CSD NOTES

UNIT 7.- DISTRIBUTED SYSTEMS BASIC CONCEPTS

1.-CONCEPT OF A DISTRIBUTED SYSTEM

1.1.-DEFINITION OF A DISTRIBUTED SYSTEM

A **distributed system** is a 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**.

The definition of **distributed system** is a set of independent computers that offer their users the image of a single coherent system.

- At **hardware** level: **autonomous machines**.
  - Hardware 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.
- 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.
- 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**.

1.2.-EXAMPLES

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.
- A set of processes within the same computer.
- A concurrent program consisting of several threads, within a single process.

1.3.-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. Advantages:

- Economy of resources: users do not need to have all the resources locally.
- Sharing of resources between users.

1.4.-MIDDLEWARE

Middleware 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.

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**.

Examples of middleware:

- Java Message Service (JMS).
- Java Remote Method Invocation (RMI).
- Domain Name System (DNS).

Java RMI 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 object, RMI dynamically creates an object called a **skeleton**. To invoke a remote object from another process, we use a **proxy** that allows locating the skeleton of the remote object.

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.

2.-FEATURES 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 numbers of nodes.
- **Security**: resources and users must coexist respecting the restrictions and access rules and security policies that will depend on each system.

2.1.-TRANSPARENCY

Distribution transparency is hiding from the user the fact that processes and resources are physically distributed over different computers. Achieving transparency, **simple services** are offered to the user. To achieve transparency of distribution, different specific aspects must be hidden, called **axes of distribution**:

- Location transparency.
- Failure transparency.
- Replication transparency.
- Others.

Location transparency is that the location of the resources is hidden from the user. **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.

Failure transparency is hiding 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. **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.

Replication transparency is hiding the fact that resources are replicated in more than one node. System are replicated to achieve available systems and to increase scalability. **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**.

Other axis of distribution transparency:

- **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.

Providing transparency has a **high cost**. Sometime total transparency in any of the axes does not interest. The convenience depends on each particular system. Sometimes total transparency is impossible to achieve or would have too much cost for a given system.

2.2.-AVAILABILITY

We can define **availability** as the probability that a certain system offers its services to users. The are three factors that affect availability:

- **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**).
- **Maintenance tasks**:
  - Every system require maintenance.
  - Systems must be designed to allow maintenance to be carried out at the same time that users access the services
- **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**).

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 ser of nodes, called **replicas**, in such a way that the failure of a replica does not prevent the service from continuing to function.

Replication introduces the problem of **consistency**, that is the degree of similarity or difference between the different replicas:

- System with a lot of consistency, or **strong consistency**. Ideally all replicas are equal to each other at all times.
- Systems with little consistency, or **weak consistency**. Replicas can diverge, so that each of them can give different answers at a certain moment.

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**, if another node is able to observe the failure, or **undetectable**, otherwise.

Simple detectable failures:

- **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.

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.

Simple undetectable failures, also called **Byzantine failures**, occur when a node fails exhibiting an arbitrary behaviour 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. 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.

Compound failures occur when 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 majority of nodes not affected by failures and the correct majority will be able to continue offering the service.

A type of compound failure that deserves special attention is **partitions**:

- Several failures occur in nodes or communication channels that leave the system divided into two or more subgroups.
- Relevant problem in the current large systems.
- **CAP Theorem**: it is impossible to achieve a system that offers at the same time strong consistency, high availability and may occur partitions.

There exists CA, CP and AP systems:

- **CP**: systems with strong consistency and partitions.
- **CA**: systems with strong consistency and high availability.
- **AP**: systems with high availability and partitions.

Mechanisms to achieve fault tolerance (assuming only simple detectable failures that do not cause partitions):

- Failure detectors.
- Group membership service.
- Replication.

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 emitting "**suspicion of failure**". More than detecting a failure, it is usually indicated that the node **suspects** the failure of another node. In case of suspecting a fault, the **failure detection module** notifies the group membership service.

Group membership service is a 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. 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.

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. We can state this form of work saying: **all detectable simple failures are converted to stop failures**.

In **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 schemes**:

- **Passive replication**: among the group of replicas, a single replica will be the **primary** replica. Th 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.

In **passive 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 responding to the client, there will be a **replicated system with strong consistency**.
- If the checkpoint message is sent to the secondaries later, we will have a **replicated system with weak consistency**.

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.
- 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 us very possible that the service is not available.

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.

In **active replication**, the name comes from the role of all replicas. All are active, because they all process requests. This scheme is also called state machine replication. 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. It implies the use of "**broadcasting algorithms**" that provide the same sequence of messages to all replicas. These algorithms have a high cost.

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.

Advantaged 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.

When it comes to 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. The different partitions will continue to offer service and therefore they will diverge. Consistency is usually sacrificed. 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.

2.3.-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.

Each system can have plausible limitations to scalability, which depend on each system. Systems with growth objectives on a global scale are called **highly scalable systems**. In general, scalability is threatened when centralized strategies are adopted to manage service, 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.
- **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.

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. 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. 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**.

Caching 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.

2.4.-SECURITY

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 is 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. For each legitimate user, the type of requests that can be made to the system must be analysed and what resources may be used (**authorization**). 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 allow to verify the validity of the credentials.

In **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.

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**: it is based on decomposing the keys used in encryption in fragments, so that an attacker who gain access to a certain node, finding the keys residing in the node, he can only get part of the key.

In **availability**, attackers may pretend to interrupt the service. 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.

UNIT 8.- COMMUNICATION

1.-FEAUTURES OF COMMUNICATION MECHANISMS

Communication mechanisms allow communication between processes running on different computers. They are offered by the communication middlewares. Their features are:

- Usage.
- Structure and content of messages.
- Addressing.
- Synchrony.
- Persistence.

1.1.-USAGE

There are two types of usages:

- Using **basic communication primitives**: **send** and **receive** operations.
- Through **constructions of the programming language**: it has higher level of abstraction and sending and receiving messages is transparent to the programmer.

1.2.-STRUCTURE AND CONTENT OF MESSAGES

There exist three types of **message structures**:

- **Non structures, only content**: free-form content.
- **Structured in header + content**: the header is a set of fields generally extensible and has free-form content.
- **Structured transparent to programmer**: determined by the communication middleware.

There are two types of **message content**:

- **Bytes**:
  - Advantages: efficient and compact.
  - Disadvantages: difficult to process, binary representation is not equal among architectures and programming languages.
- **Text**:
  - Advantages: independent of architecture and programming language.
  - Disadvantages: less efficient.
  - A language with high availability of libraries for its processing is normally used.

1.3.-ADDRESSING

There exist two types of **addressing**:

- **Direct addressing**: the sender computer sends messages **directly** to the receiver computer.
- **Indirect addressing**: the sender computer sends messages to an **intermediate** (broker), who is in charge of sending them to the receiver computer.

