Inter-Process Communication

Introduction

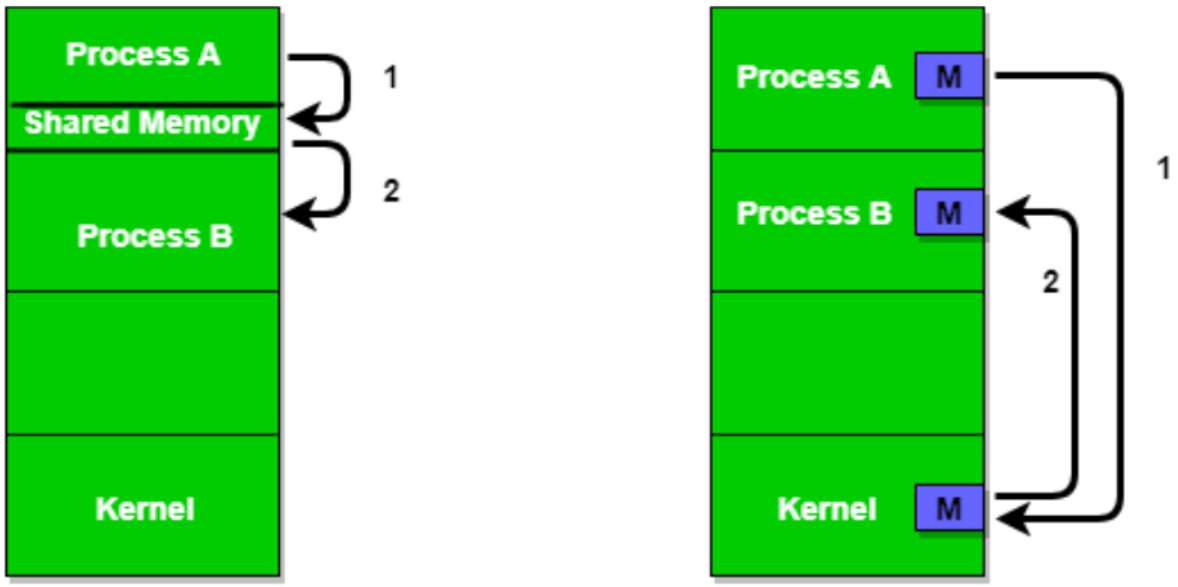


Figure 1 - Shared Memory and Message Passing

- A process can be an Independent process or Co-operating process.
- An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes.
- Processes running independently will execute very efficiently but in practice, there are many situations when cooperative nature can be utilized for increasing computational speed, convenience, and modularity.
- **Inter-process communication (IPC)** is a mechanism that allows processes to communicate with each other and synchronize their actions.
- Processes can communicate with each other in two ways: Shared Memory or Message Passing.

Remote Method Invocation (RMI)

It allows an object to invoke a method in an object in a remote process.

Remote Procedure Call (RPC)

It allows a client to call a procedure on a remote server.

Communication Pattern

The communication patterns that are most commonly used in distributed programs:

- **Client-Server Communication:** The request and reply messages provide the basis for remote method invocation (RMI) or remote procedure call (RPC).
- **Group Communication:**
  - The same message is sent to several processes.
  - Group multicast communication in which one process in a group transmits the same message to all members of the group.

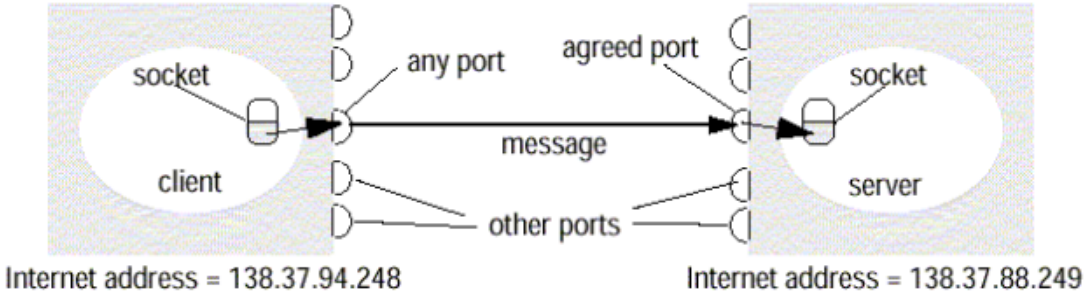
Message Passing

- The application program interface (API) to **UDP** provides a message passing abstraction.
  - Message passing is the simplest form of interprocess communication.
  - API enables a sending process to transmit a single message to a receiving process.
  - The independent packets containing these messages are called D**atagrams**.
  - In the Java and UNIX APIs, the sender specifies the destination using a **socket**.
  - The **socket** is an indirect reference to a particular port used by the destination process at a destination computer.
- The application program interface (API) to **TCP** provides the abstraction of a two-way stream between pairs of processes.
- The information communicated consists of a stream of data items with no message boundaries.

