Five elements of an RPC implementation

Client: To invoke a remote procedure, a client makes a perfectly local call that invokes the corresponding procedure in the stub

Client Stub: Tasks of the client stub

- On receipt of a call request from the client, it packs the specification of the target procedure and the arguments into a message and asks the local runtime system to send it to the server
- On receipt of the result of procedure execution, it unpacks the result and passes it to the client
Five elements of an RPC implementation...

**RPCRuntime: Functions**

- Handles the transmission of the messages across the network between client and server machines.
- Responsible for retransmissions, acks, and encryption.

- On the client side,
  - it receives the call request from the client stub and sends it to the server machine
  - it also receives reply message (result of procedure execution) from the server machine and passes it to the client stub.

- On the server side,
  - it receives the results of the procedure execution from the server stub and sends it to the client machine.
  - it also receives the request message from the client machine and passes it to the server stub.
Five elements of an RPC implementation...

**Server Stub:** Functions of server stub similar to client stub

- unpacks the call receipt messages from local RPCRuntime and makes a perfect local call to invoke the appropriate procedure in the server
- packs the results of the procedure execution received from server, and asks the local RPCRuntime to send it to the client stub.

**Server:** upon receiving the call request from the server stub, the server executes the appropriate procedure and returns the result to the server stub

How RPC based system is different from message passing system:

- client does not use send and receive primitives to access the remote services
- remote services are accessed by making ordinary local procedure calls
- all details of message passing are hidden in the server and client stubs
STUB Generation

Two ways in which stubs can be generated

**Manual Stub Generation:** RPC implementer provides a set of translation functions from which user can construct his own stubs. Simple to implement and can handle complex parameters.

**Automatic Stub Generation:** More commonly used technique

- Uses an Interface Definition Language (IDL), for defining the interface between the client and server. Some features of an IDL:
  - It is mainly a list of procedure names supported by the interface, together with the types of their arguments and results, which helps the client and server to perform compile-time type checking and generate appropriate calling sequences
  - An interface definition also contains information to indicate whether each argument is an input, output or both. This helps in unnecessary copying - input argument needs to be copied from client to server and output needs to be copied from server to client.
  - It also contains information about type definitions, enumerated types, and defined constants - so the clients do not have to store this information.

- A server program that implements procedures in an interface is said to export the interface
• A client program that calls the procedures is said to **import** the interface

• When writing a distributed application, a programmer
  – first writes the interface definition using IDL
  – then can write a server program that exports the interface and a client program that imports the interface.

• The interface definition is processed using an IDL compiler (the IDL compiler in Sun RPC is called `rpcgen`) to generate components that can be combined with both client and server programs, without making changes to the existing compilers. More specifically
  – an IDL compiler generates a client stub procedure and a server stub procedure for each procedure in the interface
  – It generates the appropriate marshaling and unmarshaling operations in each sub procedure
  – It also generates a header file that supports the data types in the interface definition to be included in the source files of both client and server.
  – The client stubs are compiled and linked with the client program and the server stubs are compiled and linked with server program
RPC messages

Two types of messages are involved in the implementation of the RPC system

**Call messages:** sent by the client to server for requesting execution of particular remote procedure.

Components of a call message:

- identification information of the remote procedure to be executed - such as program number, version number, and procedure number
- arguments necessary for the execution of the procedure
- a message identification field that consists of a sequence number
- a message type to distinguish call and reply messages
- a client identification field

**Reply messages:** sent by the server to the client for returning the result

When a server receives a call message, the server could face with one of the following conditions

- The message is not intelligible to it. Could be because the call message violates the RPC protocol. Server needs to discard such calls.
- If the server finds the client is not authorized to use the service, the requested service is not available, or
an exception condition such as division by 0 occurs then it will return an appropriate unsuccessful reply.

• If the specified remote procedure is executed successfully, then it sends a reply

Marshaling

• Marshaling involves encoding and decoding of the arguments of reply and call messages
Server Management

Server Implementation

**Stateful servers:** Characteristics

- maintains state information from one call to another call. For example, let us consider a server that supports the following operations for files:

  **Open (filename, mode):** used to open filename in specified mode. When the server executes this operation, it creates an entry for this file in a *file-table* that is used for maintaining state information.

  **Read(fid, n, buffer):** This operation returns n bytes of file data starting from the byte currently addressed by the *read-write* pointer and then increments the pointer by n.

  **Write(fid, n, buffer):** The server takes n bytes of data from the buffer and writes to the file identified by the *read-write* pointer.