1.4.-SYNCHRONY

There are three types of **synchrony**:

- **Asynchronous communication**: the middleware responds to the sender with the confirmation, after storing the message in its buffers.
- **Synchronous communication (delivery)**: the middleware responds when the receiver has confirmed the successful delivery of the message.
- **Synchronous communication (answer)**: the middleware responds when the receiver has notified to have processed the message.

1.5.-PERSISTENCE

There exist two types of persistence:

- **Persistent communication**: the middleware can store delivery pending messages. Receiver does not need to be running when sender sends the message. Sender can finish its execution before the message is delivered.
- **Non-persistent communication**: the middleware is not able to keep the messages to be transmitted. Sender and receiver must be active to let messages be transmitted.

2.-REMOTE OBJECT INVOCATION (ROI)

A **remote object** is an object that can receive invocations from other address spaces. Any remote object is instantiated in a server and can answer to invocations from both local or remote clients. Application are organized as a dynamic collection of objects that might be in different nodes. Both applications and objects can invoke methods of other objects in a remote way. We assume here that any object is totally hols in a single node.

Advantages of the **Distributed Object Mode**l:

- To make profit of expressivity, abstraction capacity and flexibility of the **OO paradigm**.
- **Location** transparency.
- You can **reuse** legacy applications “encapsulating” them into objects.
- **Scalability**.

There exist two types of invocations:

- **Local invocation**: both invoker and invoked objects are located in the same process.

- **Remote invocation (ROI)**: invoker and invoked objects are in different processes, which can be inside the same node or in different nodes.

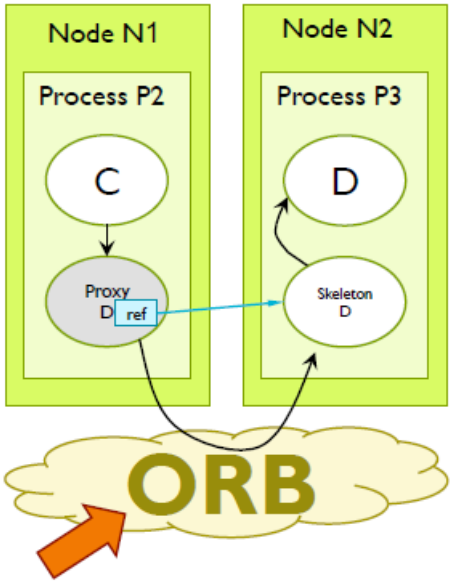
2.1.-ROI ELEMENTS

The **proxy** provides the **same interface** as the remote object. It contains a **reference** to the remote objects and provides access to the remote object and its interface. It is created at runtime when the remote object is accessed for the first time.

The **skeleton receives requests** from clients and performs the **actual calls** to the remote object methods. It is created at run-time when the remote object is created.

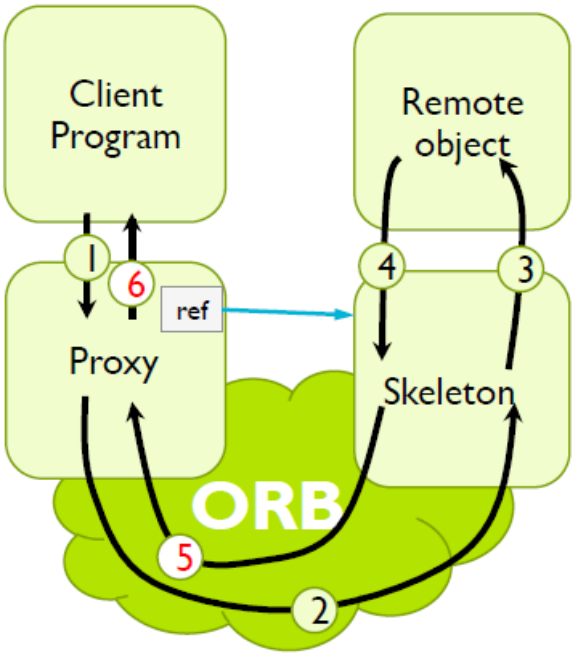
The **Object Request Broker (ORB)** is the main component of an object-oriented middleware. It is charge of:

- **Identifying and locating** the objects.
- **Carrying out the remote invocations** from proxies to skeletons.
- **Managing the object life-cycle**.



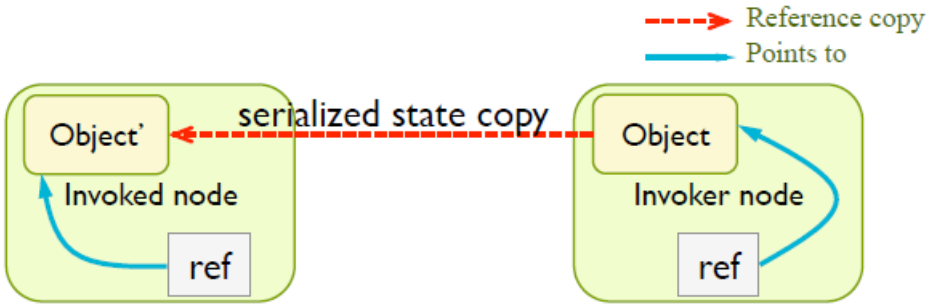
2.2.-ROI STEPS

1. The client process invokes the method on the local proxy related to the remote object.
2. The proxy marshalls the arguments and, using the object reference, call the ORB. The ORB manages the invocation, making the message arrive to the skeleton.
3. The skeleton unmarshalls the arguments, sends the invocation to the requested method, and waits until the method is completed.
4. The invoked method is completed, and the skeleton is unlocked.
5. The skeleton marshalls the result and calls the ORB who makes the message arrive to the proxy.
6. The proxy unmarshalls the results and returns them to the client process.

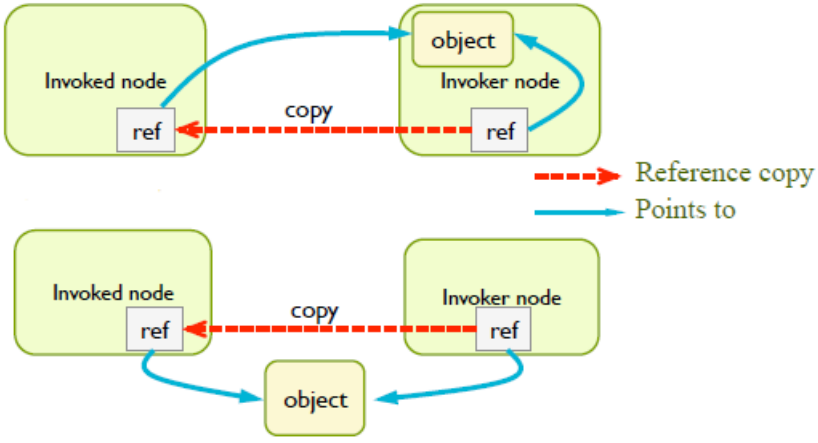


2.3.-PASSING OBJECTS AS ARGUMENTS

When **passing by value**, the state of the "original object" is packaged, by means of a process called **serialization**. The serialized object is transmitted to the destination node, where it is used to create a new copy of the original object. Both objects evolve separately.