1. The Characteristics of Inter-Process Communication

- **Synchronous and Asynchronous Communication:**
  In the **synchronous** form of communication, the sending and receiving processes synchronize with every message. In this case, both send and receive are blocking operations. Ex: Continuous chatting.
  In the **asynchronous** form of communication, the use of the send operation is non-blocking and the receive operation can have blocking and non-blocking variants. Ex: Whatsapp.
- **Message Destinations:**
  - A local port is a message destination within a computer, specified as an integer.
  - A port has exactly one receiver but can have many senders.
  - Processes may use multiple ports to receive messages.
- **Reliability:**
  - A reliable communication is defined in terms of validity and integrity.
  - A point-to-point message service is described as reliable if messages are guaranteed despite a reasonable number of packets being dropped or lost.
  - For integrity, messages must arrive uncorrupted and without duplication.
- **Ordering:**
  - Some applications require that messages to be delivered in sender order.

2. With a neat diagram explain sockets



- A **socket** can be thought of as an endpoint in a two-way communication channel. Eg: telephone call.

- Sockets are a programming abstraction that is used to implement low-level IPC. For example, two processes exchange information by each process having a socket of its own. They can then read/write from/into their sockets.
- Sockets are created, and if a client-server approach, then the sockets are prepared for sending and receiving messages. With this, we can see that sending data is just done by writing to one's socket, and receiving data is done by reading from the socket.

3. Compare and contrast between sync and async communication in the context of IPC.

Synchronous

In synchronous form of communication, the sender and receiving process synchronise at every message. In this case, both **send** and **receive** are blocking operations. Whenever a **send** is issued the sending process (or thread) is blocked until the corresponding **receive** is issued. Whenever a **receive** is issued by a process (or thread), it blocks until a message arrives.

Asynchronous

In the asynchronous form of communication, the use of the **send** operation is non-blocking in that the sending process is allowed to proceed as soon as the message has been copied to a local buffer, and the transmission of the message proceeds in parallel with the sending process. The **receive** operation can have blocking and non-blocking variants. In the non-blocking variant, the receiving process proceeds with its program after issuing a **receive** operation, which provides a buffer to be filled in the background, but it must separately receive notification that its buffer has been filled, by polling or interrupt.

4. Analyze the failure model of the Request/Reply protocol in the client-server communication using UDP.

A reliable communication channel can be described with two properties, integrity and validity. The integrity property requires that messages should not be corrupt or

duplicated. The use of a checksum ensures that there is a negligible probability that any message received is corrupted. UDP datagrams suffer from the following failures:

- **Omission failures:** Messages may be dropped occasionally, either because of a checksum error or because no buffer space is available at the source or destination.
- **Ordering:** Messages can sometimes be delivered out of sender order.
- UDP lacks built-in checks, but failure can be modelled by implementing an ACK mechanism. Applications using UDP datagrams are left to provide their own checks to achieve the quality of reliable communication they require.

5. Explain Java API for the following: a. Internet Address b. UDP datagrams c. TCP streams

Internet Address

As the IP packets underlying UDP and TCP are sent to an Internet address, Java provides a class, InetAddresss, that represents Internet addresses. Users of this class refer to computers by Domain Name System (DNS) hostnames. For example, instances of InetAddress that contain an Internet address can be created by calling a static method of InetAddress, giving a DNS hostname as the argument. The method uses the DNS to get the corresponding internet address. For example, to get an object representing the internet address of the host whose DNS name is *app.doxandbox.com*, use:

```
InetAddress aComputer = InetAddress.getByName("app.doxandbox.com");
```

UDP Datagrams

The Java API provides datagram communication by means of two classes: *DatagramPacket* and *DatagramSocket*.

- **DatagramPacket:** This class provides a constructor that makes an instance out of an array of bytes comprising a message, the length of the message and the internet address and the local port number of the destination socket. An instance of *DatagramPacket* may be transmitted between processes where one process sends it and another receives it. The *DatagramPacket* provides two methods,

getPort and *getAddress*. These methods are used to access the port and internet address.

