# TOPIC 3

# REMOTE PROCEDURE CALLS

## **Structure:**

3.1 Introduction to RPC

3.2 Transparency of RPC

3.3 Implementing RPC mechanism

3.4 Stub Generation

3.5 RPC Messages

3.6 Marshaling Arguments and Results

3.7 Server Management

**Learning Activity 3.0**

1.Differentiate between Client Stub and Server Stub

2. Discuss Marshaling Arguments and Results

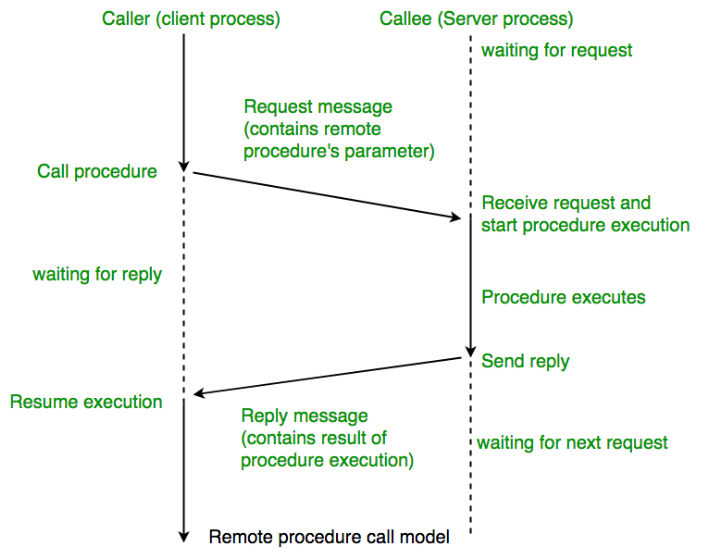
**3.1 INTRODUCTION TO RPC**

A remote procedure call (RPC) is an inter-process communication that allows a computer program to cause a procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction.

It further aims at hiding most of the intricacies of message passing and is idle for client-server application.

Other definition:-

**Remote Procedure Call** (**RPC**) is a protocol that one program can use to request a service from a program located in another computer on a network without having to understand the network's details. A **procedure call** is also sometimes known as a function **call** or a subroutine **call**



RPC allows programs to call procedures located on other machines. But the procedures ‘send’ and ‘receive’ do not conceal the communication which leads to achieving access transparence in distributed systems.

Example: when process A calls a procedure on B, the calling process on A is suspended and the execution of the called procedure takes place. (PS: function, method, procedure difference, stub, 5 state process model definition)

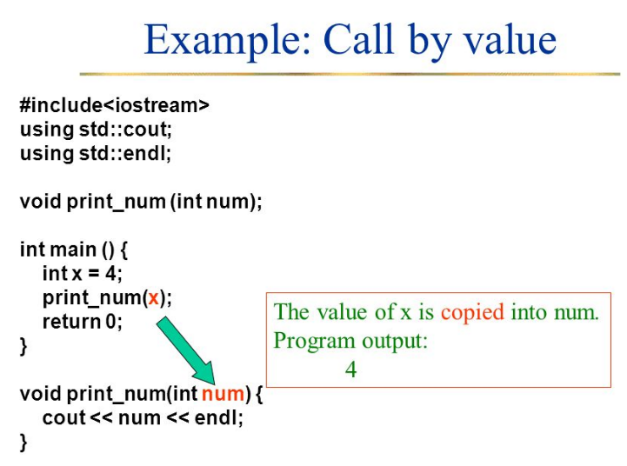
Information can be transported in the form of parameters and can come back in procedure result. No message passing is visible to the programmer. As calling and called procedures exist on different machines, they execute in different address spaces, the parameters and result should be identical and if machines crash during communication, it causes problems.

**3.1.1 RPC Operations:**

**1) Conventional procedure call**

For a call of a program, an empty stack is present to make the call, the caller pushes the parameters onto the stack (last one first order). After the read has finished running, it puts the return values in a register and removes the return address and transfers controls back to the caller. Parameters can be called by value or reference.

* + Call by Value: Here the parameters are copied into the stack. The value parameter is just an initialized local variable. The called procedure may modify the variable, but such changes do not affect the original value at the calling side.