When **passing by reference**, it's enough to copy the reference from the invoker to the invoked node. It does not matter that the object belongs to the invoker node or to a third node.



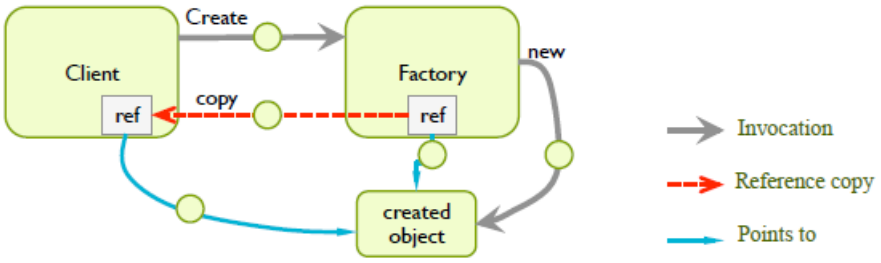
2.4.-CREATING OBJECTS

In ROI, object creation and its registry in the ORB can be done by two different ways:

- By **initiative of the client**: the client requests a factory to create the object. A **factory** is a server object that creates objects of a specific type.
- By **initiative of the server**: a server process creates the object and registers it in the ORB.

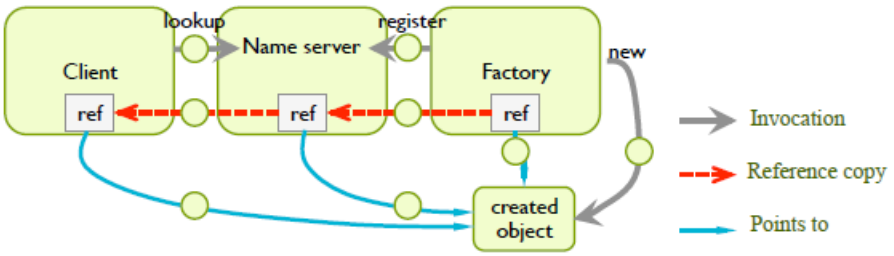
The steps in **initiative of the client** are:

1. The client requests a factory to create the object.
2. The factory creates the requested object and registers it in the ORB. As a result, it obtains a reference to the object and a skeleton.
3. The factory returns a copy of this reference to the client.



The steps in initiative of the server are:

1. A process, which will become the server, creates the object and registers it in the ORB, who creates its first reference.
2. The server process uses the reference to register it in a **name server**, giving a specific text string as name.
3. Any other process that knows the name used for registering the object can contact the name server and obtain a reference of this object.



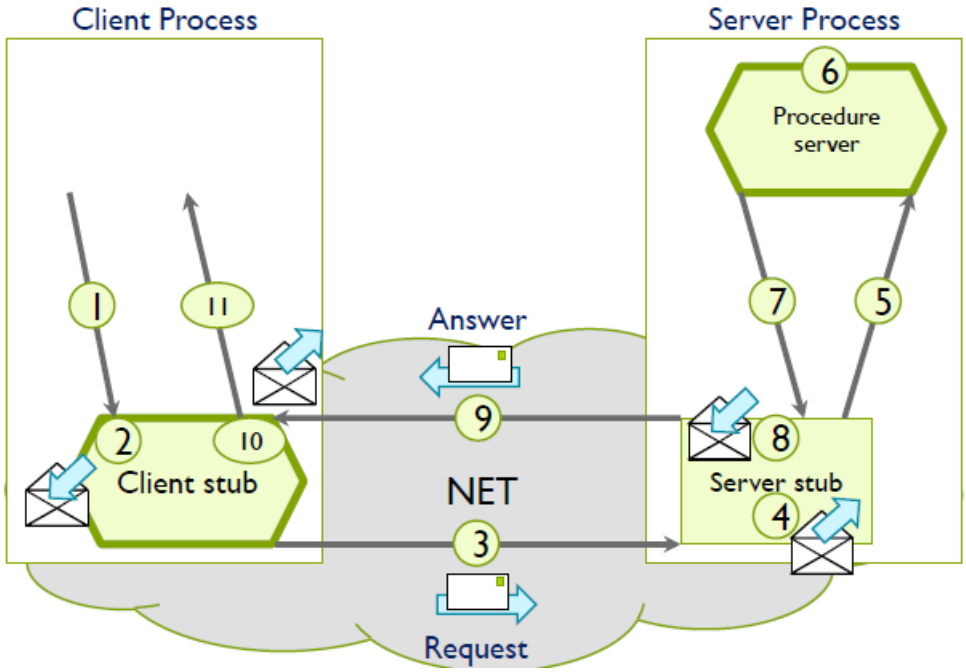
2.5.-REMOTE PROCEDURE CALL

RPC is the precursor of ROI. It has the same goals: call a "remote procedure or function" similarly as being a local call. It doesn't use objects, but procedures. The remote node offers a catalogue of procedures. These procedures can be called in a transparent way from the client nodes. This mechanism is implemented by the **client stub** (equivalent to a proxy of ROI) and the **server stub** (equivalent to a skeleton of ROI).

Steps:

1. Invocation to local procedure.
2. Marshalling of input parameters into a message.
3. Send the message to server and wait for the answer.

4. Unmarshalling of the message and extract the input parameters.
5. Call the procedure.
6. Execute the procedure.
7. Control return to the server stub.
8. Marshalling of the output parameters and result into a message.
9. Send the answer message.
10. Unmarshalling the message and extract the output parameters and result.
11. Return the control to the code that invoked the local procedure.



