed Systems
  - Middleware
- Features of Distributed Systems
  - Transparency
  - Availability
  - Scalability
  - Security

Availability

- We can define **availability** as the probability that a certain system offers its services to users.
  - We talk of highly available systems when the probability is greater than 99.999%
  - In critical systems, higher probabilities are considered and we talk about ultra-highly available systems.
- There are 3 factors that affect availability:
  - Failures
  - Maintenance tasks
  - Malicious attacks

In this section we focus on the general concepts related to **fault tolerance**

Factors that affect Availability

- **1) Failures**
  - The nodes fail and the networks fail.
  - Systems must be designed so that they continue to offer services in the presence of failures → **fault tolerant systems.**

- **2) Maintenance tasks**
  - Every system requires maintenance.
    - Tasks such as replacing disks, altering the configuration of systems, backing up, expanding systems, or even changing the entire system.
  - Systems must be designed to allow maintenance to be carried out at the same time that users access the services.

Factors that affect Availability

- **3) Malicious attacks**
  - Every system is a potential target for malicious attacks.
  - From intruders that alter, delete, intercept or impersonate resources to distributed denial of service attacks.
  - Systems must be designed to be resistant to attacks so that the service offered is not interrupted → **computer security**

Availability

In this section we focus on the general concepts related to **fault tolerance**

- **Every distributed system must be fault tolerant**. The system must continue to operate and provide service in the presence of faults.
- The basic mechanism to achieve fault tolerance in distributed systems is **replication.**
  - A service is configured as a set of nodes, called **replicas**, in such a way that the failure of a replica does not prevent the service from continuing to function.

Availability

- Replication introduces the problem of **consistency**: the degree of similarity/difference between the different replicas.
  - Systems with a lot of consistency, or **strong consistency**:
    - Ideally all replicas are equal to each other at all times.
    - This objective is impossible to achieve, but you can achieve systems with quite strong consistency, where service users get the same response regardless of the replica that serves them.
  - Systems with little consistency, or **weak consistency**:
    - Replicas can diverge, so that each of them can give different answers at a certain moment.

    - *NOTE: The study of consistency will be seen in subjects of upcoming courses. In this course we will talk about* ***strong consistency (almost identical replicas) or weak consistency (potentially different replicas).***

Availability: types of failures

- **Types of failures:**
  - Failures can be **simple**, affecting a single node or a single communication channel, or **compound**, affecting several nodes and channels simultaneously.
  - Failures can be **detectable** or **undetectable**.
    - They are *detectable* if another node is able to observe the failure.
    - They are *undetectable* otherwise.

Availability: types of failures

- **Simple detectable failures**
  - A failure is detectable if another node is able to observe the fault.
    - **Stop failure or Crash failure**
      - The node halts, but is working correctly until it halts.
      - Another node can detect it by means of continuous monitoring, with periodic "pings".
    - **Timing failure**
      - The node fails, taking too long to respond.
      - Another node can detect it with timers associated with each request, also by means of periodic "pings".
    - **Detectable response failure**
      - The node fails, providing a wrong answer, detectable as such.
      - Another node can detect it, accepting only valid response ranges.

Availability: types of failures

- **Simple detectable failures**
  - In general, all detectable simple faults can be treated in a similar way.
    - **Replication** is used.
    - When the failure of a replica is detected, the replica that has failed is expelled, and the rest of the replicas continue offering the service.
      - Note that the replica is expelled even if it continues to work (maybe it works, but it took a long time to answer, or it did wrongly). To avoid interferences from this replica, messages that this replica may send in the future will be ignored.

Availability: types of failures

- **Simple undetectable failures**
  - Also called **Byzantine failures**
  - A node fails exhibiting an arbitrary behavior or providing a response that cannot be detected as a failure.
  - Byzantine failures can be due to different causes: software errors, hardware errors and/or malicious attacks.
  - Example:
    - We have a system that measures the ambient temperature. If the temperature exceeds a certain measurement (> 25 degrees), it starts refrigeration. The admissible temperature range is 10-40 degrees.
    - A node measures the temperature, offers as a service to consult temperature. Another node uses this service and decides whether to start the refrigeration. If the node does not answer, or answers late, or answers out of range → detectable failure.
    - But what happens if the room temperature is 15 degrees, but the node answers by saying 30 degrees? → undetectable failure.

