# MODULE-2: Communication

VIT | Vidyalankar Institute of Technology
Accredited A+ by NAAC

**Prepared by Prof. Amit K. Nerurkar**

# Certificate

This is to certify that the e-book titled "COMMUNCATION" comprises all elementary learning tools for a better understating of the relevant concepts. This e-book is comprehensively compiled as per the predefined eight parameters and guidelines.

Signature Date: 20-03-2020

Prof. Amit K. Nerurkar

Assistant Professor

Department of Computer Engineering

⚠ *DISCLAIMER: The information contained in this e-book is compiled and distributed for educational purposes only. This e-book has been designed to help learners understand relevant concepts with a more dynamic interface. The compiler of this e-book and Vidyalankar Institute of Technology give full and due credit to the authors of the contents, developers and all websites from wherever information has been sourced. We acknowledge our gratitude towards the websites YouTube, Wikipedia, and Google search engine. No commercial benefits are being drawn from this project.*

**Module 2** **Communication**

## LAYERED PROTOCOLS, REMOTE PROCEDURE CALL, REMOTE OBJECT INVOCATION, MESSAGE ORIENTED COMMUNICATION, STREAM ORIENTED COMMUNICATION

### LAYERED PROTOCOLS

**Q.1 Explain Layered Protocols.**

**(A)** Due to the absence of shared memory, all communication in distributed systems is based on exchanging (low level) messages. When process *A* wants to communicate with process *B,* it first builds a message in its own address space. Then it executes a system call that causes the operating system to send the message over the network to *B.* Although this basic idea sounds simple enough, in order to prevent chaos, *A* and *B* have to agree on the meaning of the bits being sent. If *A* sends a brilliant new novel written in French and encoded in IBM's EBCDIC character code, and *B* expects the inventory of a supermarket written in English and encoded in ASCII, communication will be less than optimal.

To make it easier to deal with the numerous levels and issues involved in communication, the International Standards Organization (ISO) developed a reference model that clearly identifies the various levels involved, gives them standard names, and points out which level should do which job. This model is called the **Open Systems Interconnection Reference Model** usually abbreviated as **ISO OSI** or sometimes just the **OSI model.** It should be emphasized that the protocols that were developed as part of the OSI model were never widely used.

The OSI model is designed to allow open systems to communicate. An open system is one that is prepared to communicate with any other open system by using standard rules that govern the format, contents, and meaning of the messages sent and received. These rules are formalized in what are called **protocols.** To allow a group of computers to communicate over a network, they must all agree on the protocols to be used. A distinction is made between two general types of protocols. With **connection-oriented** protocols, before exchanging data the sender and receiver first explicitly establish a connection, and possibly negotiate the protocol they will use. When they are done, they must release (terminate) the connection. The telephone is a connection-oriented communication system. With **connectionless** protocols, no setup in advance is needed. The sender just transmits the first message when it is ready. Dropping a letter in a mailbox is an example of connectionless communication. With computers, both connection-oriented and connectionless communication are common.

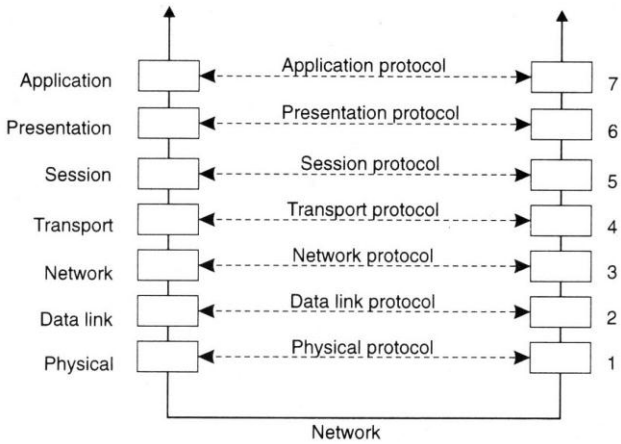


**Fig.1 :** Layers, interfaces and protocols in the OSI model

In the OSI model, communication is divided up into seven levels or layers, as shown in Fig.1 above. Each layer deals with one specific aspect of the communication. In this way, the problem can be divided up into manageable pieces, each of which can be solved independent of the others. Each layer provides an interface to the one above it. The interface consists of a set of operations that together define the service the layer is prepared to offer its users.

When process *A* on machine 1 wants to communicate with process *B* on machine 2, it builds a message and passes the message to the application layer on its machine. This layer might be a library procedure, for example, but it could also be implemented in some other way (e.g., inside the operating system, on an external network processor, etc.). The application layer software then adds a **header** to the front of the message and passes the resulting message across the layer 6/7 interface to the presentation layer. The presentation layer in turn adds its own header and passes the result down to the session layer, and so on. Some layers add not only a header to the front, but also a trailer to the end. When it hits the bottom, the physical layer actually transmits the message, which looks as shown in Fig.2.