- **DatagramSocket:** This class supports sockets for sending and receiving UDP datagrams. It provides a constructor that takes a port number as its argument, for use by processes that need to use a particular port. It also provides a no-argument constructor that allows the system to choose a free local port. These constructors can through a *SocketException* if the chosen port is already in use. The class *DatagramSocket* provides the following methods:
  - **send and receive:** These methods are for transmitting datagrams between a pair of sockets.
  - **setSoTimeout:** This method allows a timeout to be set. With a timeout set, the *receive* method will block for the time specified and throw an *InterruptedIOException*.
  - **connect:** This method is used for connecting to a particular remote port and internet address, in which case the socket is only able to send and receive messages from that address.

TCP Streams

The Java API provides TCP streams communication by means of two classes: *ServerSocket* and *Socket.*

- **ServerSocket:** This class is intended for use by a server to create a socket at a server port for listening for *connect* requests from clients. Its *accept* method gets a *connect* request from the queue or if the queue is empty blocks until one arrives. The result of executing *accept* is an instance of Socket - a socket to use for communicating with the client.
- **Socket:** This class is for use by a pair of processes with a connection. The client uses a constructor to create a socket, specifying the DNS hostname and port of a server. This constructor not only creates a socket associated with a local port but also *connects* it to the specified remote computer and port number. It can throw an *UnknownHostException* if the hostname is wrong or an *IOException* if an IO error occurs.

6. Discuss issues relating to datagram communication.

- **Message size:** The receiving process needs to specify an array of bytes of a particular size in which to receive a message. If the message is too big for the array it is truncated on arrival. The underlying IP protocol allows packet length of up to 216bytes which include the headers as well as the message. However, most environments impose a size restriction of 8 KB. Any application requiring messages larger than the maximum must fragment them into chunks of that size.
- **Blocking:** Sockets normally provide non-blocking *sends* and blocking *receives* for datagram communication. The send operation returns when it has handed the message to the underlying UDP and IP protocols which are responsible for transmitting it to its destination. The method *receive* blocks until a datagram is received unless a timeout has been set on the socket. If the process that involves the *receive* method has other work to do while waiting for the message, it should arrange to use a separate thread.
- **Timeouts:** The *receive* that blocks forever is suitable for use by a server that is waiting to receive requests from its clients. But in some programs, it is not appropriate that a process that has invoked a *receive* operation should wait indefinitely in situations where the sending process may have crashed or the expected message may have been lost. To allow for such requirements, timeouts can be set on sockets. Choosing an appropriate timeout interval is difficult, but it should be fairly large in comparison with the time required to transmit a message.
- **Receive from Any:** The *receive* method does not specify an origin for messages. Instead, an invocation of *receive* gets a message addressed to its socket from any origin. The *receive* method returns the internet address and local port of the sender, allowing the recipient to check where the message came from. It is possible to connect a datagram socket to a particular remote port and internet address, in which case the socket is only able to send and receive messages from that address.

7. Explain characteristics and issues related to stream communication.

Characteristics

- **Message Sizes:** The application can choose how much data it writes to a stream or reads from it. It may deal with very small or very large sets of data.

The underlying implementation of a TCP stream decides how much data to collect before transmitting it as one or more IP packets. On arrival, the data is handled to the application as requested. Applications can, if necessary, force data to be sent immediately.

- **Lost Messages:** The TCP protocol uses an acknowledgement scheme. As an example of a simple scheme, the sending end keeps a record of each IP packet sent and the receiving end acknowledges all the arrivals. If the sender does not receive an acknowledgement within a timeout, it re-transmits the message.
- **Flow Control:** The TCP protocol attempts to match the speed of the processes that read from and write to a stream. If the writer is too fast for the reader, then it is blocked until the reader has consumed sufficient data.
- **Message Duplication and Ordering:** Message identifiers are associated with each IP packet, which enables the recipient to detect and reject duplicates, or to reorder the messages that do not arrive in sender order.
- **Message Destinations:** A pair of communication processes establish a connection before they can communicate over a stream. Once a connection is established, the processes simply read from and write to the stream without needing the use of internet addresses and ports. Establishing a connection involves a *connect* request from the client to the server followed by an *accept* request from the server to the client before any communication can take place. This could be a considerable overhead for a single client-server request and reply.

Issues

- **Matching of Data Items:** Two communicating processes need to agree as to the contents of the data transmitted over a stream. For example, if one process writes an *int* followed by a *double* to a stream, then the reader at the other end must also read an *int* followed by a *double*. When a pair of processes do not cooperate correctly in their use of a stream, the reading process may experience errors when interpreting the data.
- **Blocking:** The data written to a stream is kept in a queue at the destination socket. When a process attempts to read data from an input channel, it will get data from the queue or it will block until data becomes available. During this step, the process that writes data to a stream may be blocked by the TCP flow mechanism.