* + Call by reference: It is a pointer to the variable. In the call to Read, the second parameter is a reference parameter. It does not modify the array in the calling procedure.

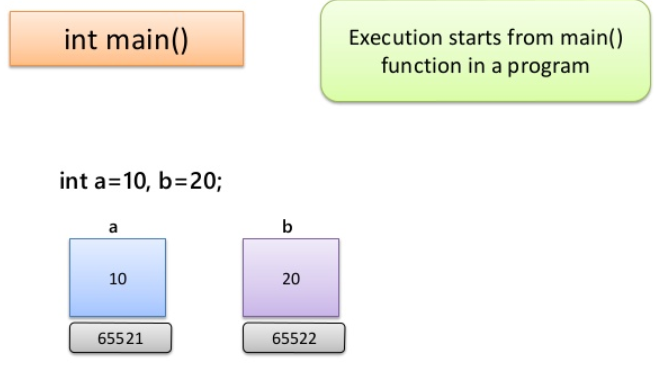
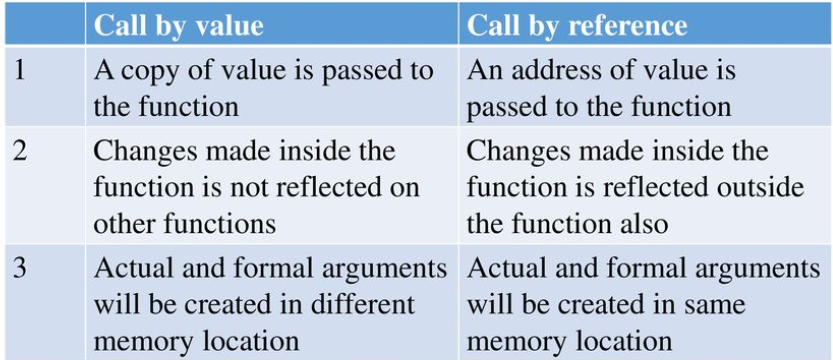
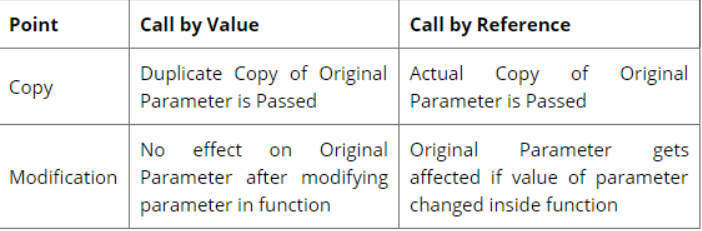


Fig. Function in C (call by reference)

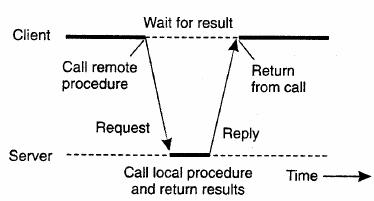
* + Call-by-copy: Another parameter passing mechanism exists along with the above two, its called call-by-copy or Restore. Here the caller copies the variable to the stack and then copies the variable to the stack and then copies it back after the call, overwriting the caller’s original values. The decision of which parameter passing mechanism to use is normally made by the language designers and is a fixed property of the language. Sometimes it depends on the data type being passed.



1. **Client and Server Stubs**
   * A stub in distributed computing is a piece of code used for converting parameters passed during a Remote Procedure Call.
   * The main idea of an RPC is to allow a local computer (client) to remotely call procedures on a remote computer (server). The client and server use different address spaces, so conversion of parameters used in a function call have to be performed; otherwise the values of those parameters could not be used, because of pointers to the computer's memory pointing to different data on each machine.
   * The client and server may also use different data representations even for simple parameters. Stubs are used to perform the conversion of the parameters, so a Remote

Function Call looks like a local function call for the remote computer.

For transparency of RPC, the calling procedure should not know that the called procedure is executing on a different machine.



**Figure 3.1: Principle of RPC between a client and server program.**

* *Client Stub:* Used when read is a remote procedure. Client stub is put into a library and is called using a calling sequence. It calls for the local operating system. It does not ask for the local operating system to give data, it asks the server and then blocks itself till the reply comes.
* *Server Stub*: when a message arrives, it directly goes to the server stub. Server stub has the same functions as the client stub. The stub here unpacks the parameters from the message and then calls the server procedure in the usual way.
* *Summary of the process:*
  1. The client procedure calls the client stub in the normal way.
  2. The client stub builds a message and calls the local operating system.
  3. The client's as sends the message to the remote as.
  4. The remote as gives the message to the server stub.
  5. The server stub unpacks the parameters and calls 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 calls its local as.

