

# ELEN-602: Computer Communications and Networks

Fall semester-2004

## Remote Procedure Calls (RPC)

Most client-server applications have a request reply semantics i.e., client makes a request and the server sends a reply to this request. TCP style connection establishment and termination may incur unnecessary overhead for such transaction-oriented applications where a single message may be exchanged each time.

RPC functions similar to a procedure call. In a normal procedure call, the caller passes a number of arguments to the called procedure; the called procedure may return data as a result of execution of the called procedure. The local procedure call passes program control to the called procedure and returns to the calling procedure after the completion of the called procedure. RPC functions very similarly across a network. A program in the client machine employs RPC to invoke a service in the server machine. The server machine returns a reply as a result of this call. This is shown in Figure 1 below. We will discuss the relevant concepts of RPC in subsequent sections.

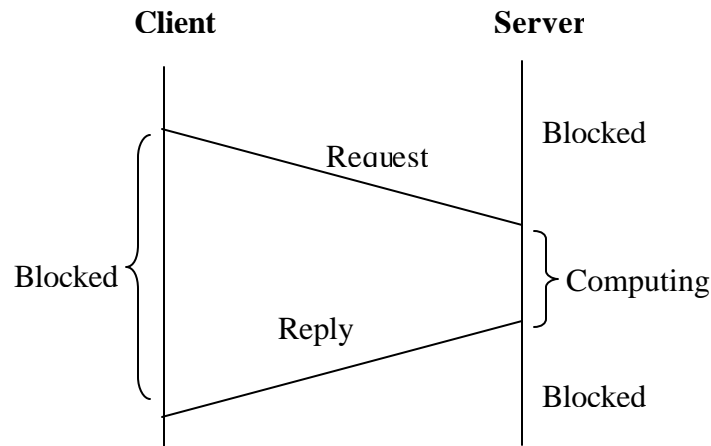
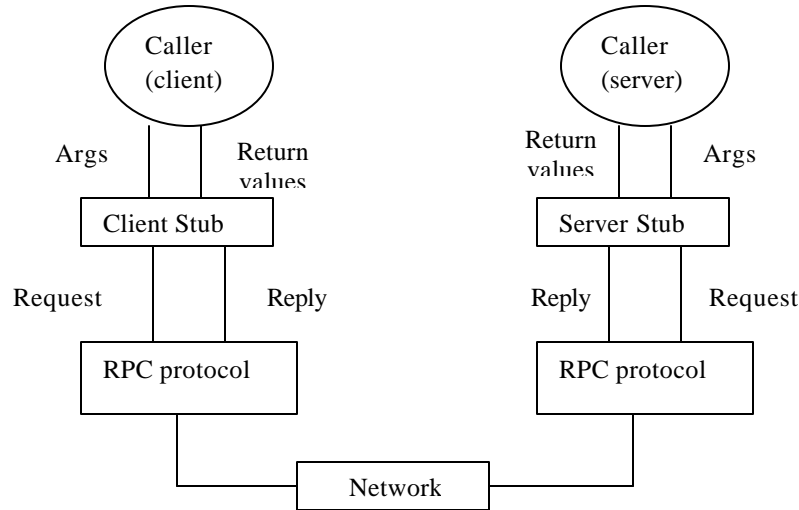


Figure 1: Timeline for RPC

RPC consists of two main components as shown in Figure 2. The *stub compilers* allow data to be formatted into an RPC-specific format to enable transport arguments or replies across heterogeneous machines on a network. The client calls a local stub for the procedure, passing it the arguments required for the procedure. The client's stub packages the arguments into a message and sends the message to the remote server. This fact that the procedure is being invoked remotely is hidden from the client. The server's stub receives the client's request, translates the arguments such that the server's application can understand and passes the arguments to the server's application. After the server application completes, the reply is returned to the server's stub, which packages the reply into a message and sends it across the network to the client. The client's stub translates the reply and returns it as a value that is returned as a result of the client's procedure call. The process of packaging the arguments passed by the client for transmission on the

network is called Marshaling. Un-marshaling is done by the server stub process when it receives the client's request.

We will focus on SunRPC as an example RPC protocol. SunRPC is Sun Microsystems' popular implementation of RPC. SunRPC uses XDR (eXternal Data Representation) to describe arbitrary data structures in a machine-independent fashion. The RPC stub



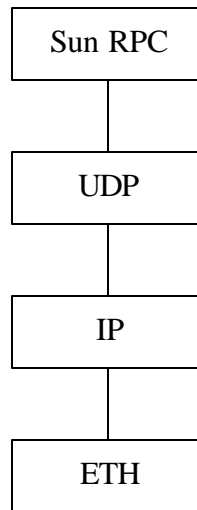
**Figure 2: The basic RPC model**

compilers convert data into and out of XDR representation on the client and the server machines. Data is converted into XDR representation before it is transmitted over the network and at the receiving end, is translated into the data format used there. By using the XDR standard data representation convention, systems do not have to understand and translate each and every data format that may exist on the network; there is only the one XDR convention. Using XDR, data can be exchanged among machines of various hardware and software architectures without worrying about word lengths, byte ordering, and floating-point representations. DCE (Distributed Computing Environment) –RPC employs NDR (Network Data Representation). We will briefly discuss the XDR standard in a later module.