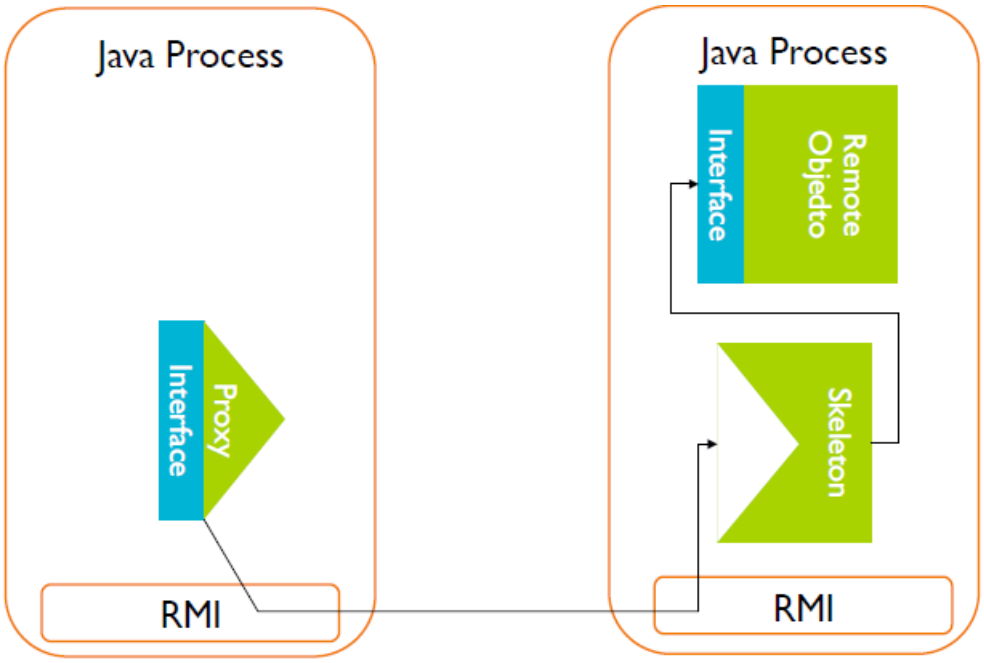
2.6.-JAVA RMI

Java RMI (Remote Method Invocation) is an **object-oriented** communication **middleware** that provides a solution for a specific OO language (Java) which gives support to portability. It is not multi-language, but it is **multi-platform**. It allows **invoking** Java object methods of another JVM and **passing Java objects** as arguments when invoking these methods.

The RMI component is automatically incorporated into a Java process when its API is used. It listens to requests that arrive in a TCP port.

An object is invokable remotely if you implement an interface that extends the Remote interface. For each of these remote objects, RMI dynamically creates an object called a skeleton.

To invoke a remote object from another process, a proxy object is used. Its interface is identical to that one of the remote object. It contains a reference to the remote object (IP address + port + which object).



The name server stores for each object: **symbolic name + reference**. It can be hold in any node and can be accessed by both the client and the server using a local interface named **Registry**. In Java Oracle distributions, the name server is launched using the **rmiregistry** command.

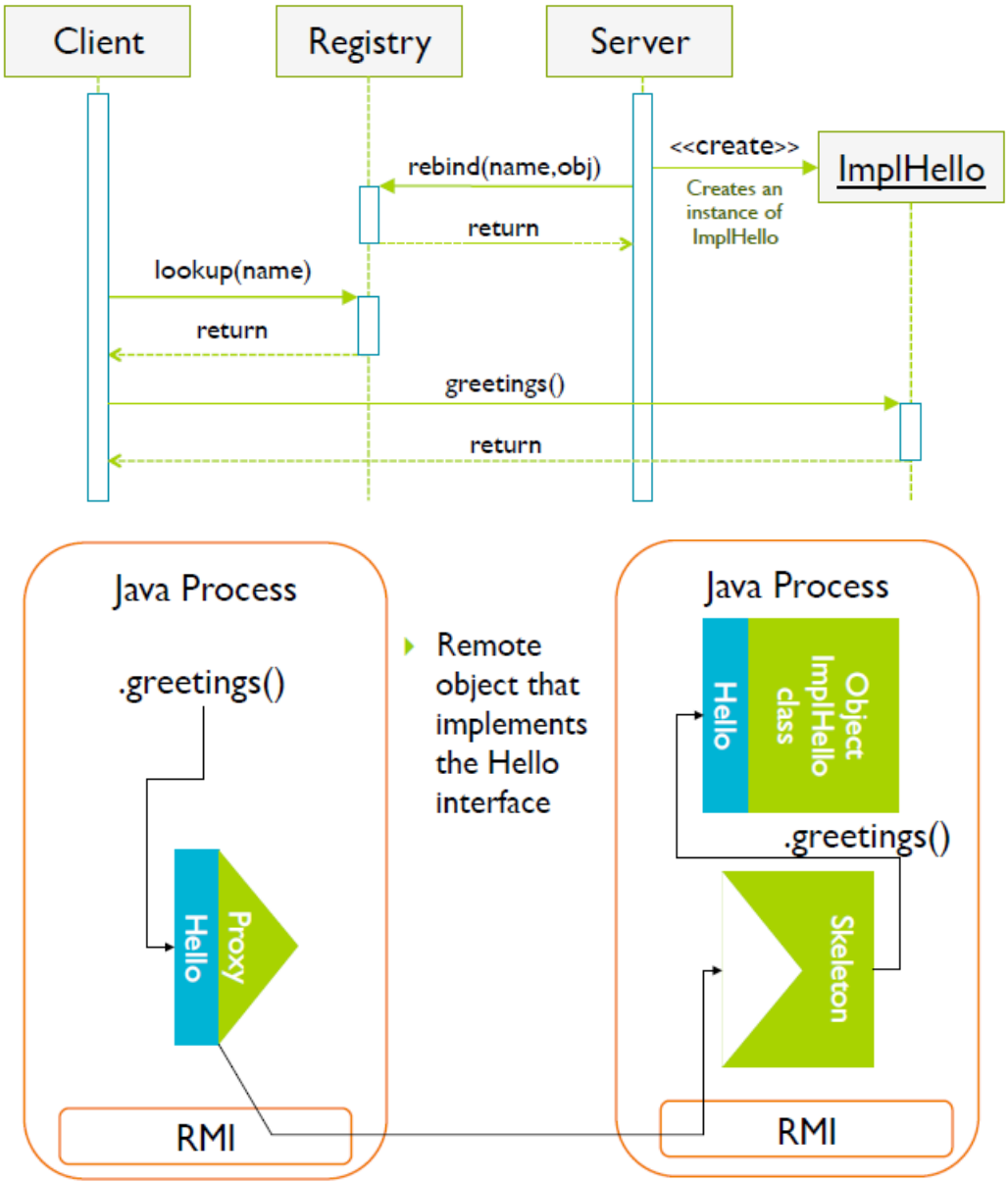
Rules for programming remote objects in Java:

- **Remote Object Interface**:
  - The interface of the remote object must extend **java.rmi.Remote**.
  - Methods of this interface must indicate that they can generate the **RemoteException** exception.
  - From the interface definition, the Java compiler generates proxies and skeletons.
- **Remote Object Class**:
  - The class of the remote object must **implement** the remote interface and **extend** java.rmi.server.UnicastRemoteObject.
  - This allows registering the objects in the Java ORB.

Steps in the execution of this example:

```
// SERVER
    ...
Registry reg = LocateRegistry.getRegistry(host, port);
reg.rebind("objectHello", new ImplHello());
System.out.println("Hello Server prepared ");

// CLIENT
...
Registry reg = LocateRegistry.getRegistry(host, port);
Hello h = (Hello) reg.lookup("objectHello");
System.out.println(h.greetings());
```



When invoking a method, we can pass objects as arguments. If the object passed as argument implements the **Remote** interface, it is passed by reference and is shared by previous and new references. If the object passed as argument **does not** implement the **Remote** interface, it is **serialized** and passed by value and an object in the final virtual machine is created totally independent from the original one.

Features of Java RMI communication:

- Usage: calls to remote methods making transparent to the programmer the usage of basic primitives for communication.
- Structure and content of messages: determined by the Java compiler, transparent to the programmer.
- Addressing: direct to the computer where the remote object is.
- Synchrony: synchronous in answer. It is blocked till the remote object finishes.
- Persistence: non-persistent. The remote object must be active.