Availability: types of failures

- **Simple undetectable failures**
  - Byzantine failures are treated differently to detectable failures.
    - Replication can be used, but in a different way to the case of detectable failures:
      - Each request is sent to all replicas and all of them reply. The majority response is chosen. These algorithms are also called **quorum algorithms**. At the same time, an alarm is generated about the lack of unanimity → scheme used in ultra-highly available systems.
    - There are other systems that use cryptography-based mechanisms to tolerate Byzantine failures → examples in crypto-currencies, and in threshold cryptography.

Availability: types of failures

- **Compound failures**
  - Multiple nodes or several communication channels fail simultaneously.
  - In most cases, they are treated in the same way as the occurrence of several simple faults consecutively.
    - **Several stop failures** → they are treated each independently and consecutively.
    - **Several Byzantine failures** → they are treated assuming that there will be a majority of nodes not affected by failures and the correct majority will be able to continue offering the service.

Availability: types of failures

- **Compound failures**
  - A type of compound failure deserves special attention: **partitions**.
    - Several failures occur in nodes or communication channels that leave the system divided into 2 or more subgroups.
    - Relevant problem in the current large systems (cloud computing)
    - **CAP Theorem:** (Consistency, Availability, Partitions)→ It is impossible to achieve a system that offers at the same time strong Consistency, high Availability and may occur Partitions.

Availability: types of failures

- **Compound failures**
  - **CAP Theorem:** impossible to achieve a system that offers at the same time strong Consistency, high Availability and may occur Partitions.
  - There are CA, CP and AP systems
    - **CP:** Systems with strong consistency and partitions (without high availability)
    - **CA:** Systems with strong consistency and high availability (without partitions)
    - **AP:** Systems with high availability and partitions (without strong consistency)

Availability

- **Mechanisms to achieve fault tolerance** (assuming only *simple detectable failures* that do not cause partitions):
  - Failure detectors
  - Group membership service
  - Replication

Availability

- **Failure detectors**
  - Failure detection is usually integrated into a **failure detection module** built into each node. This module is responsible for monitoring another node or several nodes and emiting "suspicion of failure".
  - More than detecting a failure, it is usually indicated that the node **suspects** the failure of another node
    - Note that there may be a faulty channel between both nodes and therefore both will suspect each other.
  - In case of suspecting a fault, the **failure detection module** notifies the group membership service.

Availability

- **Group membership service**
  - Service responsible for establishing an agreement between alive nodes, on which nodes have failed.
  - In case of suspicion of a failure, or several suspicions, this service initiates a **phase of agreement**, to determine which node or nodes have failed, (generally from the point of view of the majority).
  - If the failure of a node is agreed, the service will then expel it and notify all the nodes that remain "alive" informing them of the failure.
    - The nodes that receive the failure notification will ignore the possible messages that come from that node and will be reconfigured to continue working without that node.

Availability

- **Group membership service**
  - Note that it is possible that, given a certain suspicion, it is determined that the node that fails is the one that issued the suspicion, since the other nodes agree that the suspicion is unfounded.
    - Perhaps the node suspected because he himself is running too slowly.
  - We can state this form of work saying: **all detectable simple failures are converted to stop failures**
    - Independently of the type of failure, if we detect it and the failure is agreed, some node is expelled and the other nodes continue to operate, ignoring the node that was expelled.

Availability

In distributed systems, when **we say that a node has failed**, we are saying that **this node has been expelled from the system** by a certain membership service. Note that it is possible for the node to continue functioning.

All the nodes that remain alive will receive notification of the failure, sooner or later.

