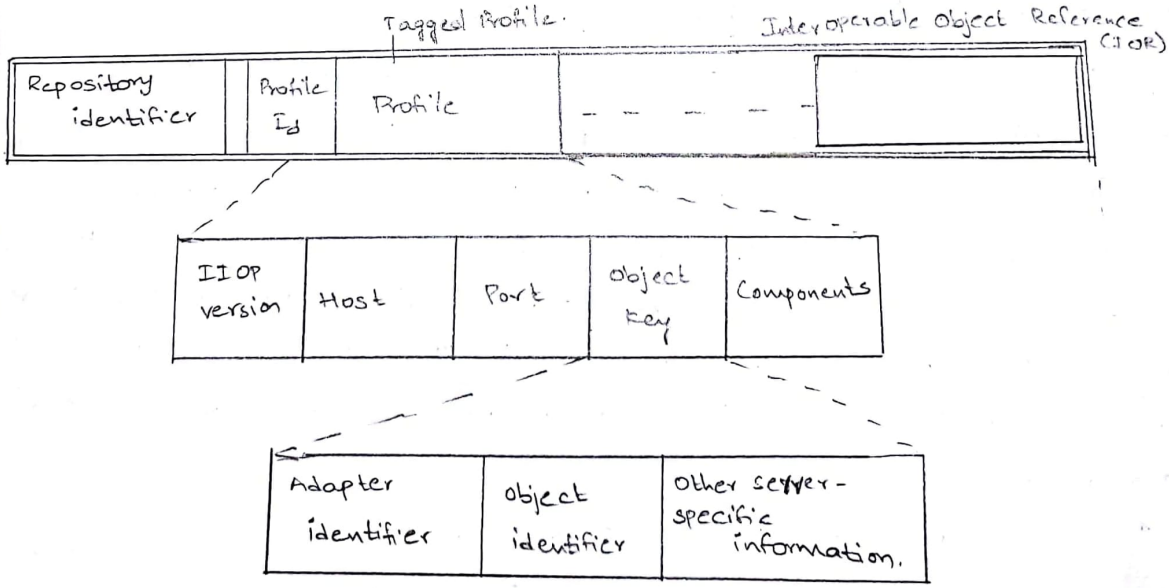
1. Explain naming in distributed object based system with an example.

When a client holds an object reference, it can invoke the methods implemented by the referenced object. It is important to distinguish the object reference that a client process uses to invoke a method, and the one implemented by the underlying RTS.

A process (client or server) can use only a language-specific implementation of an object reference. In most cases, this takes the form of a pointer to a local representation of the object. That reference cannot be passed from process A to process B, as it has meaning only within the address space of process A. Instead, process A will first have to marshal the pointer into a process-independent representation. The operation to do so is provided by its RTS. Once marshaled, the reference can be passed to process B, which can unmarshal it again. Note that the processes A and B may be executing programs written in different languages.



The organisation of an IOR with specific information for IIOP.

Each IOR starts with a repository identifier. This identifier is assigned to an interface so that it can be stored and looked up in an interface repository. It is used to retrieve information on an interface at runtime, and can assist in. Note that if this identifier is to be useful, both the client and server must have access to the same interface repository, or at least use the same identifier to identify interfaces.

The most important part of each IOR is formed by what are called tagged profiles. Each such profile contains the complete information to invoke an object. If the object server supports several protocols, information on each protocol, can be included in a separate tagged profile. CORBA used the 'Internet Inter-ORB Protocol (IIOP) for communication between nodes.

The IIOP profile is identified by a profile ID field in the tagged profile. Its body consists of five fields. The IIOP version field identifies the version of IIOP that is used in this profile.

The Host field is a string identifying exactly on which host the object is located.

The Port field contains the port number to which the object's server is listening for incoming requests.

The Object key field contains server-specific information for demultiplexing incoming requests to appropriate object. For example, an object identifier generated by CORBA object adapter will generally be part of such an object key. Also, this key will identify the specific adapter.

The components field that optionally contains more information needed for properly invoking the referenced object.

2. What is reliable client-server communication?

In many cases, fault tolerance in distributed systems concentrates on faulty processes. However, we also need to consider communication failures. Most of the failure models apply equally well to communication channels.

Point-to-point communication:

Reliable point-to-point communication is established by making use of a reliable transport protocol, such as TCP. TCP masks omission failures, which occur in the form of lost messages, by using acknowledgements and retransmissions. Such failures are completely hidden from a TCP client.