3.-WEB SERVICES

3.1.-GENERAL CONCEPTS

Web services are software systems designed to provide **computer-computer interactions** using the network. They are usually based on a **client-server** architecture implemented on the **HTTP** protocol. Web pages are not requested, but consultations and actions. There are many variants, the most representative ones are:

- Web services based on SOAP and WSDL:
  - Generally known as “web services”.
  - SOAP (Simple Object Access Protocol): XML specification of the information that is exchanged.
  - WSDL (Web Services Description Language): XML specification that describes the functionality offered by a web service.
- RESTful web services:
  - Simpler and more flexible alternative, which has made them very popular nowadays, largely replacing “classic” web services.
  - JSON (JavaScript Object Notation) is the most common format for the exchange of information.
  - It does not require any functionality description language.

The alternatives to use web services are:

- Construction and direct processing of the content of HTTP messages.
- Automatic generation of code provided by frameworks for development and deployment of web services, both for the client side and the server side, based on the definition of the service.

3.2.-RESTFUL WEB SERVICES

RESTful web services are one of the **most important technologies** for developing web applications. They are **available** in the vast majority of programming languages and development frameworks. Much of its success is due to its **ease of use**. It is **not** a strict standard. It is focused on **resources**. They are services **without state**. **URLs** are used for referring to resources. It has free resource representation, the most common ones are XML and JSON. The operation in REST are HTTP methods to tell the server the type of operator to be done:

Method	Operation in the server
GET	Read a resource
POST	Create a resource
PUT	Modify a resource
DELETE	Delete a resource
…	…

The response indicates how the service call ended by using HTTP status codes:

HTTP Code	Typical meaning
200	Everything ok
201	Resource has been created
400	Wrong request
404	Resource not found
500	Failure in the service provider

The URLs can include parameters for:

- Making queries.
- Paging the answers.
- Providing authentication information.

REST is a communication mechanism difficult to be categorized depending on the point of view used for analysis:

- First approach: from its internal mechanism point of view, its features are related to a communication based on the HTTP protocol, and thus, it uses TCP sockets.
- Second approach: from its usage point of view, it is basically a request-answer mechanism, similar to RPC and ROI in many aspects.

Features of REST communication:

- Usage:
  - First approach: basic primitives of sending and receiving.
  - Second approach: high-level API provided by the service provider.
- Structure and content of messages:
  - First approach: content codified with HTTP, XML, JSON.
  - Second approach: content hidden by the API provided by the service provider.
- Addressing: direct by means of requests to the computer supporting the service.
- Synchrony
  - First approach: synchronous in delivery.
  - Second approach: synchronous in answer.
- Persistence: non-persistent.

4.-MESSAGE ORIENTED MIDDLEWARES

4.1.-GENERAL CONCEPTS

Message Oriented Middlewares (MOMs) are middlewares that offer **message-based** communication. Senders send messages not directly to receivers, but to an intermediate element usually called **queue**:

- Once the message has been deposited in the queue, the sender continues with its execution, without waiting for the recipient to collect the message.
- The receiver collects the message at any time.
- The communication is therefore **asynchronous**.

Most of these systems are based on a communication **broker**, that is, a server process that fully manages the queues:

- Creation and deletion of queues.
- It handles senders' deliveries and deposits the messages in the indicated queue.
- It handles receivers' requests to collect messages from the indicated queue.
- It keeps pending delivery messages in the queues, although senders and receivers are not running, not even the broker itself, thus offering persistence.

There is an exception, ZeroMQ, that it does not require the existence of the broker.

Advantages:

- They allow highly decoupled components of the distributed system.
- They can offer a high degree of availability through active replicas of brokers.
- It is possible to achieve a high degree of security by being centralized systems.

Drawbacks:

- Lower performance, by introducing the broker as an intermediate element in communications.
- Difficult scalability, again due to the existence of the broker.
- Lack of standardization. There are open protocols but many of the most used implementations are property.

4.2.-JAVA MESSAGE SERVICE

Java Message Service (JMS) is a **Java API** that enables applications to send and receive messages. It offers a **loosely coupled communication**:

- The sender sends messages to a destiny and the receiver receives messages from this destiny.
- Sender and receiver do not need to know each other, they just need to agree on the format of message content.

It is interesting to use JMS when:

- You do not want that components of an application have to know the interfaces of other components.
- It is not necessary that all components are running simultaneously.
- The components, after sending a message, do not need to receive an immediate response to continue operating normally.

JMS components:

- JMS providers: messaging system that implements the JMS interfaces and provides administrative and control tools.

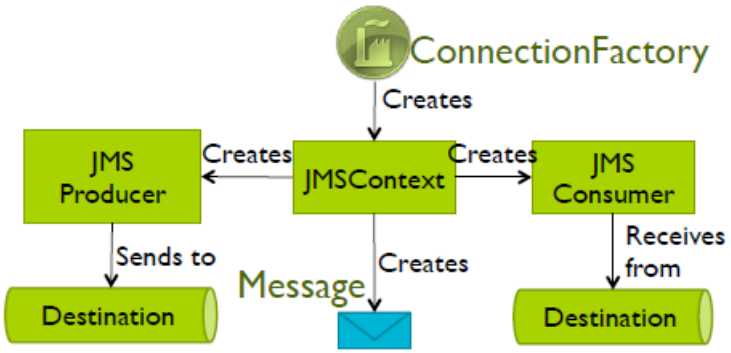
- JMS clients: program or component written in Java that produces or consumes messages.
- Messages: objects that communicate information between JMS clients.
- Administered objects: preconfigured objects created by an administrator for the use of clients. There are two kinds: connection factories and destinations (queues, for delivering to one single client, and topics, for delivering to multiples clients).

Message structure: header + properties + body.

Connection factories are created by the administrative tools of the JMS provider. They are used to create connections to the clients of the messaging system.

Elements of the programming model:

- **Connection Factory Interface**: it links the application with an administered object and creates connections with the JMS provider.
- **JMS Context Interface**: it maintains a connection with the JMS provider. They are usable by a single thread of execution and process sendings and receivings sequentially. An application can use several JMS Context objects if it requires of concurrent processing.
- **JMS Producer Interface**: they enable sending messages to queues and topics.
- **JMS Consumer Interface**: enable receiving messages from queues and topics.
- **Destination Interface**: it links the application with an administered object and encapsulates a specific JMS provider address. Subinterfaces: queue and topic.
- **Message Interface**: they are the messages that are sent and received in JMS.



Features:

- Usage: basic primitives for sending and receiving.
- Structure and content of messages: header, properties, body.
- Addressing: indirect through JMS provider.
- Synchrony: asynchronous. The sender keeps on running when it delivers the message in secondary storage.
- Persistence: persistent. Even if the JMS provider stops, because it stores the message in secondary storage.