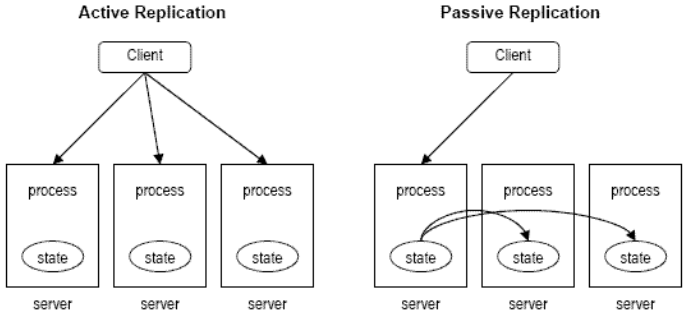
When a node receives notification of the failure of another node, it will be **reconfigured** to continue working without the expelled node:

- All the distributed algorithms in which the node intervenes will be reconfigured
- All the services present in the node will be reconfigured

Availability

- **Replication**
  - Each service is configured with more than one replica, so that in case of failures, the replicas that remain "alive" are reconfigured and continue offering service.
  - The clients of the service access the service with **replication transparency**, observing as the only difference, a greater availability.

Replication



- **Replication Schemes:**
  - **Passive replication**
    - Among the group of replicas, a single replica will be the **primary** replica. The rest of replicas are **secondary** ones.
    - The name "passive replication" comes from the role of secondary replicas, which do not process requests. They are passive.
      - The primary replica is the only "active" replica. The only one that works.
      - This scheme is also called **primary-secondary replication**.
  - **Active replication**
    - All replicas are the same. All receive the requests, all process the requests and all answer the client.
  - **Semi-active, semi-passive replication:** hybrid schemes, mixing both approaches.

Replication

- **Replication Schemes: Passive Replication (I)**
  - Among the group of replicas, a single replica will be the **primary** replica. The rest of replicas are **secondary** ones.
  - This scheme is also called **primary-secondary replication**.
  - The primary replica receives all the requests from the customers of the service, processes them and answers the clients.
  - For each request that involves a change of status, the primary replica will broadcast a status update message (**checkpoint**) to the secondary ones.
    - If the *checkpoint* message is sent to the secondaries, waiting for the corresponding confirmation, before reponding to the client, there will be a **replicated system with strong consistency**.
    - If the *checkpoint* message is sent to the secondaries later (after responding to the client), we will have a **replicated system with weak consistency**.

Replication

- **Replication Schemas: Passive Replication (II)**
  - Reconfiguration in case of failures:
    - In case of failure of a secondary replica, the reconfiguration work is small → the primary replica will send one less checkpoint message.
    - In case of failure of the primary replica, the reconfiguration work can be important:
      - To choose a secondary replica to assume the role of primary replica. All replicas must agree.
      - Make sure that the chosen replica has the most recent status possible.
      - Make sure that customers can find the new primary replica properly when they detect that the old primary does not respond.
      - During the reconfiguration, it is very possible that the service is not available.

Replication

- **Replication Schemas: Passive Replication (III)**
  - Advantages over active replication:
    - Only one replica works: more efficient during time without failures.
    - You can replicate services whose implementation is not deterministic → the majority.
    - Distributed mutual exclusion is not necessary
      - Example: you try to use a service that cannot be executed in different nodes at the same time (send an email, access an external service, etc)

Replication

- **Replication Schemes: Active Replication (I)**
  - The name of "active replication" comes from the role of all replicas. All are active, because they all process requests.
  - This scheme is also called **state machine replication**.
    - This name comes from the fact that we assume that all replicas behave in a **deterministic** way
      - Execution model very restrictive and implying in practice that the replicas will not have internal concurrence.
  - It implies the use of "**broadcasting algorithms**" that provide the same sequence of messages to all replicas.
    - These algorithms have a high cost.

Replication

- **Replication Schemes: Active Replication (II)**
  - Reconfiguration in case of failures:
    - In case of failure of a replica it is not necessary a lot of work, just eliminate the references to such replica of the clients that try to access the service.
    - All replicas have the same status, therefore no work is necessary to maintain consistency.
  - Advantages over passive replication:
    - Simple reconfiguration in case of failures.
    - It is not necessary to implement two types of replicas, they all execute the same software.