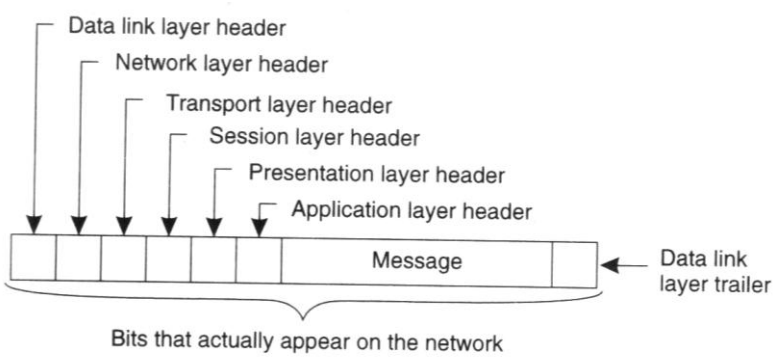


**Fig.2 :** A typical message as it appears on the network

### Lower-Level Protocols

We start with discussing the three lowest layers of the OSI protocol suite. Together, these layers implement the basic functions that encompass a computer network.

### Physical Layer

The physical layer is concerned with transmitting the 0s and 1s. How many volts to use for 0 and 1, how many bits per second can be sent, and whether transmission can take place in both directions simultaneously are key issues in the physical layer. In addition, the size and shape of the network connector (plug), as well as the number of pins and meaning of each are of concern here.

The physical layer protocol deals with standardizing the electrical, mechanical, and signaling interfaces so that when one machine sends a 0 bit it is actually received as a 0 bit and not a 1 bit. Many physical layer standards have been developed (for different media), for example, the RS-232-C standard for serial communication lines.

### Data Link Layer

The physical layer just sends bits. As long as no errors occur, all is well. However, real communication networks are subject to errors, so some mechanism is needed to detect and correct them. This mechanism is the main task of the data link layer. What it does is to group the bits into units, sometimes called **frames,** and see that each frame is correctly received.

The data link layer does its work by putting a special bit pattern on the start and end of each frame to mark them, as well as computing a **checksum** by adding ᵘP all the bytes in the frame in a certain way. The data link layer appends the checksum to the frame. When the frame arrives," the receiver recomputes the checksum from the data and compares the result to the checksum following the frame. If they agree, the frame is considered correct and is accepted. It they Disagree, the receiver asks the sender to retransmit it. Frames are assigned sequence numbers (in the header), so everyone can tell which is which.

In Fig.3, we see an example of *A* trying to send two messages, 0 and 1, to *B.* At time 0, data message 0 is sent, but when it arrives, at time 1, noise on the transmission line has damaged so that the checksum is wrong. *B* notices this, and at time 2 asks for a retransmission using a control message. Unfortunately, at the same time, *A* is sending data message 1. When *A* gets the request for retransmission, it resends 0. However, when *B* gets message 1, instead of the requested message 0, it sends control message 1 to *A* complaining that it wants 0, not 1. When *A* sees this, it shrugs its shoulders and sends message 0 for the third time.

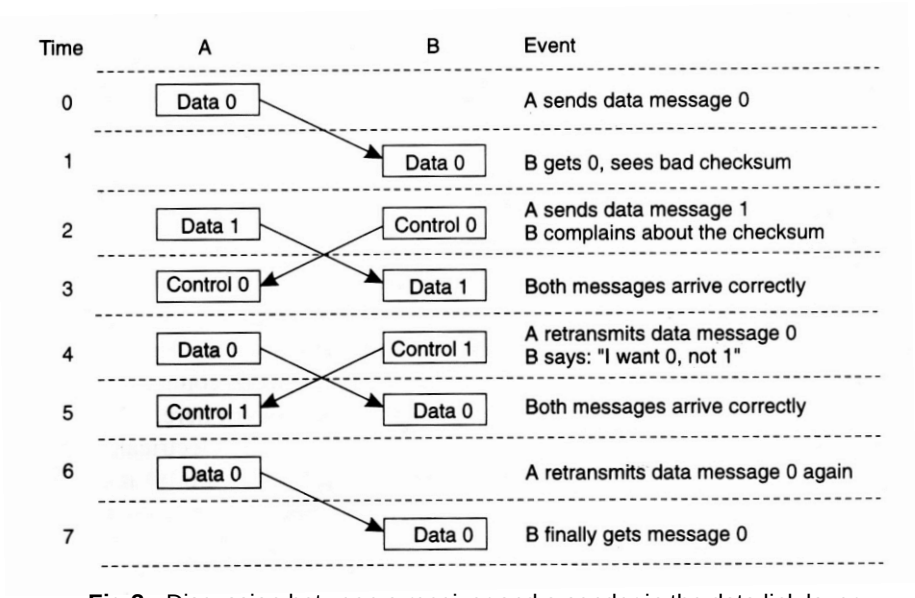


**Fig.3 :** Discussion between a receiver and a sender in the data link layer

**Network Layer**

On a LAN, there is usually no need for the sender to locate the receiver. It just puts the message out on the network and the receiver takes it off. A wide-area network, .however, consists of a large number of machines, each with some number of lines to other machines, rather like a large-scale map showing major cities and roads connecting them. For a message to get from the sender to the receiver it may have to make a number of hops, at each one choosing an outgoing line to use. The question of how to choose the best path is called **routing,** and is essentially the primary task of the network layer.