UNIT 9.- DISTRIBUTED ALGORITHMS

1.-INTRODUCTION

Fundamental Algorithmic Concepts:

- Independent of particular technologies.
- Characteristics of distributes **decentralized** algorithms:
    1. No node has all the complete system information.
    2. They run on independent processors (nodes). The failure of a node does not prevent the algorithm from progressing.
    3. They make decisions based on local information.
    4. There is no precise source of global time.

2.-GLOBAL STATES AND TIME

2.1.-CLOCKS, EVENTS AND STATES

Synchronization provides mechanisms for coordinating activities. It is **more complex** in distributed systems than in centralized ones.

Synchronising physical clocks:

- Each node i has a local clock Ci. It represents a universal temporal coordinated (UTC) value.
- Given any real instant t. The goal is that all nodes have Ci(t) = t.
- **Problem**: clock chips in which Ci clocks are based are not absolutely precise.

International Atomic Time (TAI): time established by the Bureau Internationale de l'heure (BIH) from all atomic clocks of all laboratories around the world.

Universal Time: units of time with astronomical origin, depending on the periods of rotation and translation of the Earth around the Sun. The rotation period is gradually lengthening.

The problem is that the **atomic time** and the **universal time** tend to desynchronize.

Synchronising physical clocks (2):

- We can synchronize each node with the UTC time.
- The **solution** to differences in time is to **periodically synchronize clocks** of all nodes using some reliable time source.

2.2.-CLOCK SYNCHRONIZATION ALGORITHMS: CRISTIAN, BERKELEY

Cristian Algorithm synchronizes the local clock **Cc** of a **client** computer with the local clock **Cs** of a **server** computer. Considerations:

- The server computer has a **very precise clock**, possibly synchronized with others that are still more precise.
- Clocks must not go back.

- Synchronization implies message transmission, but transmission of messages through the network takes time.

Cristian Algorithm steps:

1. The **client** asks the value of the clock to the server at instant **T0** (this instant is according to the client local clock Cc)
2. The **server** receives the request and answers with the value of its own clock: **Cs**.
3. The answer arrives to the **client** at **T1** (according to the client local clock **Cc**).
4. The client sets its clock to the value **C = Cs + (T1 – T0)/2**.
5. The **client** updates its clock according to:
    a. If **C > Cc**, the **client clock is set to Cc = C**.
    b. If **C < Cc**, the client stops Cc the next **Cc – C** units of time.

Features of **Cristian Algorithm**:

- It assumes that the sending time of both messages (request and answer) is practically the same. In T1 the value of Cs will have been incremented in (T1 – T0)/2.
- If one of the two messages take more time to be transmitted, then this adjustment is not right. Normally, the duration of both messages is the same.
- A perfect synchrony is nearly impossible.
- The right behaviour of a distributed application can be dependent on the values of local clocks only if it can tolerate the error margin inherent to the synchronization algorithm applied.

Features of **Berkeley Algorithm**:

- There is a set of nodes composed of a server, named S, and N clients, named Ci. This algorithm does not assume any computer with a precise clock. But one of the computers of the distributed system will act as a **server** and the others as its **clients**.
- Each node has its own local clock.
- The goal is to synchronize the local clocks of all nodes between them.
- Periodically, at server initiative, all nodes synchronize their clocks.

Berkeley Algorithm:

1. The **server** periodically broadcasts the value of its clock.
2. Each **client** calculates the difference **Di** between its local clock and the clock value notified by the server in its message.
3. Each client notifies the difference Di to the server.
4. Given the replies, the **server**:
    a. Computes the average difference (including the server). **Di' = Di – (T1i- T0)/2**. **D =** $\sum Di'$ **/ (N +1)**.
    b. Updates its own clock, incrementing it in **D** units.
    c. Answers to every client with the difference that each of them has to apply in order to update its local clock. **Ai = D – Di'**.

Additional **considerations**:

- It does not aim to synchronize all the clocks with the "real" instant, but to reach an agreement between the nodes.
- If any **Di** difference is very different from the others, it is not taken into account.

- If the server fails, a leader election algorithm is started to choose another server.
- Exact synchronisation is impossible due to the variability of time in the transmission of messages.

2.3.-LOGICAL CLOCKS AND VECTOR CLOCKS

The **logical clocks** (Lamport,1978) indicate the order in which certain events occur, not the actual moment in which they happen. They are useful for many types of distributed applications that only require to know if an event has happened before another. Unlike physical clocks, their timing is perfect, but they have certain limitations. Key ideas:

- If two nodes do not interact (if they do not exchange messages), it is not necessary that they have the same clock value.
- In general, it is important the global order in which events happen, but not the actual moment when they happen.

To synchronise logical clocks, Lamport defined the relation "happens before". When a -> b or "a happens before b", it means that all the nodes agree that first they see event **a** and afterwards they see event **b**. This relation can be directly observed in two situations:

- If **a** and **b** are events on the same node and **a** occurs before **b**, then **a->b** is true.
- If **a** is the event of sending a message m by a node, and **b** is the event of reception of m by another node, then **a->b** is true.

It is a **transitive** relationship. Two events **x** and **y** are **concurrent** if we can not say which of them happens before the other: **x||y**. The happens-before relation establishes a **partial order** among events in a system. Not all events are related between the, since there can be concurrent events.

It is necessary to have a way for measuring the happens-before relationship: using Lamport's logical clock. Logical clocks must mark the instant in which events occur, so that they associate a value to each event. We call **C(a)** to the value of the logical clock for event **a**. The logical clocks have to satisfy that, if **a->b**, then **C(a) < C(b)**. Additionally, the clock must never decrease.

Lamport Algorithm, every node has a counter (logical clock) **Cp** initialized to 0:

1. Each execution of an event in a node **p** increases the value of its counter **Cp** in 1.
2. Each message **m** sent by anode **p** is labelled (**Cm**) with the value of its counter **Cp**, **Cm = Cp**.
3. When a node **p** receives a message, it updates its clock as follows: **Cp = max(Cp, Cm) + 1**.

Features:

- They stablish a partial order among events. Concurrent events are not ordered between them. But you can obtain a total order by adding the node identifier as suffix.
- If **a->b**, the algorithm ensures that **C(a) < C(b)**.
- If **C(a) < C(b)** we cannot decide whether **a->b** or **a||b**.

Using **vector clocks,** we can know when events are concurrent. Vector clocks link an array value **V(x)** to each **event x** of a distributed system. They help determining whether an event **happens**

before another or if two events are **concurrent**. Given **N** nodes, each node **p** maintains an array of **N** temporal marks **Vp** such as:

- **Vp[p]** is the number of events that have occurred in p.
- If **Vp[q] = k**, then **p** knows there have been at least **k** events in node **q**.

Vector clock's algorithm, each node **p** has a vector clock **Vp** initialized with all its elements to 0:

1. Node **p** increments **Vp[p]** in 1 unit each time it sends a message or executes an internal event.
2. A message **m** sent by a node **p** carries its vector clock. That is, **Vm = Vp**.
3. When **p** receives a message m, it increments **Vp[p]** in 1 and it updates its vector clock selecting the maximum value between its local value and the value of the clock at the message, for each of its components.