1. The server's as sends the message to the client's as.
2. The client's as gives the message to the client stub.
3. The stub unpacks the result and returns to the client.

**3.2 TRANSPARENCY OF RPC**

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:

1. **Syntactic transparency** means that a remote procedures call should have exactly the same syntax as a local procedure call.
2. **Semantic transparency** means that the semantics of a remote procedure call are identical to those of a local procedure call.

It is not very difficult to achieve syntactic transparency of an RPC mechanism, and we have seen that the semantics of remote procedure calls are also analogous to that of local procedure calls for most parts:

The calling process is suspended until the called procedure returns.

The caller can pass arguments to the called procedure (remote procedure).

The called procedure (remote procedure) can return results to the caller.

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.**

1. 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.

According to Bal et al. [1989] dereferencing a pointer passed by the caller has to be done at the caller’s side, which implies extra communication. An alternative implementation is to send a copy of the value pointed at the receiver, but this has subtly different semantics and may be difficult to implement if the pointer points into the middle of a complex data structure, such as a directed graph. Similarly, call by reference can be replaced by copy in / copy out, but at the cost of slightly different semantics.

1. Remote procedure calls are more vulnerable to failure than local procedure calls, since they involve two different processes and possibly a network and two different computers. Therefore programs that make use of remote procedure calls must have the capability of handling even those errors that cannot occur in local procedure calls. The need for the ability to take care of the possibility of processor crashes and communication problems of a network makes it even more difficult to obtain the same semantics for remote procedure calls as for local procedure calls.
2. 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.

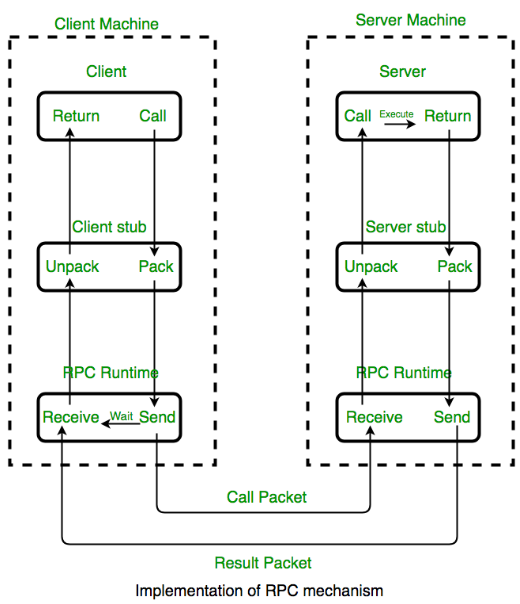
Because of these difficulties in achieving normal call semantics for remote procedure calls, some researchers feel that the RPC facility should be nontransparent. For example, Hamilton [1984] argues that remote procedures should be treated differently from local procedures from the start, resulting in a nontransparent RPC mechanism. Similarly, the designers of RPC were of the opinion that although the RPC system should hide low-level details of message passing from the users, failures and long delays should not be hidden from the caller. That is, the caller should have the flexibility of handling failures and long delays in an application – dependent manner. In conclusion, although in most environments total semantic transparency is impossible, enough can be done to ensure that distributed application programmers feel comfortable.

**3.3 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. We saw that an RPC involves a client process and a server process. Therefore, to conceal the interface of the underlying RPC system from both the client and server processes, a separate stub procedure is associated with each of the two processes. Moreover, to hide the existence and functional details of the underlying network, an RPC communication package (known as RPC Runtime) is used on both the client and server sides. Thus, **implementation of an RPC mechanism** usually involves the following five elements of program [Birrell and Nelson 1984].

1. The client
2. The client stub
3. The RPC Runtime
4. The server stub
5. The server

The interaction between them is shown in Figure 4.2. The client, the client stub, and one instance of RPCRuntime execute on the client machine, while the server, the server stub, and another instance of RPCRuntime execute on the server machine. The job of each of these elements is described below.