At present, perhaps the most widely used network protocol is the connectionless **IP** (Internet Protocol), which is part of the Internet protocol suite. An IP **packet** (the technical term for a message in the network layer) can be sent without any setup. Each IP packet is routed to its destination independent of all others. No internal path is selected and remembered.

A connection-oriented protocol that is now gaining popularity, is the **virtual channel** in ATM networks. A virtual channel in ATM is a unidirectional connection from a source to a destination, possibly crossing several intermediate ATM switches. Instead of setting up each virtual channel separately between two hosts, a collection of virtual channels can be grouped into what is called a **virtual path.** A virtual path is comparable to a predefined route between two hosts, along which all its virtual channels are laid down. Rerouting a path implies that all the associated channels are automatically rerouted as well.

**Transport Protocols**

The transport layer forms the last part of what could be called a basic network protocol stack, in the sense that it implements all those services that are not provided at the interface of the network layer, but which are reasonably needed to build network applications.

**The Function of the Transport Layer**

Packets can be lost on the way from the sender to the receiver. Although some applications can handle their own error recovery, others prefer a reliable connection. The job of the transport layer is to provide this service. The idea is that the application layer should be able to deliver a message to the transport layer with the expectation that it will be delivered without loss.

Upon receiving a message from the application layer, the transport layer breaks it into pieces small enough for transmission, assigns each one a sequence number, and then sends them all. The discussion in the transport layer header concerns which packets have been sent, which have been received, how many more the receiver has room to accept, which should be retransmitted, and similar topics.

**Higher-Level Protocols**

Above the transport layer, OSI distinguished three additional layers. In practice, only the application layer is ever used. In fact, in the Internet protocol suite, everything above the transport layer is grouped together. In the face of middleware systems.

**Session and Presentation Protocols**

The session layer is essentially an enhanced version of the transport layer. It provides dialog control, to keep track of which party is currently talking, and it provides synchronization facilities. The latter are useful to allow users to insert checkpoints into long transfers, so that in the event of a crash, it is necessary to go back only to the last checkpoint, rather than all the way back to the beginning. In practice, few applications are interested in the session layer and it is rarely supported. It is not even present in the Internet protocol suite.

Unlike the lower layers, which are concerned with getting the bits from the sender to the receiver reliably and efficiently, the presentation layer is concerned with the meaning of the bits. Most messages do not consist of random bit strings, but more structured information such as people's names, addresses, amounts of money, and so on. In the presentation layer it is possible to define records containing fields like these and then have the sender notify the receiver that a message contains a particular record in a certain format. This makes it easier for machines with different internal representations to communicate.

**Application Protocols**

The OSI application layer was originally intended to contain a collection of standard network applications such as those for electronic mail, file transfer, and terminal emulation. By now, it has become the container for all applications and protocols that in one way or the other do not fit into

one of the underlying layers. From the perspective of the OSI reference model, virtually all distributed systems are just applications.

What is missing in this model is a clear distinction between applications, application-specific protocols, and general-purpose protocols. For example, the Internet File Transfer Protocol (FTP) defines a protocol for transferring files between a client and server machine. The protocol should not be confused with the *ftp* program, which is an end-user application for transferring files and which also happens to implement the Internet FTP.

Another example of a typical application-specific protocol is the HyperText Transfer Protocol (HTTP), which is designed to remotely manage and handle the transfer of Web pages. The protocol is implemented by applications such as Web browsers and Web servers. However, HTTP is now also used by systems that are not intrinsically tied to the Web. For example, Java's RMI uses HTTP to request the invocation of remote objects that are protected by a firewall.

**VIDEO**

**RPC IN DETAIL**

**Q.2 How to perform Remote Procedure Call (RPC)?**

**(A)** The problem with the basic client-server model is that conceptually interprocess communication is handled as I/O. To provide a better obstruction remote procedure call (RPC) is widely used.

In a nutshell, what Birrell and Nelson suggested was allowing programs to call procedures located on other machines when a process on machine. A calls procedure on machine B, the calling process on A is suspended, and the execution of the called procsedure takes place on B. Information can be transported from the caller to the callee in the parameter and can came back in the procedure result. No message passing or I/O at all i.e. visible to the programmer. This method is known as remote procedure call (RPC).