Crash failures of connections are not masked. A crash failure may occur when a TCP connection is abruptly broken so that no more messages can be transmitted through the channel. In most cases, the client is informed that the channel has crashed by raising an exception. The only way to mask such failures is to let the distributed system attempt to automatically set up a new connection, by simply resending a connection request. The underlying assumption is that other side is still, or again, responsive to such requests.

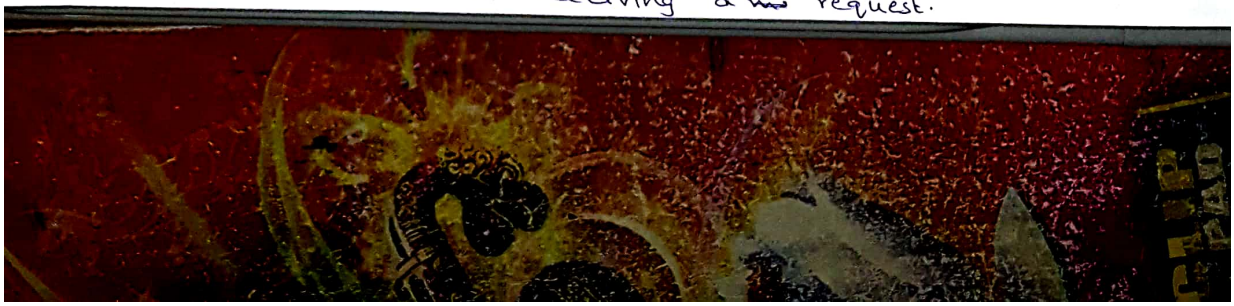
RPC semantics in the Presence of Failures:

The goal of Remote Procedure Calls (RPCs) is to hide communication by making remote procedure calls look just like local ones. With a few exceptions, so far we have come fairly close. Indeed, as long as both client and server are functioning perfectly, RPC does its job well. The problem comes about when errors occur. It is then that the differences between local and remote calls are not always easy to mask.

3. Explain briefly the RPC semantics in the presences of failures.

There are five different classes of failures that can occur in RPC systems.

1. The client is unable to locate the server.
2. The request message from the client to the server is lost
3. The server crashes after receiving a request.



crashes before it can send the reply. Finally, Again a request arrives, but this time the server crashes before

4. The reply message from the server to the client is lost.
5. The client crashes after sending a request.

Client cannot ~~re~~locate the server:

To start with, it can happen that the client cannot locate a suitable server. All servers might be down. Alternatively, suppose that the client is compiled using a particular version of the client stub, and the binary is not used for a considerable period of time.

One possible solution is to have the error raise an exception. In some languages, programmers can write special procedures that are invoked upon specific errors, such as division by zero. In C, signal handlers can be used for this purpose. In other words, we could define a new signal type SIGNO_SERVER, and allow it to be handled in the same way as other signals.

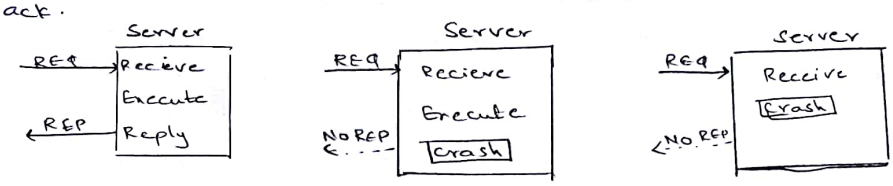
Lost Request Messages.

This is the easiest one to deal with: just have the operating system or client stub start a timer when sending a request. If the timer expires before a reply or acknowledgement comes back, the message is sent again. If the message was truly lost, the server will not be able to tell the difference between retransmission and the original, and everything will work fine. Unless, of course, so many request messages are lost that the client gives up and falsely concludes that the server is down, in which case we are back to "Cannot locate server". If the request was not lost, the only thing we need to do is let the server be able to detect it is dealing with a retransmission. Unfortunately, doing so is not so simple. ~~so~~

Server Crashes:-

The next failure on the list is server crash. The normal sequence of events at a server are as follows. A request arrives, is carried out, and a reply is sent. A request arrives and is carried out, just as before, but

crashes before it can send the reply. Finally, Again a request arrives, but this time the server crashes before it can even be carried out. And, of course, no reply is sent back.

