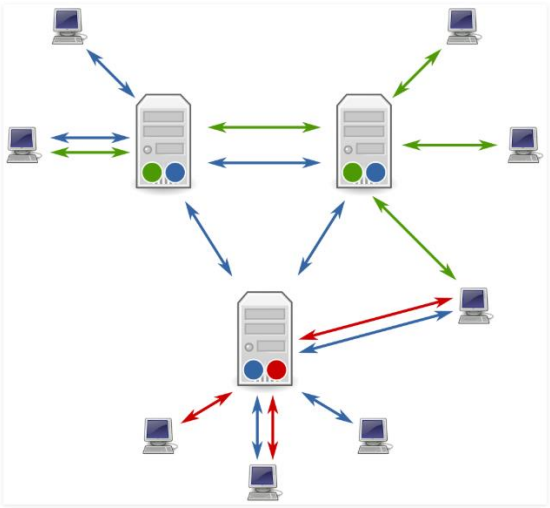
Inter process Communication in Distributed Systems

Inter process Communication is a process of exchanging the data between two or more independent process in a distributed environment.

Having powerful and flexible facilities for communication between processes is essential for any distributed system.

Examples of Inter process Communication:

1. N number of applications can communicate with the X server through network protocols.
2. Servers like Apache spawn child processes to handle requests.
3. Pipes are a form of IPC: grep foo file | sort



It has two functions:

1. **Synchronization:** Exchange of data is done synchronously which means it has a single clock pulse.
2. **Message Passing:** When processes wish to exchange information. Message passing takes several forms such as: pipes, FIFO, Shared Memory, and Message Queues

Characteristics of Inter-process Communication

There are mainly five characteristics of inter-process communication in a distributed environment/system.

1. **Synchronous System Calls**

   In the synchronous system calls both sender and receiver use blocking system calls to transmit the data which means the sender will wait until the acknowledgment is received from the receiver and receiver waits until the message arrives.

2. **Asynchronous System Calls**

   In the asynchronous system calls, both sender and receiver use non-blocking system calls to transmit the data which means the sender doesn't wait from the receiver acknowledgment.

3. **Message Destination**

   A local port is a message destination within a computer, specified as an integer. A port has exactly one receiver but many senders. Processes may use multiple ports from which to receive messages. Any process that knows the number of a port can send the message to it.

4. **Reliability:**

   It is defined as validity and integrity.

5. **Integrity:**

   Messages must arrive without corruption and duplication to the destination.

6. **Validity:**

   Point to point message services are defined as reliable, If the messages are guaranteed to be delivered without being lost is called validity.

7. **Ordering:**

   It is the process of delivering messages to the receiver in a particular order. Some applications require messages to be delivered in the sender order i.e the order in which they were transmitted by the sender.