**A remote procedure call occurs in the following steps**

1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and sends to the kernel.
3. The Kernel sends the message to the remote Kernel.
4. The remote Kernel gives the message to the server stub.
5. The server stub unpacks the parameters and call the server.
6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and traps to the Kernel.
8. The remote Kernel sends the message to the client's kernel.
9. Client's kernel gives the message to the client stub.
10. The stub unpacks the result and returns to the client.

The net effect of all these steps is to convert the local call by the client procedure to the client stub to a local call to the server procedure without either client or server being aware of the intermediate steps.

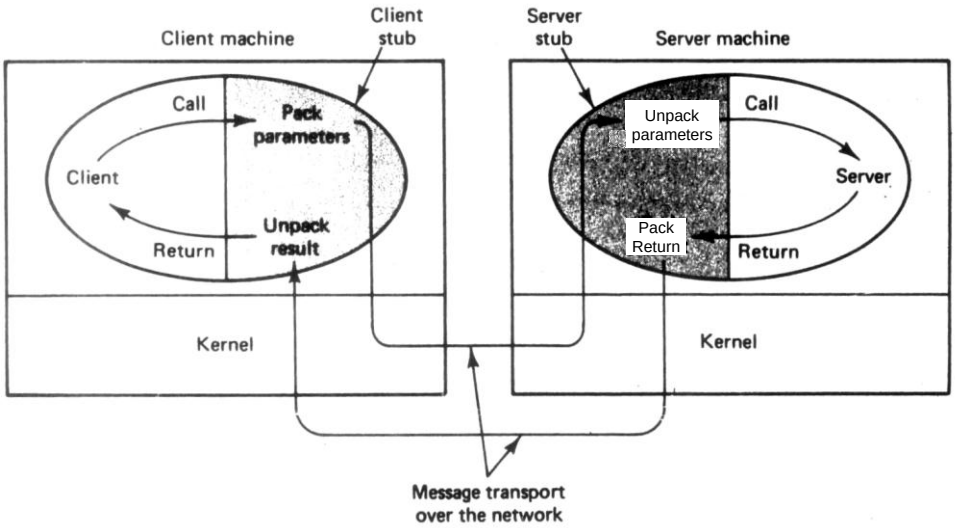


**Fig.:** Calls and messages in an RPC. Each ellipse represents a single process, with the shaded portion being the stub

**The RPC Model**

The RPC model is similar to the well-known and well-understood procedure call model used for the transfer of control and data within a program in the following manner:

i) For making a procedure call, the caller places arguments to the procedures in some well-specified location.
ii) Control is then transferred to the sequence of instructions that constitutes the body of the procedure.
iii) The procedure body is executed in a newly created execution environment that includes copies of the arguments given in the calling instruction.
iv) After the procedure's execution is over, control returns to the calling point, possibly returning a result.

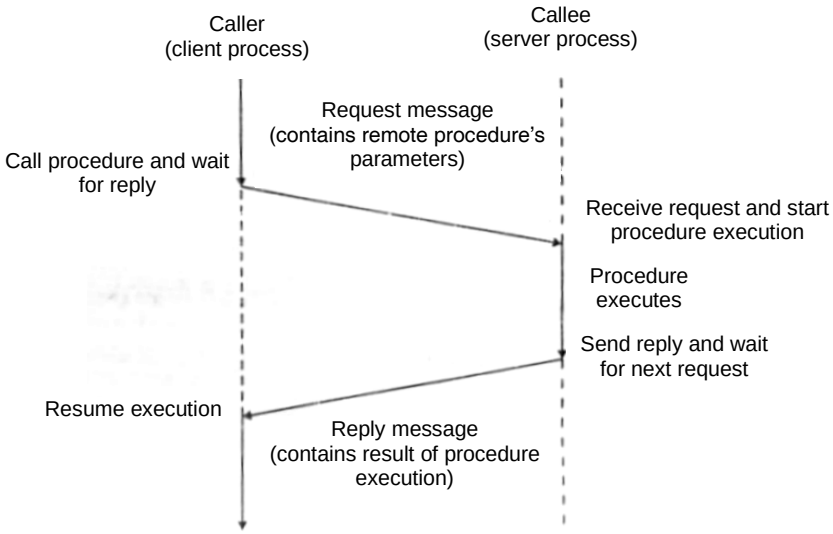


**Fig. :** A typical model of Remote Procedure Call

As shown in figure, when a remote procedure call is made, the caller and the callee processes interact in the following manner :

i) The caller (commonly known as *client process*) sends a call (request) message to the callee (commonly known as server process) and waits (blocks) for a reply message. The request message contains the remote procedure's parameters, among other things.
ii) The server process executes the procedure and then returns the result of procedure execution in a reply message to the client process.
iii) Once the reply message is received, the result of procedure execution is extracted, and the caller's execution is resumed.