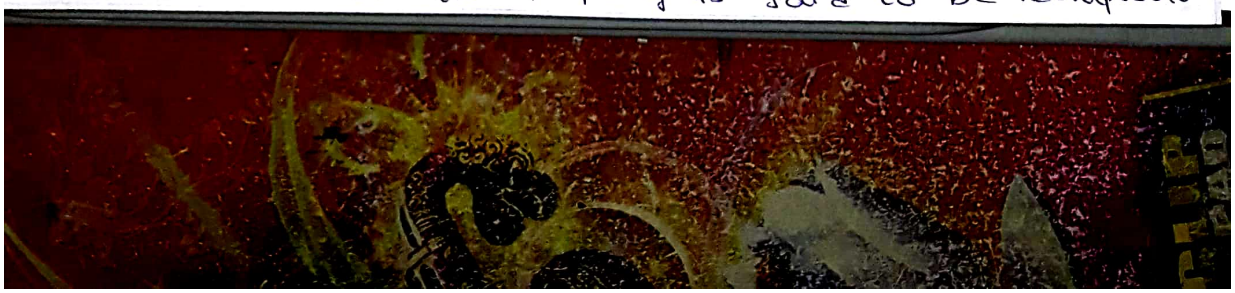


There are four strategies the client can follow. First, the client can decide to never reissue a request, at the risk that the text will not be printed. Second, it can decide to always reissue a request, but this may lead to its text being printed twice. Third, it can decide to reissue a request only if it did not yet receive an acknowledgement that its print request had been delivered to the server. In that case, the client is counting on the fact that the server crashed before the print request could be delivered. The fourth and last strategy is to reissue a request only if it has received an acknowledgement for the print request.

Lost Reply Messages:

Lost replies can also be difficult to deal with. The obvious solution is just to rely on a timer again that has been set by the client's operating system. If no reply is forthcoming within a reasonable period, just send request once more. The trouble with this solution is that the client is not really sure why there was no answer. Did the request or reply get lost, or is the server merely slow? It may make a difference.

In particular, some operations can safely be repeated as often as necessary with no damage being done. A request such as asking for the first 1024 bytes of a file has no side effects and can be executed as often as necessary without any harm being done. A request that has this property is said to be idempotent.



Client crashes:

The final item on the list of failures is client crash. What happens if a client sends a request to a server to do some work and crashes before the server replies? At this point a communication computation is active and no parent is waiting for the result. Such an unwanted computation is called an Orphan.

Before a client stub sends an RPC message, it makes a log entry telling what it is about to do. The log is kept on disk or some other medium that survives crashes. After a reboot, the log is checked and the orphan is explicitly killed off. This solution is called orphan extermination.

Other solutions are to create grand orphans, or reincarnation, gentle reincarnation and expiration.

4. In detail explain client-centric consistency models.

An important assumption is concurrent processes may be simultaneously updating the data store, and that it is necessary to provide consistency in the face of such concurrency.

Eventual Consistency:

If no updates take place for a long time, all replicas will gradually become consistent. This form of consistency is called eventual consistency.

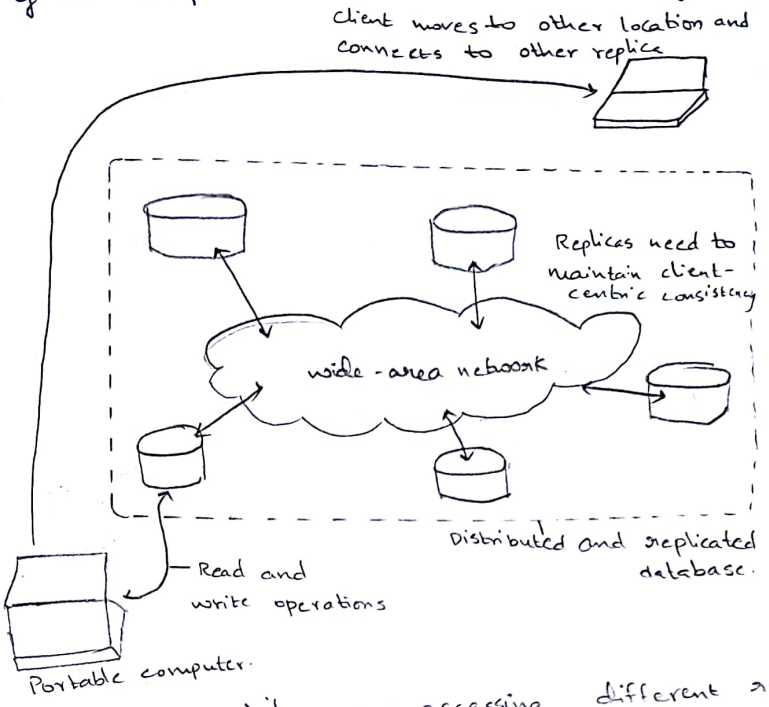
Data stores that are eventually consistent thus have the property that in the absence of updates, all replicas converge toward identical copies of each other.