- **Threads:** When a server accepts a connection, it generally creates a new thread in which to communicate with the new client. The advantage of using a separate thread for each client is that the server can block when waiting for input without delaying other clients.

8. Define Marshalling and Unmarshalling.

Marshalling

Marshalling is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message. The purpose of the data marshalling mechanism is to represent the caller’s argument in a way that can be efficiently interpreted by a server program.

Unmarshalling

Unmarshalling is the process of disassembling a collection of data on arrival to produce an equivalent collection of data items at the destination.

9. Explain CORBA CDR with an example.

CORBA’s Common Data Representation is the external data representation defined with CORBA 2.0. CDR can represent all of the data types that can be used as arguments and return values in remote invocations in CORBA. These consist of 15 primitive types, which include short (16-bit), long (32-bit), unsigned short, unsigned long, float, double, char, boolean, octet, and any.

Example:

example: struct with value {'Smith', 'London', 1934}

index in sequence of bytes	← 4 bytes →	*notes on representation*
0–3	5	*length of string*
4–7	"Smit"	*'Smith'*
8–11	"h___"	
12–15	6	*length of string*
16–19	"Lond"	*'London'*
20–23	"on__"	
24–27	1934	*unsigned long*

Figure 9. CORBA CDR message

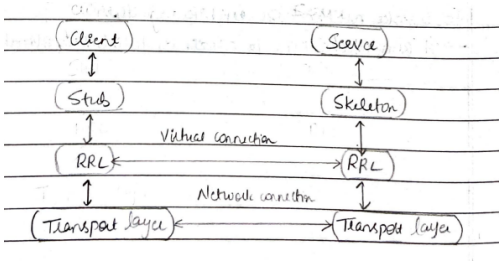
10. Explain JAVA object serialization with an example.

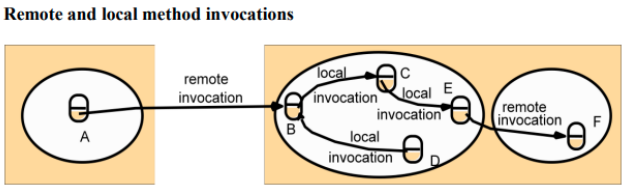
Serialized values				*Explanation*
Person	8-byte version number		h0	*class name, version number*
3	int year	java.lang.String name	java.lang.String place	*number, type and name of instance variables*
1934	5 Smith	6 London	h1	*values of instance variables*

In Java RMI, both object and primitive data values may be passed as arguments and results of method invocation. An object is an instance of a Java class.

Distribution Object and RMI

1. Explain communication between distributed objects by means of RMI.



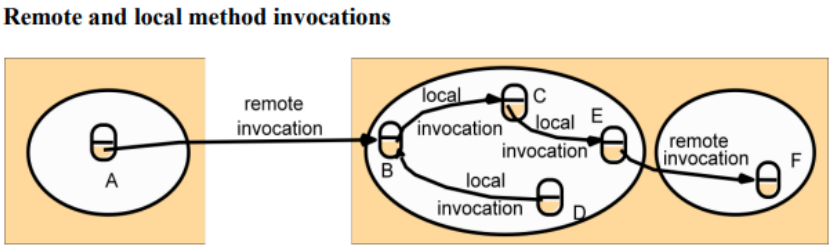


- The term distributed objects usually refer to software modules that are designed to work together but reside either in multiple computers connected via a network or in different processes inside the same computer.
- The state of an object consists of the values of its instance variables.
- Distributed object systems may adopt the client-server architecture. In this case, objects are managed by servers and their clients invoke their methods using Remote Method Invocation (RMI).
- In RMI, the client's request to invoke a method of an object is sent in a message to the server managing the object. The invocation is carried out by executing a method of the object at the server and the result is returned to the client in another message.
- To allow for chains of related invocation, objects in servers are allowed to become clients in other servers.
- **Transport Layer:** Connects client to the server.
- **Stub:** Representation (proxy) of the remote object at the client.
- **Skeleton:** Representation (proxy) of the remote object at the client.
- **RRL (Remote Reference Layer):** Manages references made by the client to the remote object.