  **Seek(fid, position):** causes to change the value of the read pointer.

  **Close(fid):** causes the server to delete the file state information from the file-table.
Server Implementation...

Stateless Server: A stateless server does not maintain any client state information. So every request must accompany with all the necessary parameters. Some operations that a stateless file server can support

**Read(filename, position, n, buffer):** Read n bytes from the file from position

**Write(filename, position, n, buffer):** Write n bytes from buffer to file starting at position

Advantages and disadvantages of stateless server.

- **Advantages of a stateful server**
  - provides an easier programming paradigm
  - typically more efficient than stateless servers

- **Disadvantages of stateful server**
  - If the server crashes and restarts, the state information it was holding may be lost and the client may produce inconsistent results
  - If the client process crashes and restarts, the server will have inconsistent information about the client

- **Handling failure under stateless server is easy**
  - When server crashes and restarts, it does not result in any inconsistencies
– When client crashes and restarts, it does not lead to any inconsistencies either

• How do we choose which approach to take while designing servers
  – depends on the application
Server Creation Semantics

Based on the time duration for which the RPC servers survive, they can be classified as follows:

**Instance-per-call-servers:** They exist only for the duration of a single call.

- Such a server is created by the RPCRuntime when the call arrives
- The server is deleted when the call has been executed
- Not a commonly used semantics because
  - These servers are stateless. Any state that has to preserved across calls should be handled by the OS
  - The overhead involved in the creation and destruction of servers is expensive, especially if it is for the same type of service.
Server Creation Semantics...

**Instance-per-Session-Servers:** Servers belonging to this category exist for the entire session for which the client and server interact

- These servers can maintain state information across calls
- Overhead involved in the server creation for each call does not exist
- Under this approach
  - There is a server manager for each type of service
  - All the server managers register with the binding agent (later)
  - Client first contacts the binding agent with the type of service needed
  - The binding agent returns to the client the address of the server manager that provides that type of service
  - Client contacts the server manager to create a server for it
  - Server manager spawns a server and returns the address of the server to the client
  - Client then interacts with this server for the entire session
  - The server is destroyed when the client informs back the server manager of the corresponding type that the server is no longer needed
Server Creation Semantics...

Persistent Servers: Characteristics of persistent servers

- Generally remains indefinitely
- Usually shared by many clients
- Servers are created and installed before the clients
- Each server independently exports its service by registering itself with the binding agent
- When a client contacts the binding agent for a particular service, the binding agent selects a server of that type and returns its address to the client
- The client then interacts with the server.
- An advantage of this approach is it can improve performance, since it interleaves requests of several clients
- Care should be taken in designing the procedures so that interleaved concurrent requests from different clients do not interfere with each other.
Parameter-Passing Semantics...

**Call-by-Value:** All parameters are copied into a message that is transmitted

- It does not pose problem for simple data-types such as integers, small arrays and so on
- Passing large data-types like multi-dimensional arrays, trees, etc. can consume much time for transmission of data that may not be used

**Call-by-Reference:** This is possible only in a distributed shared memory system.

- It is also possible in object-based systems, because in this case client needs to pass the names of objects, which are like reference. In object-based systems it is called *call-by-object-reference*.
- A remote invocation operation may cause another remote invocation, etc. To avoid many remote references, another parameter-passing mode, called *call-by-move* was proposed; in this approach, the object to which a reference is made is moved to the site of the callee first and then executed.
Call Semantics…

As we saw earlier, the following types of failures can occur

- The call message gets lost
- The response message gets lost
- The callee node crashes and is restarted
- The caller node crashes and is restarted

Mechanisms for handling such failures

- The RPCRuntime should be designed to provide flexibility to the application programmers to select from different possible call semantics supported by an RPC system

**Possibly or May-be Call Semantics:** The weakest semantics. Not really appropriate for RPC
  - Caller waits until a predetermined amount of time and then continues with the execution
  - Suitable in an environment that has high probability of successful transmission of messages

**Last-One Call Semantics:** The calling of remote procedure by the caller, execution of procedure by callee, and return of the result to the caller will eventually be repeated until the result of the procedure execution is received by the caller. i.e., the results of the last executed call are used by the caller.
– Easy to achieve if only two processors are involved. For example, A process P1 on N1 calls F1 on N2 and F1 calls F2 on N3; N1 fails and restarts; P1’s call to F1 will be repeated which in turn will call F2 again; N3 is unaware of N1’s failure; so N3 may send the result of the two executions in any order, violating last-one semantics.

– Above problem occurs due to orphan calls. An orphan call is one whose parent is dead due to node crash. To achieve last-one semantics, these parent calls must be terminated before restarting

**Last-of-Many Call Semantics:** Similar to last-one semantics, except orphan calls are neglected.

– Call identifiers are used to uniquely identify each call.