Remote Procedure Call (RPC) in Operating System

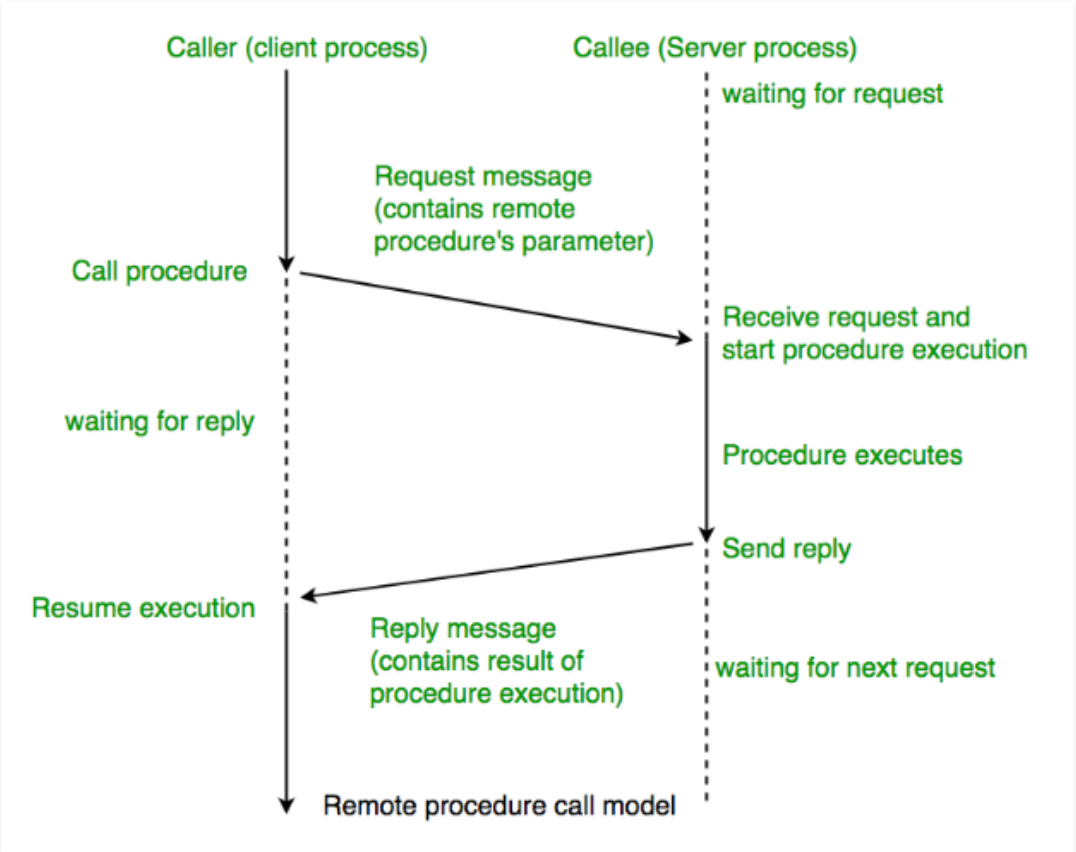
In traditional network applications, communication is often based on the **low-level message-passing primitives** offered by the **transport layer**.

An important issue in **middleware systems** (Distributed systems) is to offer a higher level of abstraction that will make it easier to express communication between processes than the support offered by the interface to the transport layer.

One of the most widely used abstractions is the **Remote Procedure Call (RPC).**

Remote Procedure Call (RPC) is a powerful technique for constructing **distributed, client-server-based applications**. It is based on extending the conventional local procedure calling so that the **called procedure need not exist in the same address space as the calling procedure**. The two processes may be on the same system, or they may be on different systems with a network connecting them.

When making a Remote Procedure Call:



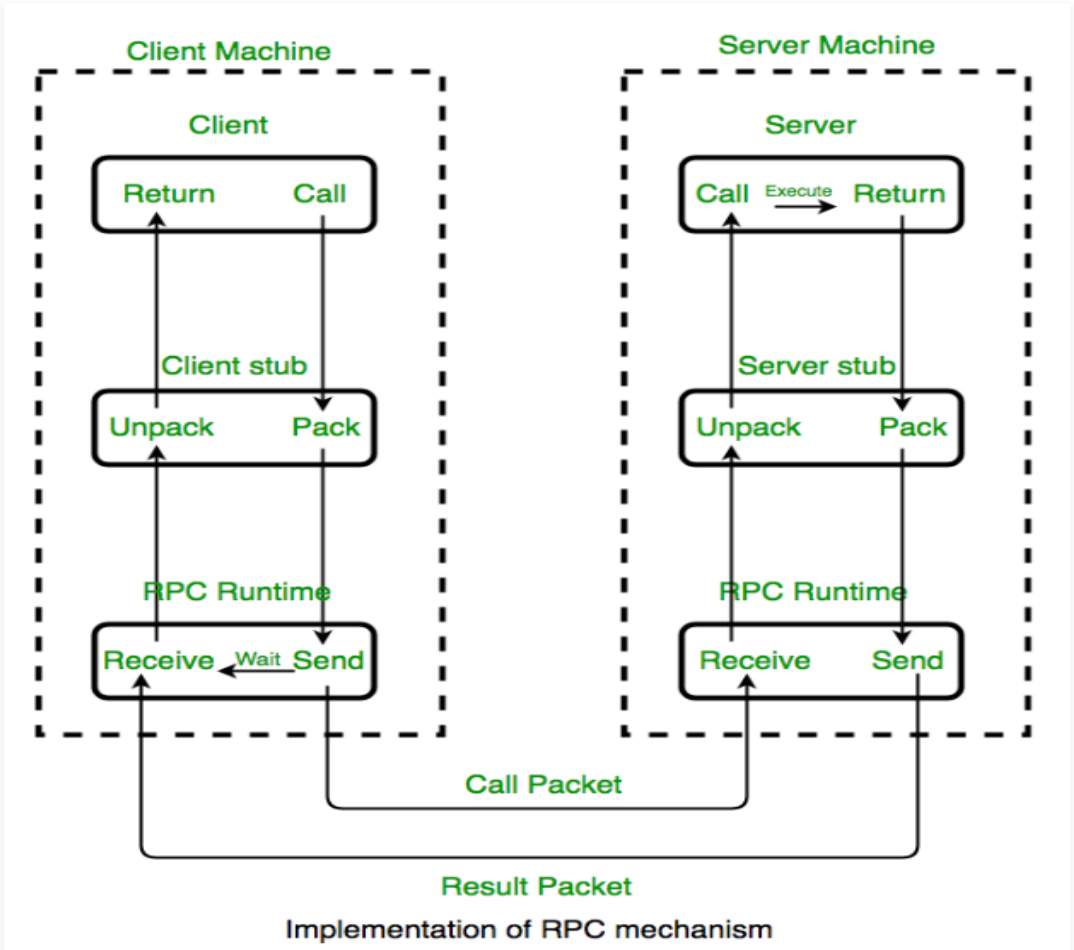
1. The calling environment is suspended, procedure parameters are transferred across the network to the environment where the procedure is to execute, and the procedure is executed there.

2. When the procedure finishes and produces its results, its results are transferred back to the calling environment, where execution resumes as if returning from a regular procedure call.

NOTE:

RPC is especially well suited for client-server **(e.g. query-response)** interaction in which the flow of control **alternates between the caller and callee**. Conceptually, the client and server do not both execute at the same time. Instead, the thread of execution jumps from the caller to the callee and then back again.

Working of RPC



Implementation of RPC mechanism

The following steps take place during a RPC:

1. A client invokes a **client stub procedure**, passing parameters in the usual way. The client stub resides within the client's own address space.

2. The client stub **marshalls (pack)** the parameters into a message. Marshalling includes converting the representation of the parameters into a standard format and copying each parameter into the message.

3. The client stub passes the message to the transport layer, which sends it to the remote server machine.

4. On the server, the transport layer passes the message to a server stub, which **demarshalls (unpack)** the parameters and calls the desired server routine using the regular procedure call mechanism.

5. When the server procedure completes, it returns to the server stub **(e.g., via a normal procedure call return)**, which marshalls the return values into a message. The server stub then hands the message to the transport layer.

6. The transport layer sends the result message back to the client transport layer, which hands the message back to the client stub.

7. The client stub demarshalls the return parameters and execution return to the caller.

RPC ISSUES

1. RPC Runtime: RPC run-time system is a library of routines and a set of services that handle the network communications that underlie the RPC mechanism. In the course of an RPC call, client-side and server-side run-time systems' code handle **binding, establish communications over an appropriate protocol, pass call data between the client and server, and handle communications errors.**

2. Stub: The function of the stub is to **provide transparency to the programmer-written application code**.

On the client side, the stub handles the interface between the client's local procedure call and the run-time system, marshaling and unmarshaling data, invoking the RPC run-time protocol, and if requested, carrying out some of the binding steps.