The RPC protocol must consider the following three issues: (a) fragmenting large messages, (b) synchronizing request and reply messages and (c) dispatching request messages to the appropriate procedure. SunRPC employs IP at the network layer. IP takes care of fragmentation and reassembly of messages into suitable sized packets based on network MTU (we learnt about IP's fragmentation/reassembly in module 4).

The protocol graph for SunRPC is shown in Figure 3. SunRPC is layered above UDP. SunRPC uses two-tier addresses to identify the remote procedure, through a 32-bit program number and a 32-bit procedure number. UDP port numbers are used to reach the correct program by assigning individual ports to individual programs. SunRPC directs the message to the appropriate procedure within the program. In order to send a message to a remote procedure, the UDP port number of the program and the procedure number of the

required procedure (within that program) need to be identified. Port Mapper program running at the well-known UDP port number of 111 is used to identify the port number of



**Figure 3: Protocol Graph for Sun-RPC [1]**

an individual program. Then to send a message to procedure  $x$  of that program, an RPC message with procedure number  $x$  is sent to that UDP port corresponding to the program.

As described above IP handles issue (a) and the two-tier addressing scheme handles issue (c) in SunRPC. The remaining issue we need to discuss is how to synchronize the request and reply messages. To understand issue (b), let us first see the problems associated with the request-reply semantics of an RPC call. The following definitions are in order.

- **Idempotent operations:** An operation is said to be idempotent if repeated executions of the operation does not do us harm. For example, if the execution of the operation returns to us the current date, repeated executions of this procedure would not do any harm. The client can wait for one reply from the server, and just ignore further replies.
- **Non-Idempotent operations:** An operation is said to be non-idempotent if repeated executions of the operation would produce undesirable results. For example, if a single execution of the call transfers \$100 from account A to account B, we might not want to have multiple (redundant) calls to this procedure! Another example can be that of appending some bytes at the end of a file. Repeated executions of this procedure would be undesirable too.

Further, an RPC request, as explained in earlier sections, is packaged and transmitted by the client stub routine over the network. The network being IP based does not provide any guarantees on reliable and in-order delivery, thereby potentially causing loss/duplication of request packets. In case of a lost request, the client has to somehow determine this loss (say, by a timeout mechanism), and retransmit the request. Now, consider the following scenario:

A client transmits its request over the network, and the packet does not reach the server, thereby making the client to timeout and retransmit the same request again. Now, if the earlier request was delayed somewhere on the network, and if it eventually reaches the server, we can have the case where both the original and retransmitted requests reach the server and result in two executions of the call. This may not be desirable in cases where the procedure is non-idempotent.

The above scenario necessitates the implementation of what we call *at-most-once* semantics. This means that for every copy of the request that the client sends, at most one copy is delivered (and subsequently served) to the server. We explain how Sun RPC preserves *at-most-once* semantics below:

Sun RPC clients (and servers) maintain *transaction-Ids* or XIDs to identify client requests and server replies. When the RPC client issues a request, it associates each request with a unique 32-bit XID. The XID is packaged and sent along with the RPC message. The server returns this XID in its reply. Retransmission of a request by a client preserves the original XID. Now, based on this simple mechanism, the client just compares the XID of the reply with the XID of the request and determines whether it is a “correct” reply. The client ignores replies in case of an XID mismatch. The server maintains a cache of the replies that it sends and checks for duplicate requests among the cache entries.

#### **Brief discussion on the XDR standard:**

To build an RPC application, a specification file is written using XDR conventions. This specification (generally given a “.x” extension), contains commands and constructs to specify the various functions (procedures) to be called remotely, with the arguments they take in and the return values they yield. An example XDR file could be as below:

```
program TESTPROG {
    version TESTVER {
        int SQUAREOF (int) = 1;
    } = 1;
} = 0x34567889;
```

The above XDR specification file contains the following major elements:

- Program number – the Program number is the 32 bit hexadecimal number specified at the end of the outermost loop (0x34567889) . Also remember that only a specific range of program numbers can be assigned by the user. Sun RPC specifies that range as 0x20000000 – 0x3fffffff.
- Version number – Each program can have different versions. The version number in the above case is 1
- Procedure number – The procedure SQUAREOF is given the procedure number 1. Procedure numbers start from 0, but 0 is a special NULL procedure which is system defined. The user defined procedure numbers start from 1 onwards.

These numbers are used to uniquely identify a server (and the service it thereby provides).

XDR uses implicit typing, meaning that both the ends (client and server) must exactly know the data types and ordering of data.

The XDR format has its own conventions and rules. For example, in the XDR convention, a null terminated string, generally represented by saying `'char*'` in C, would be represented by saying `'string'`. There are other data types defined by XDR, and a detailed discussion about that can be found in Sun's documentation on `ONC+ Developer's guides`.

The `rpcgen` compiler takes in a `“.x”` file, and produces the client and server stubs and header files. It can also generate program templates for the client and server.