– When a call is repeated, it is assigned a new identifier

– Each response message has the corresponding call identifier

– A caller accepts a response only if the call identifier matches with that of the most recently repeated call

**At-Least-Once Call Semantics:** Weaker than last-of-many. It just guarantees that the call is executed more than once, but does not specify which result will be returned.

**Exactly-Once Call Semantics:** Strongest and most desirable semantics. This eliminates the possibility of a procedure being executed more than once no matter how many times a call is retransmitted.
Communication Protocols for RPCs

The Request Protocol: Also known as the R protocol

• Useful for RPCs in which the called procedure has nothing to return and the client does not require confirmation for the procedure having been executed

• An RPC protocol that uses R protocol is also called asynchronous RPC

• For asynchronous RPC, the RPCRuntime does not take responsibility for retrying a request in case of communication failure.

• So, if an unreliable transport protocol such as UDP is used, then request messages could be lost.

• Asynchronous RPCs with unreliable transport protocols are generally useful for implementing periodic updates. For example, a time server node in a distributed system, may send synchronization messages every $T$ seconds.

Request/Reply Protocol (RR protocol): Basic idea is to eliminate acks.

• A server's reply message is regarded as an acknowledgment of the client's request

• A subsequent call message is regarded as an ack for the server's reply

• RR protocol does not possess failure-handling capabilities
- A timeout and retry is normally used along with RR protocol, for taking care of lost messages

- If duplicate messages are not filtered, RR protocol provides at least once semantics

- Servers can support exactly-once semantics by keeping records of replies in a reply cache. How long the reply needs to be kept?

**The Request/Reply/Acknowledge-Reply Protocol (RRA):**

The server needs to keep a copy of the reply only until it receives the ack for reply from client. Exactly-once semantics can be implemented easily using this protocol.
Complicated RPCs

Two types of complicated RPCs and how to handle them

- RPCs involving long-duration calls or large gaps between calls. How to handle such calls?

  **Periodic probing of the server by client:** After a client sends a request, it periodically sends a probe packet which the server acks.
  - periodic probing helps the client to detect server crash or communication link failure
  - acks to probe message can also contain information about lost request in which case the client can retransmit.

  **Periodic generation of acks by server:** The server itself generates acks periodically and sends to client before sending reply; longer the time it takes to send a reply, more will be the number of acks generated.

- RPCs involving arguments and/or results that are too large to fit in a single datagram packet. How to handle such calls?
  - Use several physical RPCs for one logical RPC
  - Use multi-datagram messages. i.e., RPC argument is fragmented and transmitted in multiple packets.
  - For example, Sun RPC is limited to 8 KB. So RPC's involving larger than allowed limit must be handled by breaking into several physical RPCs.
Client-Server Binding

What is binding?

- The process by which a client becomes associated with a server so that calls can take place is known as binding. Client-server binding involves handling of several issues:
  - How does a client specify a server to which it wants to get bound?
  - How does the binding process locate the specified server?
  - When is it proper to bind a client to server?
  - Is it possible for a client to change a binding during execution?
  - Can a client be simultaneously bound to multiple servers that provide the same service?
Client-Server Binding...

**Server Naming:** Birrell and Nelson’s proposal

- Use *interface name* for this purpose
- An *interface name* has two parts - a *type* and an *instance*. Type specifies the interface itself, and instance specifies a server providing the services within that interface.
  - For example, there may be an interface type *file_server*, and there may be many instances of servers providing file service
- Type part also has generally version number field to distinguish between old and new versions of interface (which may provide different sets of service)
- Interface names are created by users.
- The RPC package only dictates the means by which an importer uses the interface name to locate an exporter.

**Server Locating:** Common methods used for locating

**Broadcasting:** A broadcast message is sent to locate the server. The first server responding to this message is used by the client. OK for small networks.

**Binding Agent:** A binding agent is basically a name server used to bind a client to a server by providing information about the desired server.
  - The binding agent maintains a binding table which is a mapping of the server’s interface name to its locations
- All servers register themselves with the binding agent as a part of their initialization process.

- To register, the server gives the binder its identification information and a handle to look at it, for example IP address.

- Server can deregister when it is no longer prepared to offer this service

- Binding agent’s location is known to all nodes

- Binding agent interface has three primitives: register, deregister, and lookup (used by client).

**Binding Time:** When can a client be bound to a server

**Binding at Compile Time:** The client and server modules are programmed as if they were linked together. For example server’s network address can be compiled into client’s code.