Availability

Availability in large-scale systems (I)

- In large-scale systems, **partitions** occur.
- By the **CAP** theorem we know that availability or consistency must be sacrificed.
- Many large-scale systems today are designed to provide high availability by reducing consistency.
- Availability is essential for most systems
  - Eg: commercial systems that lose business if they are not available.
- The different partitions will continue to offer service, and therefore they will diverge.
  - Consistency is usually sacrificed.

Availability

Availability in large-scale systems (II)

- The **eventual consistency** is especially relevant
  - Sooner or later, when the partition disappears, the different replicas will need to converge.
  - State **convergence algorithms** must be executed in the replicas when the partition disappears.

- A detailed study of the different operations that modify the state of the replicas is vital to ensure that they can converge when the partition disappears.

Content

- Concept of a Distributed System
  - Definition of a Distributed System
  - Examples
  - Objectives of Distributed Systems
  - Middleware
- Features of Distributed Systems
  - Transparency
  - Availability
  - Scalability
  - Security

Scalability

- Distributed systems must be designed to continue providing service as they grow.
- We say that a system is **scalable** if the service offered does not suffer alterations in performance and availability from the point of view of the user when increasing:
  - The number of users
  - The number of resources
  - The number of nodes
  - The number of simultaneous service requests

Scalability

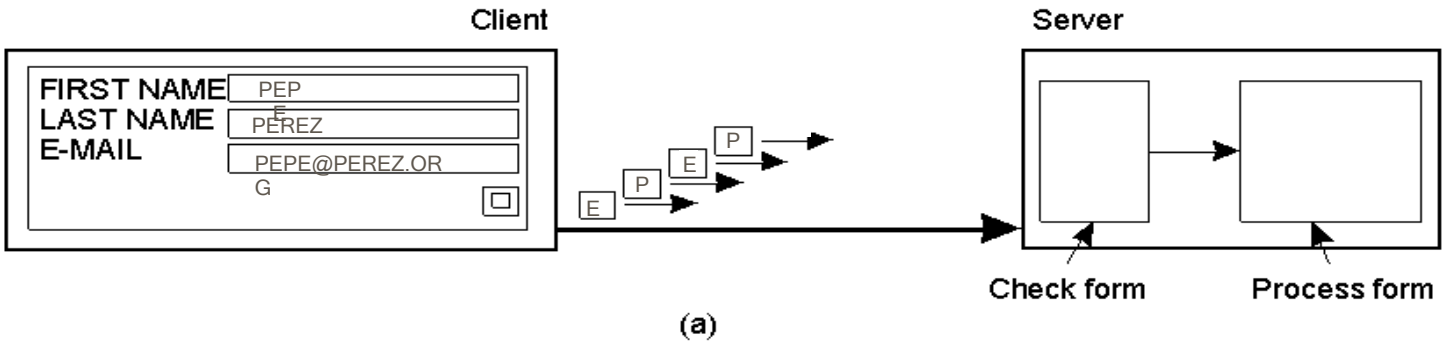
- Each system can have plausible limitations to scalability, which depend on each system.
  - Example
    - When designing a system like "poliformaT", it is not necessary to do it thinking that it will be used by everybody. Its scalability is limited to the scope of the university.
    - However, it must be scalable within its limitations to be able to serve an increasing number of simultaneous requests, increasing amount of resources, increasing number of nodes, increasing number of users.
- Systems with growth objectives on a global scale are called **highly scalable systems.**
  - Examples: Google, Facebook, Netflix, Whatsapp, DNS, email, etc.