On the server side, the stub provides a similar interface between the run-time system and the local manager procedures that are executed by the server.

3. Binding: How does the client know who to call, and where the service resides?

The most flexible solution is to use dynamic binding and find the server at run time when the RPC is first made. The first time the client stub is invoked, it contacts a name server to determine the transport address at which the server resides.

Binding consists of two parts:

- Naming:

Remote procedures are named through interfaces. **An interface uniquely identifies a particular service, describing the types and numbers of its arguments**. It is similar in purpose to a type definition in programming languauges.

- Locating:

Finding the transport address at which the server actually resides. Once we have the transport address of the service, we can send messages directly to the server.

A Server having a service to offer exports an interface for it. Exporting an interface registers it with the system so that clients can use it.

A Client must import an (exported) interface before communication can begin.

ADVANTAGES

1. RPC provides **ABSTRACTION** i.e message-passing nature of network communication is hidden from the user.

2. RPC often omits many of the protocol layers to improve performance. Even a small performance improvement is important because a program may invoke RPCs often.

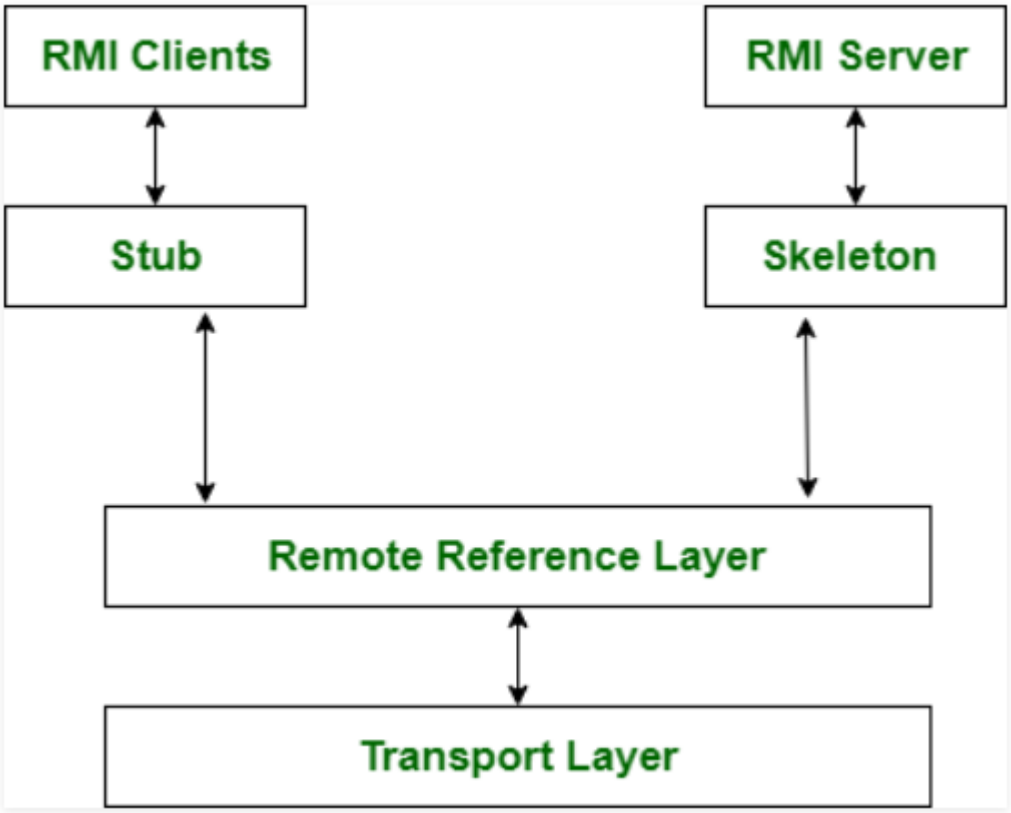
3. RPC enables the usage of the applications in the distributed environment, not only in the local environment.