**Video**

## TRANSPARENCY OF RPC

**Q.3 Discuss Transparency of RPC.**

**(A)** A major issue in the design of an RPC facility is its transparency property. A transparent RPC mechanism is one in which local procedures and remote procedures are (effectively) indistinguishable to programmers. This requires the following two types of transparencies.

i) Syntactic transparency means that a remote procedure call should have exactly the same syntax as a local procedure call.

ii) Semantic transparency means that the semantics of a remote procedure call are identical to those of a local procedure call.

Unfortunately, achieving exactly the same semantics for remote procedure calls as for local procedure calls is close to impossible. This is mainly because of the following differences between remote procedure calls and local procedure calls :

i) Unlike local procedure calls, with remote procedure calls, the called procedure is executed in an address space that is disjoint from the calling program's address space. Due to this reason, the called (remote) procedure cannot have access to any variables or data values in the calling program's environment. Thus in the absence of shared memory, it is meaningless to pass addresses in arguments, making call-by-reference pointers highly unattractive. Similarly, it is meaningless to pass argument values containing pointer structures (e.g., linked lists), since pointers are normally represented by memory addresses.

ii) Remote procedure calls are more vulnerable to failures than local procedures calls, since they involve two different processes and possibly a network and two different computers. Therefore programs that make use of remote procedures call must have the capability of handling even those errors that cannot occur in local procedure calls.

iii) Remote procedure calls consume much more time (100-1000 times more) than local procedure calls. This is mainly due to the involvement of a communication network in RPCs. Therefore applications using RPCs must also have the capability to handle the long delays that may possibly occur due to network congestion.

### Implementing RPC Mechanism

To achieve the goal of semantic transparency, the implementation of an RPC mechanism is based on the concept of *stubs* which provide a perfectly normal (local) procedure call abstraction by concealing from programs the interface to the underlying RPC system. Implementation of an RPC mechanism usually involves the following five elements of program

1. The client
2. The client stub
3. The RPCRuntime
4. The server stub
5. The server

The interaction between them is shown in figure.

1. **Client :** The client is a user process that initiates a remote procedure call. To make a remote procedure call, the client makes a perfectly normal local call that invokes a corresponding procedure in the client stub.
2. **Client Stub :** The client stub is responsible for carrying out the following two tasks:
   - On receipt of a call request from the client, it packs a specification of the target procedure and the arguments into a message and then asks the local RPCRuntime to send it to the server stub.
   - On receipt of the result of procedure execution, it unpacks the result and passes it to the client.
3. **RPCRuntime :** The RPCRuntime handles transmission of messages across the network between client and server machines. It is responsible for retransmissions, acknowledgments, packet routing, and encryption. The RPCRuntime on the client machine receives the call request message from the client stub and sends it to the server machine. It also receives the message containing the result of procedure execution from the server machine and passes it to the client stub.

   On the other hand, the RPCRuntime on the server machine receives the message containing the result of procedure execution from the server stub and sends it to the client machine. It also receives the call request message from the client machine and passes it to the server stub.

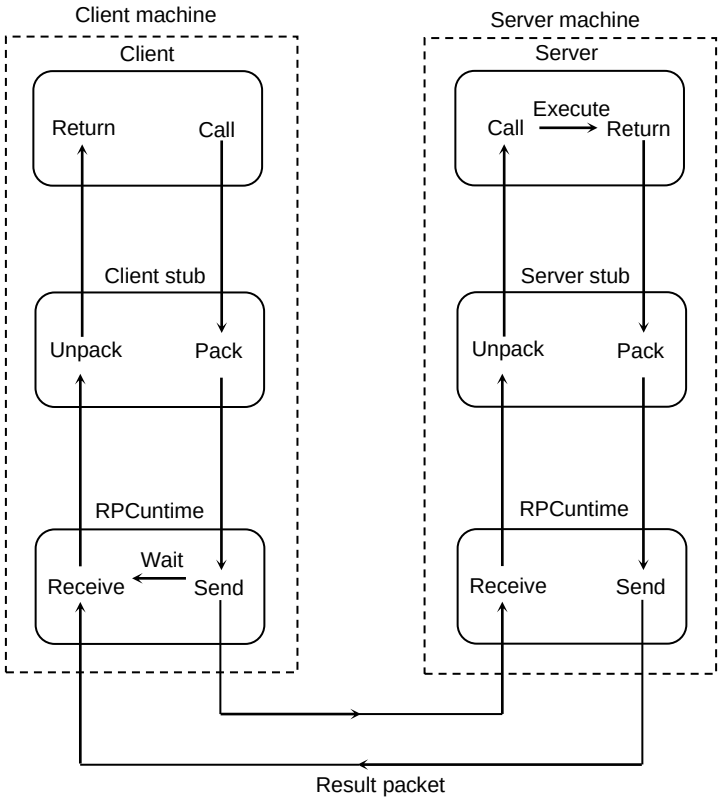


**Fig. :** Implementation of RPC mechanism

4. **Server Stub :** The job of the server stub is very similar to that of the client stub. It performs the following two tasks:
   - On receipt of the call request message from the local RPCRuntime, the server stub unpacks it and makes a perfectly normal call to invoke the appropriate procedure in the server.
   - On receipt of the result of procedure execution from the server, the server stub packs the result into a message and then asks the local RPCRuntime to send it to the client stub.
5. **Server :** On receiving a call request from the server stub, the server executes the appropriate procedure and returns the result of procedure execution to the server stub.