Scalability

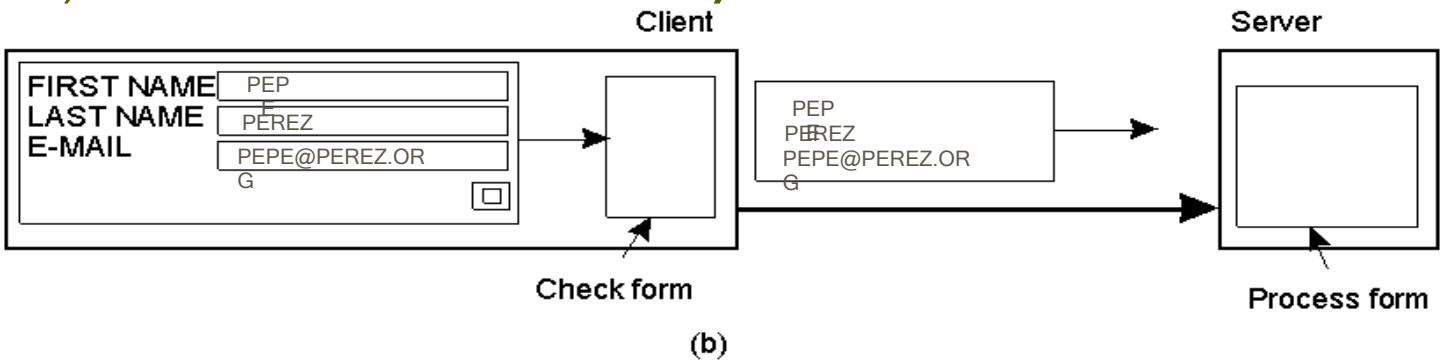
- In general, scalability is threatened when centralized strategies are adopted to manage services, data or algorithms in a distributed system.
- The most important **techniques** to increase scalability involve increasing the distribution by eliminating centralization.
  - **Distribute the load:** distribute the processing performed by the service to different nodes (including customers)
  - **Distribute the data:** distribute the resources in different nodes, so that each node serves a part of the resources of the system.
    - This technique is also known as **data partitioning**
  - **Replication:** Replicate the resources to allow each replicate to attend part of the total requests on the same resource.
    - Load balancers are used to distribute the load between the different replicas.
  - **Caching**: particular case of replication, where we have a copy of the resource in the client.

Scalability

- Example of scalability improvement by load distribution → move part of the calculations made by a Web system to the clients.
- a) The forms are checked by a server