Eventual consistency essentially requires only that updates are guaranteed to propagate all replicas.

This data work fine long as clients always access the same replica.

Client-centric consistency provides guarantees for a single client concerning the consistency of accesses to a data store by that client.

The mobile user accesses that database by connecting to one of the replicas in a transparent way.



The principle of a mobile user accessing different replicas of a distributed database.

Monotonic Reads:

The first client-centric consistency model is that of monotonic reads. A data store is said to prove monotonic read consistency if the following condition holds:

If a process reads the value of a data item x, any successive read operations on x by that process will always return that same value or a more recent value.

Monotonic-read consistency guarentees that if a process has seen a value of x at time t, it will never see an older version of x at a later time.

The read operations performed by single process P at two diff. local copies of the same data store. (a) A monotonic-read consistent data store. b) A data store that does not provide monotonic reads.

Monotonic Writes:

It is important that write operations are propagated in the correct order to all copies of the data store. This property is expressed in monotonic-write consistency. In a monotonic-write consistent store, the following condition holds:

A write operation by a process on a data item x is completed before any successive write operation on x by the same process.

(a) (b)

The write operations performed by a single process P at two different local copies of the same data store. (a) A Monotonic-write consistent data store. b) A data store that does not provide Monotonic-write consistency.

Read your writes.

A client-centric consistency model that is closely related to monotonic reads is as follows: A data store is said to provide read-your-writes consistency, if the following condition holds.

The effect of a write operation by a process on data item x will always be seen by a successive read operation on x by the same process.

a) A data store that provides read-your-writes consistency.
b) A datastore that does not.

Writes Follows Reads.

The last client-centric consistency model is one in which updates are propagated as the result of previous read operations. A data store is said to provide writes-follow-reads consistency, if the following holds.

A write operation by a process on a data item x following a previous read operation on x by the same process is

guaranteed to take place on the same or a more recent value of x that was read.

L1: WS(x1)	R(x1);
L2:	WS(x1, x2) ; WC(x2)

(a)

provides

L1: WS(x1)	R(x);
L2:	WS(x2) ; W(x2)

(b)

does not provide.

5. Explain ~~election~~ algorithms for synchronisation in DS.

Many distributed algorithms require one process to act as Co-ordinator, initiator, or otherwise perform some special role. In general, it does not matter which process takes on this special responsibility, but one of them has to do it.

If all ~~res~~ processes are exactly the same, with no distinguishing characteristics, there is no way to select of them to be special. Consequently, we will assume that each processes has a unique number. The algorithms differ in the way they do the location.

Traditional Election Algorithms:

Bully Algorithm:

When any process notices that the coordinator is no longer responding to requests, it initiates an election. A process P, holds an election as follows:

1. P sends an ELECTION message to all processes with higher numbers.
2. If no one responds, P wins the election and become co-ordinator.
3. If one of the higher-ups answers, it takes over. P's job is done.

A Ring Algorithm:

Another election algorithm is based on the use of a ring. Unlike some ring algorithms, this one does not use a token. When any process notices that the co-ordinator is not functioning, it builds an ELECTION message containing its own process number and sends the message to its successor. If the successor is down, the sender (keeps) skips over the successor and goes to the next member along the ring, or the one after that, until a running process (number)

is located. At each step along the way, the sender adds its own process number to the list in the message effectively making itself a candidate to be elected as co-ordinator.

Elections in Wireless Environments:

Consider a wireless ad hoc network. To elect a leader, any node in the network, called the source, can initiate an election by sending an ELECTION message to its immediate neighbours (i.e., the nodes in its range). When a node receives an ELECTION for the first time, it designates the sender as its parent, and subsequently sends out an ELECTION message to all its immediate neighbours, except for the parent. When a node receives an ELECTION message from a node other than its parent, it merely acknowledges the receipt.

Elections in Large-scale systems.

There are situations when several nodes should actually be selected, such as in case of superpeers in peer-to-peer networks, which we concentrate specifically on the problem of selecting superpeers.