**Fig. 3.2**

**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.

**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.

**RPCRuntime:**

The RPCRuntime handles transmission of messages across the network between client and server machines. It is responsible for retransmissions, acknowledgements, 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.

**Server Stub:**

The job of the server stub is very similar to that of the client stub. It performs the following two tasks:

* On the 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.

**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.

Note here that the beauty of the whole scheme is the total ignorance on the part of the client that the work was done remotely instead of by the local kernel. When the client gets control following the procedure call that it made, all it knows is that the results of the procedure execution are available to it. Therefore, as far as the client is concerned, remote services are accessed by making ordinary (local) procedure calls, not by using the send and receive primitives. All the details of the message passing are hidden in the client and server stubs, making the steps involved in message passing invisible to both the client and the server.

**3.4 STUB GENERATION**

Stubs can be generated in one of the following two ways:

1. **Manually:** In this method, the RPC implementer provides a setof translation functions from which a user can construct his or her own stubs. This method is simple to implement and can handle very complex parameter types.
2. **Automatically:** This is the more commonly used method forstub generation. It uses Interface Definition Language (IDL) that is used to define the interface between a client and a server. An interface definition is mainly a list of procedure names supported by the interface, together with the types of their arguments and results. This is sufficient information for the client and server to independently perform compile-time type checking and to generate appropriate calling sequences. However, an interface definition also contains other information that helps RPC reduce data storage and the amount of data transferred over the network. For example, an interface definition has information to indicate whether each argument is input, output, or both – only input arguments need be copied from client to server and only output arguments need be copied from server to client. Similarly, an interface definition also has information about type definitions, enumerated types, and defined constants that each side uses to manipulate data from RPC calls making it unnecessary for both the client and the server to store this information separately.

A server program that implements procedures in an interface is said to export the interface and a client program that calls procedures from an interface is said to import the interface. When writing a distributed application, a programmer first writes an interface definition using the IDL. He or she can then write the client program that imports the interface and the server program that exports the interface. The interface definition is processed using an IDL computer to generate components that can be combined with client and server programs, without making any changes to the existing compliers. In particular, from an interface definition, an IDL complier generate a client stub procedure and a server such procedure for each procedure is the interface, the appropriate marshaling and un-marshaling operations (described later in this chapter) in each stub procedure, and a header file that supports the data types in the interface definition. The header file is included in the source files of both the client and server programs, the client stub procedures are complied and linked with the client program, and the server stub procedures are compiled and linked with the server program. An IDL compiler an be designed to process interface definitions for use with different languages, enabling clients and servers written in different languages, to communicate by using remote procedure calls.

**3.5 RPC MESSAGES**

Any remote procedure call involves a client process and a server process that are possibly located on different computers. The mode of interaction between the client and server is that the client asks the server to execute a remote procedure and the server returns the result of execution of the concerned procedure to the client. Based on this mode of interaction, the two types of messages involved in the implementation of an RPC system are as follows :

1. Call messages that are sent by the client to the server for requesting execution of a particular remote procedure.
2. Reply messages that are sent by the server to the client for returning the result of remote procedure execution.

The protocol of the concerned RPC system defines the format of these two types of message. Normally, an RPC protocol is independent of transport protocols. That is, RPC does not care how a message is passed from one process to another. Therefore an RPC protocol deals only with the specification and interpretation of these two types of messages.

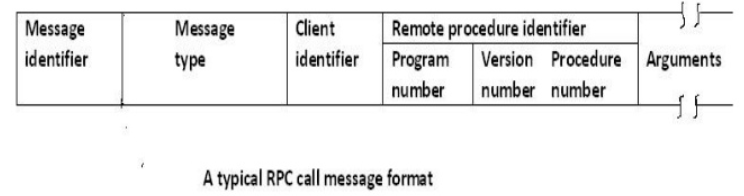
**Call Messages:**

Since a call message is used to request execution of a particular remote procedure the two basic components necessary in a call message are as follows:

1. The identification information of the remote procedure to be executed.
2. The arguments necessary for the execution of the procedure.

In addition to these two fields, a call message normally has the following fields.

