Remote Procedure Call (RPC)

- RPC allows clients to call procedures in servers running on remote hosts
  - Parameters and results are packed in messages that are passed between the client and the server
- RPC offers access transparency
  - Clients call local procedures and remote procedures in the same way
- Message passing is invisible to the programmer
  - Message passing is hidden in two library procedures: the client stub and the server stub
- Example RPC systems: Sun RPC, DCE RPC, XML-RPC

RPC between a client and a server
The steps involved in a remote procedure call.

### RPC Steps

1. The client process calls the **client stub**, which resides within the client’s address space.
2. The client stub packs the parameters into a message. This is called **marshaling**. The client stub then executes a system call (e.g., `sendto`) to send the message.
3. The kernel sends the message to the remote server machine.
4. The server stub receives the message from the kernel.
5. The server stub **unmarshals** the parameters.
6. The server stub calls the desired procedure.
7. The server process executes the procedure and returns the result to the server stub.
8. The server stub **marshals** the results into a message and passes the message to the kernel.
9. The kernel sends the message to the client machine.
10. The client stub receives the message from the kernel.
11. The client stub **unmarshals** the results and passes them to the caller.
Marshaling and Unmarshaling

• Marshaling is the process taken by client stub/server stub to pack parameters/return values into a message
  – A message contains a sequence of bytes, which can be represented by an array of unsigned chars in C
• Marshalling includes converting the representation of the parameters into a standard format, and copying each parameter into the message
  – Computers differ in host byte order and in representation of various data types → Need convert parameters from the local format to a standard format
  – External Data Representation (XDR) (RFC 4506) is a standard data representation format used by many systems (e.g., NFS, ONC RPC)
• During unmarshaling, parameters in the message should be converted from the standard format to the local format

Dealing with Pointers

• A pointer is meaningful only within the address space of the process in which it is being used → messages passed between client and server must not contain pointers
• Then how do we pass a pointer?
  – If it points to a simple array or structure, pass a copy of the array or structure to the server
• How to pass data structures containing pointers (e.g., graphs)?
  – Pass the pointer to the server
  – Server stub sends a request for the referenced data to the client stub every time a pointer is encountered
RPC Protocol

- Client and server must follow the same RPC protocol, which specifies
  - The format of the messages exchanged
  - The representation of various data types (e.g., integers in two’s complement, characters in ASCII, floats in IEEE standard #754, with everything stored in big endian)
  - The transport protocol used for message exchange (e.g., TPC, UDP, HTTP)

```
foo_bar( char x; float y; int z[5] )
{
    ...
}
```

(a) A procedure. (b) The corresponding message

Stub Generation

- All RPC systems offer an Interface Definition Language (IDL) and an IDL compiler
- IDL is used to specify service interfaces
  - A service interface consists of a collection of procedures that are implemented by a server and can be called by a client
- Given a service interface specified in IDL, an IDL compiler (e.g., rpcgen in Sun RPC) can automatically generate the client stub and the server stub
Conventional and Asynchronous RPC

- Conventional RPC is **synchronous**: client blocks until a reply is returned.
- **Asynchronous RPC**
  - RPC returns as soon as the server acknowledges acceptance of the request message.
  - Useful when there is no need to wait for a reply (e.g., there is no result to return).

(a) Conventional RPC. (b) Asynchronous RPC.

Implementing RPC

- RPC is generally implemented over a **request-reply protocol** that supports two-way exchange of messages in client-server interactions.
- The protocol is based on 3 communication primitives:
  - **doOperation**: used by a client to invoke a remote operation
    - **doOperation** sends a request message to the remote server and returns the reply message.
  - **getRequest**: used by a server to acquire request messages.
  - **sendReply**: used by a server to send the reply message to the client after it has invoked the operation specified in the request message.
Request-Reply Communication

- The request/reply message contains the following fields:
  - **MessageType** (integer): 0=request, 1=reply
  - **RequestID** (integer): Generated by the client for each request message, copied by the server into the corresponding reply message
    - Allows client to match reply with request
  - **OperationID** (integer): identifies the procedure to be called
  - **Arguments** (an array of bytes): arguments/results of the remote procedure call
Timeouts

- **Timeout** is a method in which a process allows a fixed period of time for something to occur.
- Since a server might fail and a request or reply message might be dropped, **doOperation** uses a timeout when it is waiting to get the server’s reply message.
- Options after a timeout:
  - Returns immediately with an indication to the client that **doOperation** has failed.
  - Sends the request message repeatedly until either a reply is received or a predetermined number of attempts have been made.

Matching Reply with Request
Remote Method Invocation (RMI)

- RMI is essentially the same as RPC, except that it operates on distributed objects
  - E.g., Java RMI
- In RMI, a calling object can invoke a method in a remote object
  - A remote object is an object that exists in another process, which can be on the same machine or on a different machine
- All objects in an RMI-based system have unique object references
  - Object references can be passed as arguments and results of remote method invocations