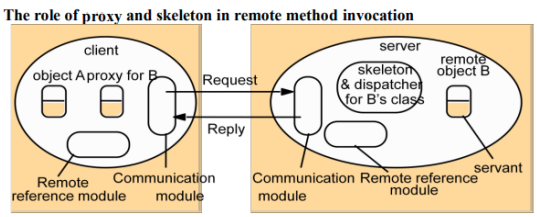
Working of RMI Application

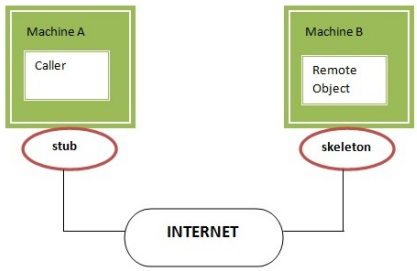
- When a client makes a request to the remote object, it is received by stub and eventually, the request is passed to RRL.
- When client-side RRL receives the request and it calls the method `invoke()` of the object remoteRef. It passes the request to RRL on the server-side.
- The RRL on the server-side passes the request to the skeleton which then finally executes the required object on the server.
- The result is passed all the way back to the client.

2. Explain remote and local invocation with neat diagrams.



3. With a neat diagram explain the role of Proxy & Skeleton in RMI.





RMI uses proxy and skeleton objects for communication with the remote object.
A remote object is an object whose method can be invoked from another JVM.

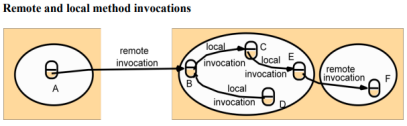
Proxy

- The role of the proxy is to make remote method invocation transparent to clients by behaving like a local object to the invoker; but instead of executing an invocation, it forwards it in a message to the remote object.
- The proxy acts as a gateway for the client-side. All the outgoing requests are routed through it.
- When the caller invokes method on the stub object, it does the following tasks:
  - It initiates a connection with the remote Virtual Machine (JVM)
  - It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM)
  - It waits for the result
  - It reads (unmarshals) the return value or exception
  - And finally, it returns the value to the caller.
- There is one proxy for each remote object for which a process holds a remote object reference.

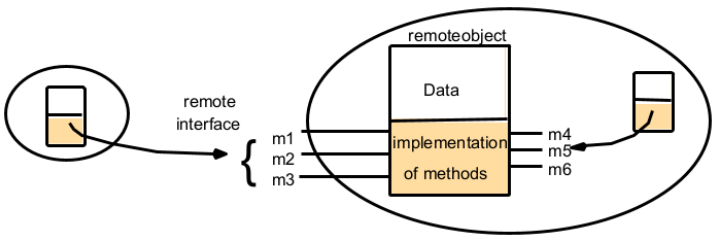
Skeleton

- The skeleton acts as a gateway for the server-side object.
- All the incoming requests are routed through it.
- When the skeleton receives the incoming request, it does the following tasks:
  - It reads (unmarshals) the parameter for the remote method
  - It invokes the method on the actual remote object
  - It writes and transmits (marshals) the result to the caller.

4. Explain the fundamental concepts of the distributed object model.



Remote Method Invocation



Remote Interface`

- Each process contains a collection of objects, some of which can receive both local and remote invocations, whereas the other objects can receive only local invocations.
- Method invocations between objects in different processes, whether in the same computer or not, are known as remote method invocations.
- Method invocations between objects in the same process of the computer are local method invocations and objects that can receive remote invocations are called remote objects.
- There are two fundamental concepts in distributed object model:
    - Remote Object Reference
    - Remote Interfaces
- **Remote Object References:**
  A remote object reference is an identifier that can be used throughout a distributed system to refer to a particular unique remote object. Other objects can invoke the methods of a remote object if they have access to its remote object reference. For example, a remote object reference for B in the figure must be available to A.
    - The remote object receiving a remote method invocation is specified by the invoker as a remote object reference.
    - Remote object references may be passed as an argument.
- **Remote Interfaces:**
    - The class of a remote object implements the methods of its remote interface.
    - It specifies which methods can be invoked remotely.
    - Objects in other processes can invoke only the methods that belong to its remote interface.
    - Remote interfaces, like all interfaces, do not have constructors.
    - The CORBA system provides an **Interface Definition Language (IDL),** which is used for defining remote interfaces.

5. Discuss RMI invocation semantics and tabulate failure handling mechanism for

each.