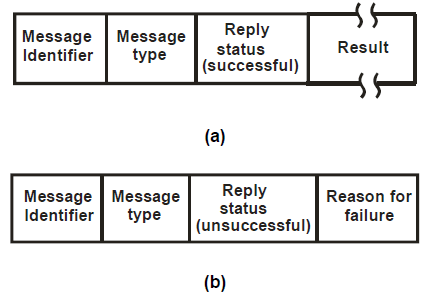
1. *A message identification field* that consists of a sequence number. This field is useful of two ways – for identifying lost messages and duplicate messages in case of system failures and for properly matching reply messages to outstanding call messages, especially in those cases when the replies of several outstanding call messages arrive out of order.
2. *A message type field* that is used to distinguish call messages from reply messages. For example, in an RPC system, this field may be set to 0 for all call messages and set to 1 for all reply messages.
3. *A client identification field* that may be used for two purposes – to allow the server of the RPC to identify the client to whom the reply message has to be returned and to allow the server to check the authentication of the client process for executing the concerned procedure.



Thus, a typical RPC all message format may be of the form shown in Figure 3.2.

**Reply Messages:**

When the server of an RPC receives a call message from a client, it could be faced with one of the following conditions. In the list below, it is assumed for a particular condition that no problem was detected by the server for any of the previously listed conditions:



**Fig. 3.3 A typical RPC reply message format : (a) a successful reply message format; (b) an unsuccessful reply message format**

**39**

**3.6 MARSHALING ARGUMENTS AND RESULTS**

Implementation of remote procedure calls involves the transfer of arguments from the client process to the server process and the transfer of results from the server process to the client process. These arguments and results are basically language-level data structures (program objects), which are transferred in the form of message data between the two computers involved in the call. The transfer of message data between two computers requires encoding and decoding of the message data. For RPC this operation is known as marshaling and basically involves the following actions.

1. Taking the arguments (of a client process) or the result (of a server process) that will form the message data to be set to the remote process.
2. Encoding the message data of step 1 above on the sender’s computer. This encoding process involves the conversion of program objects into a stream form that is suitable for transmission and placing them into a message buffer.
3. Decoding of the message data on the receiver’s computer. This decoding process involves the reconstruction of program objects from the message data that was received in stream form.

In order that encoding and decoding of an RPC message can be performed successfully, the order and the representation method (tagged or untagged) used to marshal arguments and results must be known to both the client and the server of the RPC. This provides a degree of type safety between a client a server because the server will not accept a call from a client until the client uses the same interface definition as the server. Type safety is of particular importance to servers since it allows them to survive against corrupt call requests.

The marshaling process must reflect the structure of all types of program objects used in the concerned language. These include primitive types, structured types, and user defined types. Marshaling procedures may be classified into two groups:

1. Those provided as a part of the RPC software. Normally marshaling procedures for scalar data types, together with procedures to marshal compound types built from the scalar ones, fall in this group.

**40**

1. Those that are defined by the users of the RPC system. This group contains marshaling procedures for user – defined data types and data types that include pointers. For example, in Concurrent CLU, developed for use in the Cambridge Distributed Computer System, for user-defined types, the type definition must contain procedures for marshaling.

A good RPC system should always generate in-line marshaling code for every remote call so that the users are relieved of the burden of writing their own marshaling procedures. However, practically it is difficult to achieve this goal because of the unacceptable large amounts of code that may have to be generated for handling all possible data types.

**3.7 SERVER MANAGEMENT**

In RPC based applications, two important issues that need to be considered for every management are server implementation and server creation.

**Server Implementation:**

Based on the style of implementation used, servers may be of two types :

1. Stateful and
2. Stateless

**Stateful Servers:**

A stateful server maintains clients’ state information from one remote procedure call to the next. That is, in case of two subsequent calls by a client to a stateful server, some state information pertaining to the service performed for the client as a result of the first call execution is stored by the server process. These clients’ state information is subsequently used at the time of executing the second call.

For example, let us consider a server for byte-stream files that allows the following operations on files :

**Open (filename, mode) :** This operation is used to open a fileidentified by filename in the specified mode. When the server executes this operation, it creates an entry for this file in a file-table that it uses for maintaining the file state information of all the open files. The file state information normally consists of the identifier of the file, the open mode, and the current position of a nonnegative integer pointer, called the read write pointer. When a file is opened, its read-write pointer is set to zero and the server returns to the client a file identifier (fid), which is used by the client for subsequent accesses to that file.