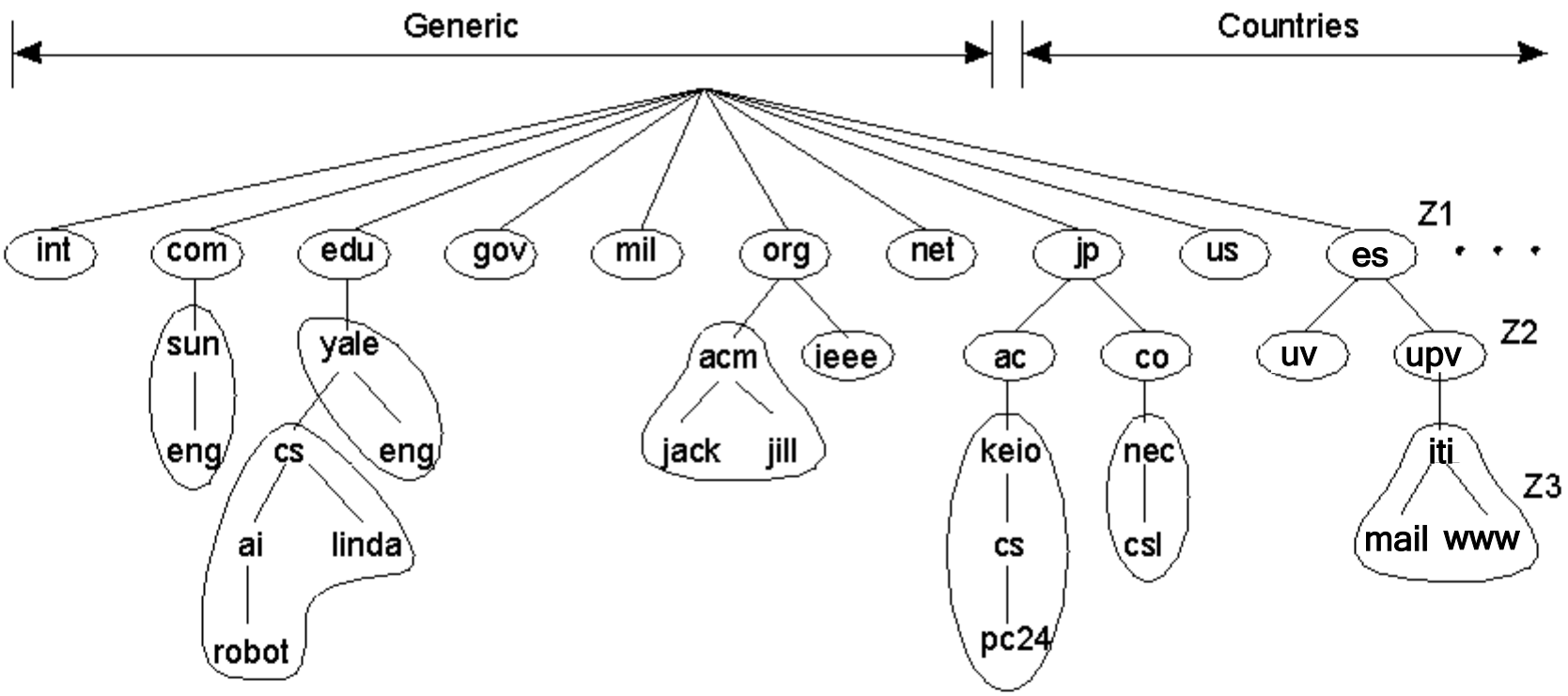


b) The forms are checked by a client



Scalability

- Example of scalability improvement by data distribution → Division of DNS in zones



Scalability

Replication (I)

- Replication is essential in distributed systems to **increase availability**.
- It is also essential to **increase scalability**, especially in highly scalable systems.
- It is very effective in systems where most requests are read-only:
  - Examples: Google, Facebook, Netflix, DNS, peer-to-peer file sharing systems, etc.

Scalability

Replication (II)

- In **highly scalable systems**, where there are data modification operations, consistency is often sacrificed
  - Because it is intended high availability and there may be partitions.
  - Example: the comments that someone enters a Facebook photo, may take time to be visible to other users who visit that profile.
    - This is because different users access different replicas and not all of them are updated at the same time.

Replication (III)

- In systems where there is a majority of writing operations and **strong consistency** is intended, replication does not scale well
- Replicas must be kept mutually consistent and it is necessary to use algorithms that block access to the replicated resource while the changes are being made.
  - In systems that demand strong consistency, it is usually adopted a meticulous **data partitioning**
    - Example: PayPal → the accounts are kept in different nodes. Replication is used to tolerate failures and not so much to increase scalability.

Scalability

- **Caching**
  - It consists of remembering the latest versions of the information accessed in each component of a distributed application.
  - For repeated accesses on the same element, the value of the copy saved locally is obtained, without the need to access the remote service.
  - It is a particular case of replication, where **the same client keeps a replica.**
    - This replica will have a **weak consistency** with respect to the service and, therefore, it will be suitable for those services that do not require strong consistency and for read-only resources.
  - Examples:
    - Web caches.
    - Caches of files.
    - Caches of Facebook photos.

Content

- Concept of a Distributed System
  - Definition of a Distributed System
  - Examples
  - Objectives of Distributed Systems
  - Middleware
- Features of Distributed Systems
  - Transparency
  - Availability
  - Scalability
  - Security

Security

Most security concepts are studied in specific computer security subjects.

- **Every distributed system must offer an available and correct service to the legitimate users of the system and only to them.**
- This fact implies the following general characteristics common to computer security:
  - **Authentication**: users and the requests they make must be properly identified
  - **Integrity**: the system must maintain the data without malicious or unauthorized modifications.
  - **Confidentiality**: only authenticated and authorized users can access certain resources
  - **Availability**: the service should not suffer interruptions in all or part.