The RMI invocation semantics are as follows:

Maybe Semantics

- With *maybe* semantics, the remote procedure call may be executed once or not at all.
- Maybe semantics arises when no fault-tolerance measures are applied and can suffer from the following types of failures:
  - Omission failures if the request or result message is lost.
  - Crash failures when the server containing the remote operation fails.
- If the result message has not been received after a timeout and there are no retries, it is uncertain whether the procedure has been executed.

At-least-once Semantics

- With *at-least-once* semantics, the invoker receives wither a result, in which case the invoker knows that the procedure was executed at least once, or an exception informing it that no result was received.
- *At-least-once* semantics can be achieved by the retransmission of request messages.
- *At-lease-once* semantics can suffer from the following types of failures:
  - Crash failures when the server containing the remote procedure fails.
  - Arbitrary failures - in cases when the request message is retransmitted, the remote server may receive it and execute the procedure more than once, possibly causing wrong values to be stored and returned.

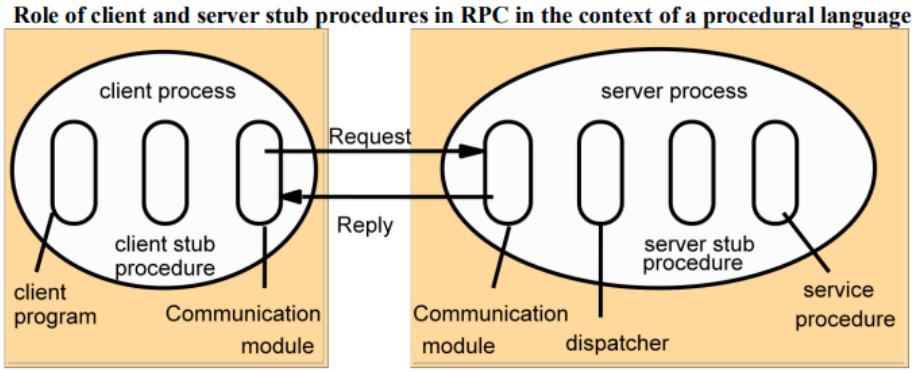
At-most-once Semantics

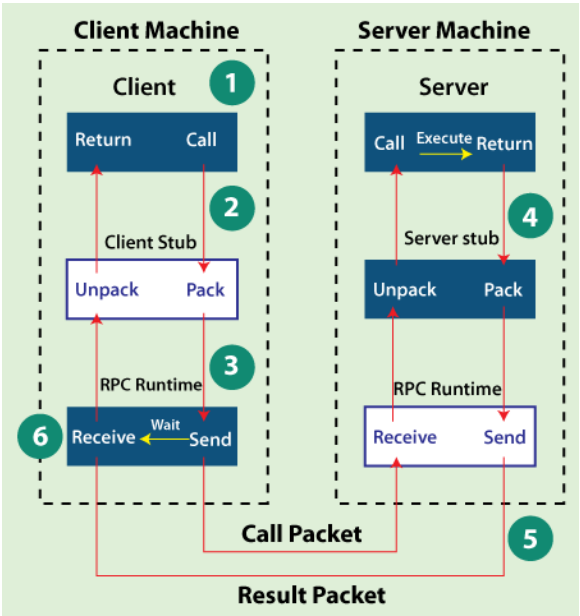
- With *at-most-once* semantics, the caller receives either a result, in which case the caller knows that the procedure was executed exactly once, or an exception informing it that no result was received, in which case the procedure will have been executed either once or not at all.

Failure handling mechanism

Invocation Semantics	Fault tolerance measures	Fault tolerance measures	Fault tolerance measures
	Retransmit request message	Duplicate filtering	Re-execute procedure or retransmit reply
Maybe	No	Not applicable	Not applicable
At-least-once	Yes	No	Re-execute procedure
At-most-once	Yes	Yes	Retransmit reply

6. Define RPC and with a neat diagram explain its implementation.





- The *Remote Procedure Call (RPC)* is a protocol that one program can use to request a service from a program located in another computer in a network

without having to understand network details.

- An RPC is analogous to a function call. Like a function call, when RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure.

Implementation

- The software components required to implement RPC are shown in the figure.
- The client calls the client stub. The call is a local procedure call, with parameters pushed onto the stack in the normal way.
- The client stub packs the parameters into a message and makes a system call to send the message. Packing the parameters is called Marshalling.
- The client's local operating system sends the message from the client machine to the server machine.
- The local operating system on the server machine passes the incoming packets to the server stub.
- The server stub unpacks the parameters from the message. Unpacking the parameters is called Unmarshalling.
- Finally, the server stub calls the server procedure. The reply traces the same steps in the reverse direction.