**41**

**Read (fid, n, buffer) :** This operation is used to getnbytes of datafrom the file identified by fid into the buffer named buffer. When the server executes this operation, it returns to the client n bytes of file data starting from the byte currently addressed by the read – write pointer and then increments the read – write pointer by n.

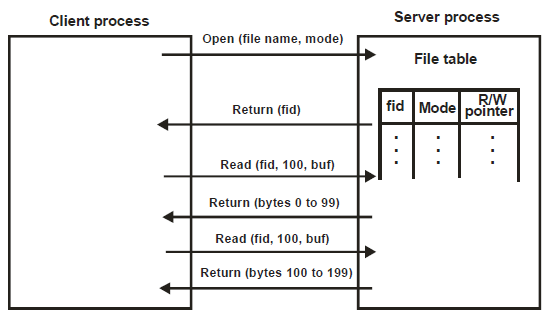
**Write (fid, n, buffer) :** On execution of this operation, the servertakes n bytes of data from the specified buffer, writes it into the file identified by fid at the byte position currently addressed by the read

– write pointer, and then increments the read – write pointer by n.

**Seek (fid, position ) :** This operation causes the server to changethe value of the read write pointer of the file identified by fid to the new value specified as position.

**Close (fid) :** This statement causes the server to delete from its filetable the file state information of the file identified by fid.

The file server mentioned above is stateful because it maintains the current state information for a file that has been opened for use by a client. Therefore, as shown in Fig. 3.3, after opening a file, if a client makes two subsequent Read (fig, 100, buf), calls, the first call will return the first 100 bytes (bytes 0 – 99) and the second call will return the next 100 bytes (bytes 100 – 199).



**Fig. 3.3 An example of a stateful file server**

To keep track of the current record position for each client that has opened the file for accessing. Therefore to design an idempotent interface for reading the next record from the file, it is important that each client keeps track of its own current record position and the server is made stateless, that is, no client state should be maintained on the server side. Based on this idea, an idempotent procedure for reading the next record from a sequential file is

**ReadRecordN (Filename, N)**

which returns the Nth record from the specified file. In this case, the client has to correctly specify the value of n to get desired record from the file.

However, not all non idempotent interfaces can be so easily transformed to an idempotent form. For example, consider the following procedure for appending a new record to the same sequential file.

**AppendRecord (Filename, Record)**

It is clearly not idempotent since repeated execution will add further copies of the same record to the file. This interface may be converted into an idempotent interface by using the following two procedures instead of the one defined above :

**GetLastRecordNo (Filename)**

**WriteRecordN (Filename, Record, N)**

The first procedure returns the record number of the last record currently in the file, and the second procedure writes a record at specified in the file. Now, for appending a record, the client will have to use the following two procedures:

**Last = GetLastRecordNo (Filename)**

**WriteRecordN (Filename, Record, Last)**

For exactly-once semantics, the programmer is relieved of the burden of implementing the server procedure in an idempotent manner because the call semantics itself takes care of executing the procedure only once. The implementation of exactly-once call semantics is based on the use of timeouts, retransmissions, call identifiers with the same identifier for repeated calls, a reply cache associated with the callee.

**Revision Exercise:**

1. What is the primary motivation for development of RPC?
2. What is the main difference between RPC model and an ordinary procedure call model?
3. What is a stub? How are they generated? State their functionality and purpose.
4. What are the issues in developing a transparent RPC mechanism?

**REFERENCES:**

1. Ghosh, Sukumar (2007), Distributed Systems – An Algorithmic Approach, Chapman & Hall/CRC, [ISBN](https://en.wikipedia.org/wiki/International_Standard_Book_Number) [978-1-58488-564-1](https://en.wikipedia.org/wiki/Special:BookSources/978-1-58488-564-1).

[Lynch, Nancy A.](https://en.wikipedia.org/wiki/Nancy_Lynch) (1996), Distributed Algorithms, [Morgan Kaufmann](https://en.wikipedia.org/wiki/Morgan_Kaufmann_Publishers), [ISBN](https://en.wikipedia.org/wiki/International_Standard_Book_Number) [1-55860-348-4](https://en.wikipedia.org/wiki/Special:BookSources/1-55860-348-4).