Features:

- **V(a) < V(b)** if all the components of V(a) is lower or equal to the respective component of V(b) and moreover there is at least one component that is strictly **lower**.
- If **a->b**, then **V(a) < V(b)**.
- If **V(a) < V(b)** then **a->b**.
- If **V(a) < V(b) does not hold** and **V(b) < V(a) does not hold**, then **a||b**.

2.4.-GLOBAL STATES

The **global state** is the local state of each process plus the state of each communication channel:

- State of each process: state of the variables that interest us.
- State of each communication channel: messages sent and not yet delivered.

Preliminary concepts:

- A distributed snapshot reflects a state that could have occurred in the system.
- The snapshots reflect only consistent states.
- The notion of "global state" is reflected by the concept of **cut**.

The **precise snapshot** is not factible. For a precise snapshot it is required that all nodes have their clocks perfectly synchronized. The snapshot is **consistent** when it does not show any message delivery without its respective sending. The snapshot is **inconsistent** when it shows the delivery of some messages but not its sending.

The **Chandy-Lamport algorithm** (1985) creates a consistent snapshot of the global state of a distributed system. We assume that:

- The distributed system is formed by several nodes.
- Network with a complete topology, there is a channel between every pair of nodes.
- All channels are reliable. They transmit their messages in FIFO order. They are unidirectionals, between two nodes **p** and **q** there are two channels (**p, q**) and (**q, p**).

Chandy-Lamport Algorithm:

1. Initiator node p stores its local state, then it sends a MARK message to the rest of nodes.

2. When a node p receives the MARK message through channel c:
    a. If it has not stored its local state yet:
        i. It stores its local state and sets this channel c as empty.
        ii. Then, previous to any other message, it sends MARK to all other nodes and starts registering all the messages that arrive from the other channels different to c.
    b. If it has already stored its local state:
        i. It stores all messages received by this channel c and stops registering the activity if that channel.
3. When a node p has received MARK by all its incoming channels. It sends its local state previously stored and the state of its incoming channels to the initiator node (except if ii is already) and finishes its participation in the algorithm.

Additional considerations:

- The algorithm can work as described here if you add the identifier of the initiator node to the MARK message. Then nodes that receive MARK can know who the initiator is and can later send it the information with its registered state and received messages.
- In the algorithm described, the authors do not specify how the recorded information by each node has to be collected.

3.-EXAMPLES OF ALGORITHMIC PROBLEMS IN DISTRIBUTED SYSTEMS

3.1.-MUTUAL EXCLUSION

The problem to solve is the access to a critical section of several processes that are in different nodes:

- Avoid inconsistences.
- Solutions need to ensure access with mutual exclusion by processes.

Proposed solutions:

- Centralized algorithm.
- Distributed algorithm.
- Algorithm for rings.

The correction conditions are:

- **Security**: at most, only one process can be running within the critical section at a given time.
- **Liveness**: any process that wants to enter the critical section achieves it at some point.
    - **Progress**: if the critical section is free and there are processes that wish to enter, one of them is selected in finite time.
    - **Bounded waiting**: if a process wants to enter the critical section, it should only wait a finite number of times for others to enter before it enters.

Centralized algorithm (Lamport 1978), a node is selected as coordinator (leader), this leader will control the access to the critical section:

1. When a node wants to enter into the critical section, it sends a REQUEST message to the leader, asking for permission.
2. If no other node is in the critical section, the leader replies OK.
    a. The leader registers that now the critical section is occupied.
    b. The requesting node begins using the critical section.
3. If the critical section is occupied, and another node requests the leader to use it.
    a. The leader registers the identifier of the node requesting to access the critical section and does not answer him.
    b. The requesting node remains blocked waiting for the answer.
4. When a node leaves its critical section, notifies the leader with a RELEASE message.
    a. If there is any node blocked waiting for a permit to go inside the critical section, the leader sends OK to this node.
    b. If there are several nodes waiting, the leader can select.
    c. If there is not any node waiting for accessing the critical section, then the leader registers that the critical section is now free.

Distributed algorithm (Ricart-Agrawala 1981), we suppose that all events are ordered:

1. When a process wants to enter the critical section, it broadcasts a TRY message to all process receives a TRY message:
2. When a process receives a TRY message:
    a. If it is not in its critical section neither wants to enter it, then it replies with OK.
    b. If it is in its critical section, it does not answer and queues the request.
    c. If it is not in its critical section, but it wants to enter, it compares the mark of the incoming message with the one it sent to the rest of processes. The lowest number wins:
        i. If the incoming message is lower, it answers OK.
        ii. If its higher, it does not answer and queues the message.
3. A process enters the critical section when it receives OK from all.
4. When it leaves the critical section, it sends OK to all the processes that sent the message retained in its queue.

Features:

- Message are labelled with the Lamport's logical clock plus the ID of the sender node.
- When a node receives a TRY message, and he is not in its critical section, but he wants to enter it:
    - If the label of the incoming message is lower than the label of the message he sent, then he answers OK.
    - If it is higher, then he does not answer and stores the identity of the sender of the TRY message.

In the **algorithm for rings** (Le Lann 1977), there is not any coordinator node. Coordinator is solved by ring communication and the use of a token that circulates through the ring:

- Requires reliable communication.
- If a node falls down with the token, then a new token must be built.

Only the node with the token can be inside the critical section.

Algorithm for rings:

Initial situation: the critical section is free and there is one single token in the ring, in one of the nodes.

1. If the node with the token does not want to go inside its critical section, then it sends the token to the next node.
2. A node waits to go inside its critical section if it does not have the token.
3. A node goes inside its critical section when it gets the token. And it does not pass the token to next node until it finishes its critical section.

Therefore, if a node wants to enter its critical section, it has to wait that the critical section gets free and that it obtains the token.

Comparative:

Algorithm	Messages to enter or leave	Delay before entry	Problems
Centralized	3	2	Coordinator crash
Distributed	2(n - 1)	2(n - 1)	Crash of any process
Ring	1 to Infinite	0 to n – 1	Lost of token

3.2.-LEADER ELECTION

The leader election is done at the beginning or when the nodes detect that the leader does not answer any more. It is assumed that all nodes know the identifiers of the other nodes of the distributed system. Leader election is a articular case of consensus. Examples:

- Bully algorithm.
- Algorithm for rings.

The **Bully algorithm** (García-Molina 1982) is used to select a new leader. It is started by one of the nodes of the distributed system. The new leader will be the active node with the highest identifier. Once selected, the leader will notify the others that he is the leader.

Bully algorithm:

1. When a node wants to start an election, he sends ELECTION to all nodes with identifier (ID) higher than its own.
2. When a node receives ELECTION, sends OK to whom sent it.
3. When a node receives at least one OK message, he stops participating in the process.
4. The previous steps are repeated until a node does not receive any answer. Then he is the new leader.
5. The new leader broadcasts a COORFDINATOR message to all the other nodes, to let them know that he is now the new leader.