Authentication (I)

- The concept of legitimate users implies authentication
  - Users are identified when accessing the service in a certain way, providing **credentials**:
    - User / password pairs
    - Digital certificates
    - Biometrics
    - Cards or smart devices
- Many systems include the "anonymous" user, who does not require authentication as a legitimate user.
  - Example: user searching on Google, or DNS name resolution requests.
- For each legitimate user, the type of requests that can be made to the system must be analyzed and what resources may be used → **authorization**

Authentication (II)

- As the authenticated user makes requests, it must be verified that he/she has permission to do so → **access control.**
- In **highly scalable distributed systems,** access authentication and control mechanisms that are also scalable must be used
  - **Partitioning** the set of users in different authentication servers.
  - **Using replication** of the credentials stored in the system, or of those data that allow to verify the validity of the credentials.
  - Example: LDAP, as distributed authentication scheme.

Security

Integrity

- You must prevent malicious or unauthorized users from making changes to the system.
- In case such malicious modifications are successful, we say that the integrity of the system has been violated.
- Classic mechanisms to **prevent** attacks on integrity:
  - Authentication, authorization and access control: only legitimate users can modify data.
  - Encryption and electronic summaries. Without having the necessary key, the data cannot be modified.
- Advanced mechanisms to **tolerate** attacks on integrity
  - Replication by quorum
    - Attacks on integrity can be seen as the arbitrary functioning of replicas, which will exhibit **Byzantine failures** → Attacks on integrity are treated in the same way as Byzantine failures.

Security

- **Confidentiality**
  - The fundamental classic mechanism for preserving confidentiality is **encryption**.
  - In systems with replication, **attacks on confidentiality are more difficult to protect**, as attackers have more nodes to attack
    - All replicas keep copies of resources, even encrypted.
  - Advanced mechanisms to protect replicated systems:
    - Use of **fragmentation** techniques combined with replication → No node has the complete data, only a fragment. The fragments are replicated.
    - Use of **threshold cryptography** (its study escapes the objective of the subject)→ it is based on decomposing the keys used in encryption in fragments, so that an attacker who gains access to a certain node, finding the keys residing in the node , he can only get part of the key.

Security

Availability

- Attackers may pretend to interrupt the service → denial of service attacks (DoS attack)
- These are very difficult attacks to combat. If the attacker has a large number of nodes to attack from, it may be able to flood the service with requests → **distributed denial of service attacks**.
- Mechanisms to combat attacks on availability:
  - Set the maximum admissible rate of requests from the same remote computer.
  - **Replication**, hiding the location of the different replicas.
    - The greater the number of replicas, the lower the probability that they will all be unavailable.
  - To alleviate the attacks by flooding messages, you must have the collaboration of Internet providers and large nodes that link different networks.

Learning results of the Teaching Unit

- At the end of this unit, the student should be able to:
  - Identify the advantages and problems involved in the development of applications and distributed systems.
  - Characterize distributed systems, compared to centralized, parallel systems, network systems and concurrent systems not distributed.
  - Understand the problem of distribution transparency
  - Identify the different mechanisms necessary to obtain availability in a distributed system
  - Distinguish the different mechanisms to achieve scalability of a distributed system.
  - Know the basic concepts of computer security, especially relevant in distributed systems.

Bibliography

Distributed Systems

- Chapter 7: Francisco Muñoz, Estefanía Argente, Agustín Espinosa, Pablo Galdámez, Ana García-Fornes, Rubén de Juan, Juan Salvador Sendra. ***Concurrencia y Sistemas Distribuidos***. Editorial Universidad Politécnica de València. , ISBN: 978-84-8363-986-3.

- Andrew S. Tanenbaum y Maarten van Steen. **Sistemas Distribuidos: Principios y Paradigmas**. Pearson, 2a edición, 2008. ISBN 9789702612803.

Middleware

- Philip A. Bernstein**. Middleware: A model for Distributed System Services**. Communications of the ACM, Vol. 39(2), pp. 86-98. 1996.