**Advantages**

The RPC has become a widely accepted IPC mechanism in distributed systems. The popularity of RPC as the primary communication mechanism for distributed applications is due to its following features:

i) Simple call syntax.
ii) Familiar semantics (because of its similarity to local procedure calls).
iii) Its specification of a well-defined interface. This property is used to support compile-time type checking and automated interface generation.
iv) Its ease of use. The clean and simple semantics of a procedure call makes it easier to build distributed computations and to get them right.
v) Its efficiency. Procedure calls are simple enough for communication to be quite rapid.
vi) It can be used as an IPC mechanism to communicate between processes on different machines as well as between different processes on the same machine.

**Limitations of RPC**

i) The correct server has to be located.
ii) Pointers and complex data structure are hard to pass.
iii) Global variable are difficult to use.
iv) The exact semantics of RPC are tricky because clients and servers can fail independently of one another.
v) Implementing RPC efficiently is not straightforward and requires careful thought.
vi) RPC is limited to those situations where a single client wants to talk a single server when a collection of processor, for example, replicated file servers, need to communicate with each other as a group, group communication is needed.

**LPC VS RPC**

**RPC SEMANTICS IN THE PRESENCE OF FAILURES**

**Q.4 Explain RPC Semantics in the Presence of Failures.**

**(A)** There are five different classes of failures that can occur in RPC systems, as follows:

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

**Client Cannot Locate the Server**

To start with, it can happen that the client cannot locate a suitable server. The server might be down, for example. 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. In the meantime, the server evolves and a new version of the interface is installed and new stubs are generated and put into use. When the client is finally run, the binder will be unable to match it up with a server and will report failure.

**Lost Request Messages**

The second item on the list is dealing with lost request messages. This is the easiest one to deal with: just have the kernel start a timer when sending the request. If the timer expires before a reply or acknowledgment comes back, the kernel sends the message again. If the message was truly lost, the server will not be able to tell the difference between the retransmission and the original, and everything will work fine. Unless, of course, so many request messages are lost that the kernel gives up and falsely concludes that the server is down, in which case we are back to "Cannot locate server."

**Lost Reply Messages**

Lost replies are considerably more difficult to deal with. The obvious solution is just to rely on the timer again. If no reply is forthcoming within a reasonable period, just send the request once more. The trouble with this solution is that the client's kernel 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 properly is said to be **idempotent**.

Now consider a request to a banking server asking to transfer a million dollars from one account to another. If the request arrives and is carried out, but the reply is lost, the client will not know this and will retransmit the message. The bank server will interpret this request as a new one, and will carry it out too. Two million dollars will be transferred.

**Server Crashes**

The next failure on the list is a server crash. It too relates to idempotency, but unfortunately it cannot be solved using sequence numbers. The normal sequence of events at a server is shown in figure (a). A request arrives, is carried out, and a reply is sent. Now consider figure (b). A request arrives and is carried out, just as before, but the server crashes before it can send the reply. Finally, look at figure (c). Again a request arrives, but this time the server crashes before it can even be carried out.

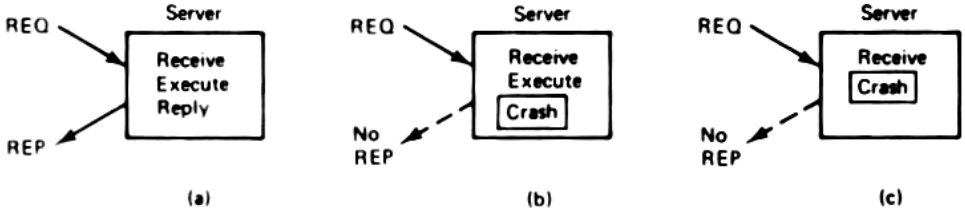


**Fig. :** (a) Normal case. (b) Crash after execution. (c) Crash before execution.

**Client Crashes**

The final item on the list of failures is the 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 computation is active and no parent is waiting for the result. Such an unwanted computation is called an **orphan**.

Orphans can cause a variety of problems. As a bare minimum, they waste CPU cycles. They can also lock files or otherwise tie up valuable resources. Finally, if the client reboots and does the RPC again, but the reply from the orphan comes back immediately afterward, confusion can result.