- Very inflexible because if the server moves or the server is replicated or the interface changes, all client programs need to be recompiled

- However, it is useful in an application whose configuration is expected to last for fairly long time.

**Binding at Link Time:** A server exports its service by registering with the binding agent as part of the initialization process

- A client then makes an import request to the binding agent before making a call

- The binding agent binds the client and server by returning the server’s handle.
• The server’s handle is cached by client to avoid contacting the binding agent.

**Binding at Call Time:** A client is bound to a server at the time when it calls the server for the first time during execution.

**Indirect Call Method:** When a client calls the server for the first time,

• it passes the server’s interface name and the arguments of the RPC call to the binding agent.

• the binding agent looks up the location of the target’s server and forwards the RPC message to it

• When the target server returns the results to the binding agent, the binding agent returns the result along with the handle of the target server to the client.

• The client subsequently can call target server directly.
Security

Security Issues:

• Is the authentication of the server by the client required?

• Is the authentication of client by server required?

• Is it alright if the arguments and results are accessible to users other than the caller and the callee?

Security is an important issue. More about security later?
Some Special Type of RPCs

Callback RPC: How it works

- In the usual RPC there is a client-server relationship between callee and caller.
- The callback RPC facilitates peer-to-peer paradigm among participating processes.
- The server, before sending the results to the callee can make a callback RPC to the client process.
- The client process takes necessary action based on server’s request and returns reply to the server.
- Before returning result, server could make several callback RPCs.
- The following are necessary for supporting callback RPC facility:
  - Providing the server with client’s handle: The client uses a transient program number for the callback service and exports the callback service by registering the program number with the binding agent.
    - The program number is sent then as part of the RPC request.
    - To make a callback RPC, the server initiates a normal RPC request to the client using the given program number.
    - The client could also send a handle such as port number instead of program number.
**Making the client wait for callback RPC:** The client process must wait for callback RPC

- To wait for the callback, a client process normally makes a call to a `service_routine` and the `service_routine` waits until it receives request and then dispatches the request to appropriate procedure.

**Handling callback deadlocks:** Callback deadlocks can occur

P1 makes a call to P2 and P2 makes a call to P3 and P3 makes a call to P1, and we have a circular wait.

**Broadcast RPC:** Broadcast RPC can use the following two methods

- A client’s request is broadcast on the network and is processed by all servers that have a procedure for processing the request.

Two methods for broadcasting:

- Method1: Client sends the request to the binding agent and the binding agent forwards it to all servers registered with it.

- Method 2: A network port (a queuing point on the node for broadcast messages) of each node is connected to a broadcast port. The client first obtains a binding for the broadcast port and then broadcast RPC message by sending the RPC message to this port.
**Batch-Mode RPC:** Separate RPC messages are queued in a transmission buffer on the client side and then sent over the network in one batch to the server.

Advantages:

- Reduces the overhead in sending requests
- Applications requiring higher call rates (50-100 calls per second) may not be feasible with most RPC implementations. This approach can be useful in such scenarios.
RPC in Heterogeneous Environments

**Data Representation:** Machines may use different data representations. For example, little Endian, big Endian representations.

**Transport Protocol:** For portability, an RPC system should be independent of underlying network transport protocols.

**Control Protocol:** For better portability, an RPC must also be independent of the underlying network control protocol that defines the control information in each packet to track the state of the call.
Optimizations

Some optimizations to increase performance are:

- Concurrent access to multiple servers
  - Use of threads in clients can help a client to make remote procedure calls concurrently to different servers
  - Early reply approach
    - An RPC Call is split into two RPC calls, one passing the parameter to the server and the other requesting result
    - In reply to the first call, the server sends a tag
    - In the second call the client sends the tag to match the result
    - A problem is server has to keep the result until client requests for it
  - Call buffering approach:
    - Clients and servers do not interact directly with each other but through a call buffer server.
    - Client sends its call request to call buffer server
    - Server polls the call buffer to check if there are outstanding calls
    - Client checks the call buffer for results

- Serving multiple requests simultaneously: Use of multiple threaded servers with dynamic thread creation facility
• Reducing per-call workload of servers: use stateless servers and let client maintain state information

• Reply caching of idempotent remote procedures.

• Proper selection of timeout values

• Proper design of RPC protocol specifications
Case Study: Sun RPC

Stub Generation

• Sun RPC uses the automatic stub generation approach

• Application’s interface definition is written in an IDL (Interface Definition Language), called RPC Language (RPCL).