In the **algorithm for ring**, the nodes are located in a logical ring and they send messages through the channels of this ring. This algorithm is used to select a new leader, as initiative of one of the nodes of the ring. Once selected, the identify of the new leader will be propagated through the ring.

Algorithm for rings:

1. When a node wants to begin a leader election:
    a. He builds an ELECTION message with two fields:
        i. Initiator.
        ii. List of participant nodes.
    b. It assigns its identifier to both fields and sends the message to the next node in the ring.
2. When a node receives an ELECTION message, if he is not the initiator:
    a. It answers the sender node with an OK message.
    b. He inserts itself in the list of participant nodes.
    c. He sends the message to next node.
    d. If no confirmation arrives, then it resends the message to the following node of the ring.
3. When a node receives an ELECTION message and he is the initiator:
    a. He selects as leader the highest identifier of list of participant nodes.
    b. He builds a COORDINATOR message including there the identifier of the new leader.
    c. He broadcasts the message through the ring.

3.3.-CONSENSUS

Consensus: we define the problem as the agreement that the participating nodes must reach in the value of a variable.

Consensus Problem Definition:

- N nodes try to agree on the value of certain **Variable V**.
- All nodes receive the command STAR at approximately the same time. In other words, we start from a situation in which we assume that all nodes want to participate in the consensus when starting to execute their algorithms.
- Each node "i" has an initial estimation of the variable: **estimate (Vi)**.
- After executing the consensus algorithm, all must provide **decision (Vj)** as output, with "j" being the node that proposed the estimated value.
- It is not necessary to know "j", but it is necessary that "j" exists. The agreement must have arisen by adopting the estimate of some node. It is not a consensus to decide a value "derived" or generated from the estimates. In these cases, they are problems different from consensus.

Correction conditions for consensus:

- **Termination**: every correct node eventually decides some value.
- **Uniform integrity**: every node decides at most once.
- **Agreement**: no two correct nodes decide differently.
- **Uniform validity**: if a node decides **v**, then **v** was proposed by some node.

Distributed algorithm (simplified):

- We assume N nodes in a totally connected network.

- Of the N nodes, some nodes might be “turned off”, because they have previously failed, but it is unknown which ones.
- We assume that during the execution of the algorithm, no node fails.

4.-ALGORITHMS IN THE PRESENCE OF FAILURES

Distributed Consensus Algorithms considering failures:

- We assume N nodes in a totally connected network.
- Of the N nodes, some nodes might be “turned off”, because they have previously failed, but it is unknown which ones.
- We assume that **during the execution of the algorithm**, the **nodes can fail** with **stop failure**.
- We suppose that the nodes have a “failure detector”, based on “timeouts”, to recognize when a node has failed.
- To solve most problems in distributed algorithms in the presence of failures, we assume that timers are “fine” adjusted.
    - Sooner or later, all the “right” nodes will see that the others correct nodes are correct.
    - Sooner or later, the timers are well adjusted.
    - We call these well-adjusted timers as “eventually perfect failures detectors”.

Basic algorithm:

- The nodes are numbered and ordered, from 0 to N-1.
- The nodes may be running or being “off” or fail during the algorithm.
- We can tolerate $\lfloor (N-1)/2 \rfloor$ failures.
- Therefore, it **only works** if there is at least $\lceil (N+1)/2 \rceil$ **correct nodes**, where $\lceil X \rceil$ is the ceiling of the real number x.
- Rounds are executed until there is a round where at least $\lceil (N+1)/2 \rceil$ nodes successfully participate.
- If more nodes than $\lfloor (N-1)/2 \rfloor$ fail, the algorithm does NOT work, because the coordinating nodes get blocked.

General behaviour:

- The nodes will execute rounds until they issue “decision(V)”.
- In each round they select as coordinator the next node.
- In each round we distinguish what the **coordinator** of the round does and the other nodes (**ordinary nodes**).
- All nodes maintain these variables:
    - Current round (r).
    - Current coordinator (NC).
    - Value that the node proposes to the variable (lastEstimate).
    - Most recent round that made the node change the proposal (lastR).

Start:

1. All nodes start r = 0, NC = 0, lastR = 0, lastEstimate = value proposed by each node.

In each round "r":

Phase 1:

1. All nodes send to the coordinator of the NC round: "estimate(r, lastEstimate, lastR)".
2. Ordinary nodes wait response a "propose" message.
    a. They wait for the response message a maximum time.
    b. If the time is exceeded, we say that its "failure detector" believes that the coordinator has failed.

Phase 2:

1. The coordinator waits to receive $\lceil (N+1)/2 \rceil$ "estimate" messages.
    a. Timeouts are not used.
    b. The coordinator knows that he will receive at least that number of messages, since we do not admit a greater number of possible failures.
    c. Until it does not receive this quantity of messages, the coordinator is waiting. And the algorithm does not progress.
2. The coordinator chooses one of the "lastEstimate" values that he receives, from among those with the highest "lastR" value, and this will be the proposal of the round.
    a. Assigns its "lastEstimate" value to this chosen value.
    b. Assigns lastR = r (its own round).
    c. The coordinator broadcasts "propose(r, lastEstimate)".

Phase 3:

1. Ordinary nodes wait to receive "propose(r,proposeR)" from the coordinator or their maximum waiting timeout expires.
    a. If they receive "propose" then:
        i. They answer with "ACK(r)" to NC.
        ii. They update lastEstimate = proposeR and lastR = r.
    b. If its timeout expires when waiting for proposal, they answer with "NACK(r)" to NC.

Phase 4:

1. The coordinator waits ACK or NACK responses from the ordinary nodes.
2. If the coordinator receives $\lceil (N+1)/2 \rceil$ ACK messages:
    a. It broadcasts "decide(lastEstimate)" and generates "decision(lastEstimate)".
    b. This node already knows that consensus has been reached. This is the final value.
3. If an ordinary node receives "decide(lastEstimate)", it generates "decision(lastEstimate)". This node now knows that consensus has been reached.
4. All the nodes that have generated "decision" continue to participate in the algorithm as PASSIVE nodes.
    a. When it receives a propose(r, proposeR) message, it answers ACK if proposeR == last Estimateand r >= lastR.

Round changes:

- **Coordinator**: after Phase 4, if the coordinator does not receive a sufficient number of ACK, then he increases the round number and will no longer be a coordinator.

- **Ordinary node**: after Phase 3, the ordinary node increases round and becomes the new coordinator if it corresponds.

Ending: the algorithm ends when all nodes are in PASSIVE mode.

Conclusion:

- To reach consensus in the presence of failures, we need to have well-adjusted timeouts.
- For the vast majority of realistic algorithms and realistic systems we need failure detectors that “sooner or later” are quite “good”.
- The treatment of failures is complex, and its complete treatment is out of the scope of this subject.