Nelson (1981) proposed four solutions. In **solution 1**, 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 **extermination**.

The disadvantage of this scheme is the horrendous expense of writing a disk record for every RPC. Furthermore, it may not even work, since orphans themselves may do RPCs, thus creating **grandorphans**.

In **solution 2**, called **reincarnation**, all these problems can be solved without the need to write disk records. The way it works is to divide time up into sequentially numbered epochs. When a client reboots, it broadcasts a message to all machines declaring the start of a new epoch. When such a broadcast comes in, all remote computations are killed. Of course, if the network is partitioned, some orphans may survive. However, when they report back, their replies will contain an obsolete epoch number, making them easy to detect.

**Solution 3** is a variant on this idea. It is called **gentle reincarnation**. When an epoch broadcast comes in, each machine checks to see if it is has any remote computations, and if so, tries to locate their owner. Only if the owner cannot be found is the computation killed.

Finally, we have **solution 4**, **expiration**, in which each **RPC** is given a standard amount of time, T, to do the job. If it cannot finish, it must explicitly ask for another quantum, which is a nuisance.

## REMOTE OBJECT INVOCATION

**Q.5 Explain Remote Object Invocation.**

**(A)** Object-based technology has proven its value for developing nondistributed applications. One of the most important aspects of an object is that it hides its internals from the outside world by means of a well-defined interface. This approach allows objects to be easily replaced or adapted, as long as the interface remains the same.

As RPC mechanisms gradually became the de facto standard for handling communication in distributed systems, people started to realize that the principle of RPCs could be equally well applied to objects.

### Distributed Objects

The key feature of an object is that it encapsulates data, called the **state,** and the operations on those data, called the **methods.** Methods are made available through an **interface.** It is important to understand that there is no legal way a process can access or manipulate the state of an object other than by invoking methods made available to it through an object's interface. An object may implement multiple interfaces. Likewise, given an interface definition, there may be several objects that offer an implementation for it.

This separation between interfaces and the objects implementing these interfaces is crucial for distributed systems. A strict separation allows us to place an interface at one machine, while the object itself resides on another machine. This organization, which is shown in Fig. below, is commonly referred to as a **distributed object.**

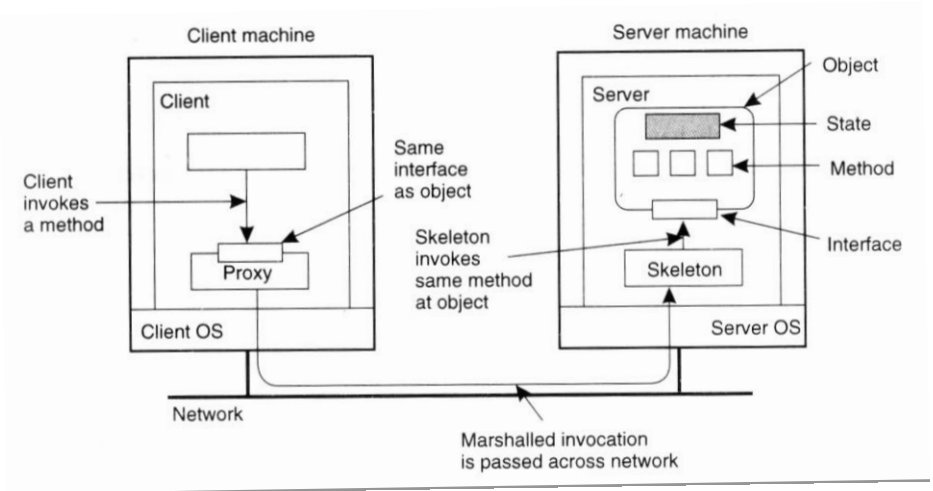


**Fig. :** Common organization of a remote object with client side proxy

When a client **binds** to a distributed object, an implementation of the object's interface, called a **proxy,** is loaded into the client's address space. A proxy is analogous to a client

stub in RPC systems. The only thing it does is marshal method invocations into messages and unmarshal reply messages to return the result of the method invocation to the client. The actual object resides at a server machine, where it offers the same interface as it does on the client machine. Incoming invocation requests are first passed to a server stub, often referred to as a **skeleton,** which unmarshals them to proper method invocations at the object's interface at the server. The server stub is also responsible for marshaling replies and forwarding reply messages to the client-side proxy.

A characteristic, but somewhat counterintuitive feature of most distributed objects is that their state is *not* distributed: it resides at a single machine. Only the interfaces implemented by the object are made available on other machines. Such objects are also referred to as **remote objects.** As we shall see in later chapters, a general distributed object, the state itself may be physically distributed across multiple machines, but this distribution is also hidden from clients behind the object's interfaces.