4. With RPC code re-writing / re-developing effort is minimized.

5. Process-oriented and thread oriented models supported by RPC.

Remote Method Invocation (RMI)

This is a similar to RPC, but it supports **object oriented** programming which is the java's feature. A thread is allowable to decision the strategy on a foreign object. In RMI, objects are passed as parameter rather than ordinary data.



This diagram shows the client-server architecture of RMI protocol.

RPC and RMI both are similar but the basic difference between RPC and RMI is that, RPC supports procedural programming on the other hand, RMI supports object-oriented programming.

Let’s see that the difference between RPC and RMI:

S.NO	RPC	RMI
1.	RPC is a library and OS dependent platform.	Whereas it is a java platform.
2.	RPC supports procedural programming.	RMI supports object-oriented programming.
3.	RPC is less efficient in comparison of RMI.	While RMI is more efficient than RPC.
4.	RPC creates more overhead.	While it creates less overhead than RPC.
5.	The parameters which are passed in RPC are ordinary or nomal data.	While in RMI, objects are passed as parameter.
6.	RPC is the older version of RMI.	While it is the successor version of RPC.
7.	There is high Provision of ease of programming in RPC.	While there is low Provision of ease of programming in RMI.
8.	RPC does not provide any security.	While it provides client level security.
9.	It’s development cost is huge.	While it’s development cost is fair or reasonable.
10.	There is a huge problem of versioning in RPC.	While there is possible versioning using RDMI.
11.	There is multiple codes are needed for simple application in RPC.	While there is multiple codes are not needed for simple application in RMI.

RPCs offer **synchronous communication facilities**, by which a client is blocked until the server has sent a reply. It turns out that **general purpose, high-level message-oriented models** are often more convenient.

In **message-oriented models**, the issues are whether or not communication is **persistent**, and whether or not communication is **synchronous**.

The essence of persistent communication is that a message that is submitted for transmission, is stored by the communication system as long as it takes to

deliver it. In other words, neither the sender nor the receiver needs to be up and running for message transmission to take place.

In **transient** communication, no storage facilities are offered, so that the receiver must be prepared to accept the message when it is sent.

In **asynchronous communication**, the sender is allowed to continue immediately after the message has been submitted for transmission, possibly before it has even been sent. In synchronous communication, the sender is blocked at least until a message has been received. Alternatively, the sender may be blocked until message delivery has taken place or even until the receiver has responded as with RPCs.

Message-oriented middleware models generally offer **persistent asynchronous communication** and are used where **RPCs are not appropriate**. They are often used to assist the integration of (widely-dispersed) collections of databases into large-scale information systems. Other applications include e-mail and workflow.

A very different form of communication is that of streaming, in which the issue is whether or not two successive messages have a temporal relationship. In continuous data streams, a maximum end-to-end delay is specified for each message.

In addition, it is also required that messages are sent subject to a minimum end-to-end delay. Typical examples of such continuous data streams are video and audio streams. Exactly what the temporal relations are, or what is expected from the underlying communication subsystem in terms of quality of service is often difficult to specify and to implement. A complicating factor is the role of jitter.

Even if the average performance is acceptable, substantial variations in delivery time may lead to unacceptable performance.

Finally, an important class of communication protocols in distributed systems is **multicasting**. The basic idea is to disseminate information from one sender to multiple receivers. We have discussed two different approaches.

First, multicasting can be achieved by setting up a tree from the sender to the receivers. Considering that it is now well understood how nodes can self-organize into

peer-to-peer system, solutions have also appeared to dynamically set up trees in a decentralized fashion.

Another important class of dissemination solutions deploys epidemic protocols. These protocols have proven to be very simple, yet extremely robust. Apart from merely spreading messages, epidemic protocols can also be efficiently deployed for aggregating information across a large distributed system.