• RPCL is an extension of Sun XDR language, originally designed for specifying external data representation
/* Interface definition for a stateless file service(StatelessFS) in the file named StatelessFS.x */

const FILE_NAME_SIZE=16
const BUFFER_SIZE=1024

typedef string FileName<FILE_NAME_SIZE>;
typedef long Position;
typedef long Nbytes;

struct Data{
    long n;
    char buffer[BUFFER_SIZE];
};

struct readargs{
    Filename filename;
    Position position;
    Nbytes n;
};

struct writeargs{
    Filename filename;
    Position position;
    Data data;
};

program STATELESS_FS_VERS{
    version STATELESS_FS_VERS{
        Data READ(readargs)=1;
        Nbytes WRITE(writeargs)=2;
    } = 1;
}=0x20000000;
The IDL compiler generates the following files from an interface definition:

- A header file that contains:
  - the constants and types defined in the interface definition file
  - external declarations for all XDR marshaling and unmarshaling procedures that are automatically generated

  The header file is manually included in the client and server programs and automatically included in the client stub, server stub and XDR filters files

- An XDR filters file that contains the XDR marshaling and unmarshaling procedures. These procedures are used by the client and server stub procedures. The name of this file is formed by taking the base name of the input file to rpcgen and adding _xdr.c

- A client stub file that contains one stub procedure for each stub procedure defined in the interface definition file.
  - A client stub procedure name is the name of the procedure given in the interface definition, converted to lower case and with an underscore and the version number appended. For example, READ procedure will be read_1. Name of the client stub file is formed by adding _clnt.c to the base name of the input file.
A server stub file that contains the *main* routine, the dispatch routine, and one stub procedure for each procedure defined in the interface definition.

- The *main* routine creates the transport handles and registers the service.

- *dispatch* routine dispatches the incoming remote procedure calls to the appropriate procedure. The name for the dispatch routine is formed by mapping the program name to lowercase and appending an underscore followed by version number.

- The name of the server stub file is formed by taking the base name of input file and appending _svc.c
Creation of an RPC application

- The application programmer manually writes the client program and the server program for the application
- The client program file is compiled to get the client object file
- Server program file is compiled to get the server object file
- Client stub file and the XDR filters file are compiled to get client stub object file
- Server stub file and the XDR filters file are compiled to get server stub object file
- Client object file, client stub object file, and client side runtime library are linked together to get the client's executable file
- Server object file, server stub object file, and client side runtime library are linked together to get the server's executable file
Steps in Creating RPC Applications in Sun

RPC

IDL Compiler, called "rpcgen"

Client Stub

Header File

C Compiler

Client Obj File

Linker

Client Exec. File

Server Stub

Server Obj. File

C Compiler

Server Stub Obj. File

Server-side RPC Runtime

Linker

Server Exec. File

C Compiler

Client Stub Obj. File

Client-side RPC Runtime Library
Some Features of Sun RPC

**Procedure arguments:** A remote procedure can accept only one argument and return only one argument

- So procedures requiring multiple parameters must include them as components of a single structure

- NULL pointer needs to be passed as parameter if the remote procedure does not take any parameter

- So, an RPC call always has two parameters, the first is a pointer to the single argument of the remote procedure and the second is a pointer to a client handle

- The returned result must be declared as a static variable in the server program.

**Marshaling Arguments and Results:** To overcome different data representation on different architectures, the data structures are converted
back and forth to eXternal Data Representation (XDR) using marshaling procedures.

- Marshaling/Unmarshaling takes place even between client and server of the same architecture, resulting in unnecessary overhead.

- To prevent this overhead users can write their own marshaling/unmarshaling procedures

**Call Semantics:** Supports at-least-once call semantics.

- After sending a request message, the RPCRuntime waits for a timeout period for reply and retransmits request

- Total time to wait and timeout period have default values of 25 and 5 seconds respectively. These values can be set to different values by users.
**Client-Server Binding:** Each node has a local binding agent called *portmapper*.

- When the server starts up, it registers its program number, version number and port number with the local portmapper.

- A client wanting to do an RPC must find the port number of the server

- Client makes a remote request for this (*clnt_create* returns a client handle that contains the necessary information for communicating with the corresponding server port.

- So, Sun RPC is not location transparent.

- The client handle is used by the client to directly contact the server when making subsequent RPCs to procedures of the service interface.
Limitations of Sun RPC

- Sun RPC lacks location transparency

- IDL of Sun RPC does not allow a general specification of procedure argument and results

- Sun RPC is not transport independent. Transport protocol is limited to TCP and UDP

- In UDP, Sun RPC messages are limited to 8KB in length

- Sun RPC does not have a network-wide client-server binding service

- Sun RPC does not include any integrated facility for threads even though it has implemented its own threads package.