**Compile-time versus Runtime Objects**

Using compile-time objects in distributed systems often makes it much easier to build distributed applications. For example, in Java, an object can be fully defined by means of its class and the interfaces that the class implements. Compiling the class definition results in code that allows it to instantiate Java objects. The interfaces can be compiled into client-side and server-side stubs, allowing the Java objects to be invoked from a remote machine. A Java developer can mostly stay unaware of the distribution of objects: he sees only Java programming code. The obvious drawback of compile-time objects is the dependency on a particular programming language. Therefore, an alternative way of constructing distributed objects is to do this explicitly during runtime. This approach is followed in many object-based distributed systems, as it is independent of the programming language in which distributed applications are written. In particular, an application may be constructed from objects written in multiple languages.

**Persistent and Transient Objects**

A persistent object is one that continues to exist even if it is currently not contained in the address space of a server process. In other words, a persistent object is not dependent on its current server. In practice, this means that the server that is currently managing the persistent object, can store the object's state on secondary storage and then exit. Later, a newly started server can read the object's state from storage into its own address space, and handle invocation requests. In contrast, a transient object is an object that exists only as long as the server that manages the object.

**Static versus Dynamic Remote Method Invocations**

After a client is bound to an object, it can invoke the object's methods through the proxy. Such a **remote method invocation** or simply **RMI,** is very similar to an RPC when it comes to issues such as marshaling and parameter passing An essential difference between an RMI and an RPC, is that RMIs generally support systemwide object references as explained above. Also, it is not necessary to have only general-purpose

client-side and server-side stubs available. Instead, we can more easily accommodate object-specific stubs as we also explained.

The usual way to provide RMI support is to specify the object's interfaces in an interface **definition** language, similar to the approach followed with RPCs. Alternatively, we can make use of an object-based language such as Java that will handle stub generation automatically. This approach of using predefined interface definitions is generally referred to as **static invocation.** Static invocations require that the interfaces of an object are known when the client application is being developed.

As an alternative, method invocations can also be done in a more dynamic fashion. In particular, it is sometimes convenient to be able to compose a method invocation at runtime, also referred to as a dynamic invocation. The essential difference with static invocation is that an application selects at runtime which method it will invoke at a remote object.

**Video**

## MESSAGE-ORIENTED COMMUNICATION

## References


1. **http://www.sanfoundry.com/computer-networks-questions-answers-rpc/**
2. https://www.youtube.com/watch?v=-6Uoku-M6oY&t=556s
3. https://www.youtube.com/watch?v=I_3zU9HeDOs
4. https://www.youtube.com/watch?v=qr7oaiUsxSU
5. https://www.youtube.com/watch?v=ILeAeFZOkMI
6. **https://www.youtube.com/watch?v=dQE6BDNc3z0**
7. Distributed Systems by P K Sinha
8. Distributed Operating Systems by Tanenbaum

**Quiz**

1. An RPC (remote procedure call) is initiated by the
a) server
b) client
c) both (a) and (b)
d) none of the mentioned

2. In RPC, while a server is processing the call, the client is blocked
a) unless the client sends an asynchronous request to the server
b) unless the call processing is complete
c) for the complete duration of the connection
d) none of the mentioned

3. Remote procedure calls is
a) inter-process communication
b) a single process
c) a single thread
d) none of the mentioned

4. RPC allows a computer program to cause a subroutine to execute in
a) its own address space
b) another address space
c) both (a) and (b)
d) none of the mentioned

5. RPC works between two processes. These processes must be
a) on the same computer
b) on different computers connected with a network
c) both (a) and (b)
d) none of the mentioned
6. A remote procedure is uniquely identified by
a) program number
b) version number
c) procedure number
d) all of the mentioned

7. An RPC application requires
a) specific protocol for client server communication
b) a client program
c) a server program
d) all of the mentioned

8. RPC is used to
a) establish a server on remote machine that can respond to queries
b) retrieve information by calling a query

c) both (a) and (b)
d) none of the mentioned

**GQ**

**1.** What is RPC?

**2.** Give the steps occurring in RPC.

**3.** Explain the RPC model.

**4.** Explain the design issues of RPC in detail.

**5.** How to implement RPC mechanism?

**6.** Explain the advantages of RPC.

**7.** Explain transparency in term of RPC. Why is it difficult to achieve? List the limitations of RPC.

**8.** Explain how transparency is achieved in Remote Procedure Calls.

**9.**

**10.** Explain the types of failure that can occur in RPC system

**11.** Write short notes on :

(i) Stub generation
(ii) Extermination
(iii) Reincarnation
(iv) Expiration

**12.** Explain RPC messages.

**13.** How to Marshall Arguments and results?

**14.** Explain the concept of server management.

**15.** Explain server implementation and reaction issues.

**16.** Explain RPC system model in detail.

**17.** Explain the difference between OSI and TCP/IP Model.

**18.** Describe OSI Reference Model with a neat diagram.