The following requirements that need to be met for superpeer selection:

1. Normal nodes should have low-latency access to superpeers.
2. Superpeers should be evenly distributed across the overlay network.
3. There should be a predefined portion of superpeers relative to the total number of nodes in the overlay network.
4. Each superpeer should not need to serve more than a fixed number of nodes.

These requirements are relatively easy to meet in most peer-to-peer systems, given the fact that the overlay network is either structured or randomly unstructured.

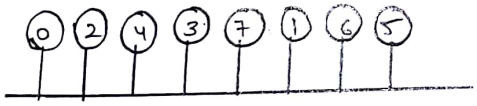
6. How do you acheive mutual exclusion? Explain Token ring algorithm.

Distributed mutual exclusion algorithms can be classified into two different categories. In token-based solutions mutual exclusion is acheived by passing a special message between between the processes, known as a token. There is only one token available and who ever has that token is allowed to access the shared resource.

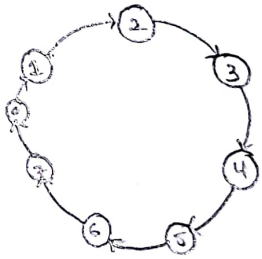
Token-based solutions have a few important properties. First, depending on the how the processes are organised, they can fairly easily ensure that every process will get a chance at accessing the resource. They avoid starvation. Second, dead-locks by which several processes are waiting for each other to proceed, can easily be avoided.

Token-Ring Algorithm.

A completely different approach to deterministically acheiving mutual exclusion in a DS is Token ring algorithm. In software, a logical ring is constructed in which each process is assigned a position in the ring. The ring positions may be allocated in numerical order of network addresses or some other means. It does not matter what the ordering is. All that matters is that each process knows who is next in line after itself.



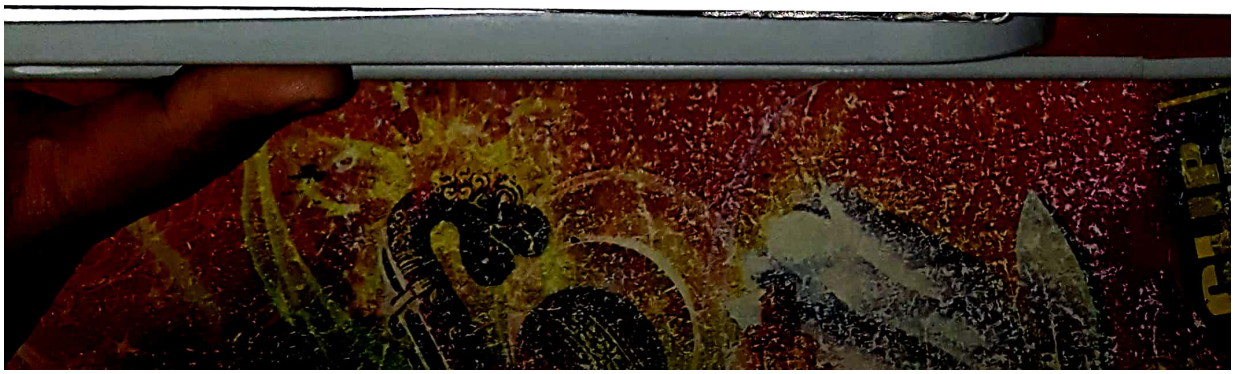
(a)



(b)

a) An unordered group of processes on a network

b) A logical ring constructed in software.



When the ring is initialized, process 0 is given a token. The token circulates around the ring. It is passed from process k to process k+1 (modulo the ring size) in point-to-point messages. When a process acquires the token from its neighbour, it checks to see if it needs to access the shared resource. If so, the process goes ahead, does all the work it needs to, and releases the resources. After it has finished, it passes the token along the ring. It is not permitted to immediately enter the resource again using the same token.

If a process is handed the token by its neighbor and is not interested in the resource, it just passes the token along. As a consequence, when no processes need the resource, the token just circulates at high speed around the ring.

The correctness of this algorithm is easy to see. Only one process has the token at any instant, so only one process can actually get to the resource. Since the token circulates among the processes in a well-defined order, starvation cannot occur. once a process decides it wants to have access to the resource, at worst it will have to wait for every other process to use the resource.

This algorithm has problems too. If the token is ever lost, it must be regenerated. In fact, detecting that it is lost is difficult, since the amount of time between successive appearences of the token on the network is unbounded. The fact that the token has not been spotted for an hour does n mean that it has been lost; somebody may still be using it.