Remote Procedure Call Concept (RPC)

Prof. Chuan-Ming Liu  
Computer Science and Information Engineering  
National Taipei University of Technology  
Taipei, TAIWAN
Introduction

continues the discussion of middleware (i.e., tools and libraries programmers used to construct client-server software)

introduces the RPC concept, and describes a particular implementation of an RPC that uses XDR standard for data representation

shows how the approach simplifies the design of client-server software and makes the resulting program easier to understand
**RPC Model**

When programmers build a client-server application, they must consider how the entire system will function and how the two components will interact.

To help programmers design and understand client-server interaction, researchers have devised a conceptual framework (*RPC model*) for building distributed programs.

The RPC model uses familiar concepts from conventional programs as the basis for the design of distributed applications.
Two Paradigms for Building Distributed Programs

- Communication-oriented Design
- Application-oriented Design
Communication-oriented Design

- Begin with the communication protocol
- Design a message format and syntax
- Design the client and server components by specifying how each reacts to incoming messages and how each generates outgoing messages
Application-oriented Design

Begin with applications

Design a conventional application program to solve the problem

Build and test a working version of the conventional program that operates on a single machine

Divide the program into two or more pieces, and add communication protocols that allows each piece to execute on a separate computer
Problems on Communication-oriented Design

By focusing on the communication protocol, programmer may miss important subtleties in the application. It usually becomes the centerpiece of the resulting programs, making application software difficult to understand or modify. Because few programmers have experience and expertise with the protocol design, they often produce awkward, incorrect, inefficient protocols.
Building Distributed Programs

- The RPC model follows the application-oriented design approach, which emphasizes the problem to be solved instead of the communication needed.

- The programmer can follow good design principles that make the code modular and maintainable.

- RPC separates the solution of a problem from the task of making the solution operate in a distributed environment.
Building Distributed Programs

The RPC paradigm for programming focuses on the application.

It allows a programmer to concentrate on devising a conventional program that solves the problem before attempting to divide the program into pieces that operate on multiple computers.
Conventional Procedure Calls

Fig. 21.1 illustrates the procedure concept. Procedures offer a powerful abstraction that allows programmers to divide programs into small, manageable, easily-understood pieces.
Figure 21.1 The procedure concept. A conventional program consists of one or more procedures, usually arranged in a hierarchy of calls. In the diagram, an arrow from procedure $n$ to procedure $m$ denotes a call from $n$ to $m$. 
An Extension of the Procedural Model

Fig. 21.2 show how the program from Fig. 21.1 can be extended to use an RPC to become a distributed program.

Before a program can use RPCs, it must be augmented with protocol software that allows it to communicate with remote procedures.
Figure 21.2 A distributed program that shows how the program from Figure 21.1 can be extended to use a remote procedure call. The division occurs between the main program and procedure 4. A communication protocol is needed to implement the remote call.
Execution of Conventional Procedure Calls

Fig. 21.3 shows a conceptual model of execution that explains flow of control during procedure call and return.

A single thread of control (execution) begins in the main program, passes through procedure A and B, and eventually returns to the main program.
Figure 21.3 A conceptual model of execution that explains flow of control during procedure call and return. A single thread of control begins in the main program, passes through procedures A and B, and eventually returns to the main program.
The procedure Model in Distributed Systems

Fig. 21.4 illustrates the model of execution used with RPCs.

A request sent from a client to a server corresponds to a *call* of a remote procedure, and a response sent from a server back to a client corresponds to the execution of a *return* instruction.
Figure 21.4 The model of execution used with remote procedure calls. A single thread of control executes in a distributed environment. Dashed lines show how control passes from a client to a server during a remote procedure call, and how it passes back when the server responds.
Analogy Between Client-Server and RPC

- RPC transfers control to the called procedure like a conventional procedure call.
- The system suspends execution of the calling procedure during the call and only allows the called procedure to execute.
Analogy Between Client-Server and RPC

As Fig. 21.4 illustrates, nested RPCs corresponds to a server that becomes a client of another service.

Conventional procedures usually accept a few arguments and return a few results.

However, a server can accept or return arbitrary amount of data (i.e., It can accept or return an arbitrary stream over a TCP connection).
Practical Constraints

Network delays can make an RPC several orders of magnitude more expensive than a conventional procedure call.

An RPC cannot have pointers arguments because the remote procedure operates in a completely different address space than the caller.

Because a remote procedure does not share the caller’s environment, it does not have direct access to the caller’s I/O descriptors or OS functions.
Thinking of a distributed computation as a single program in which control passes across the network to a remote procedure and back helps programmers specify client-server interaction.

It relates the interaction of distributed computations to the familiar notions of procedure call and return.
Sun Microsystem’s RPC

Definition

Sun has defined a special form of RPC: Sun RPC, Open Network Computing (ONC) RPC, or simply RPC.

It has been used as an implementation mechanism for many applications, including the NFS.

ONC RPC defines:
- format of messages that the caller (client) sends to invoke a remote procedure on a server,
- format of arguments, and
- format of results that the called procedure returns to the caller.
Sun Microsystem’s RPC Definition (cont.)

- It permits the calling program to use either UDP or TCP to carry messages and uses XDR to represent procedure arguments as well as other items in an RPC message header.

- In addition to the protocol specification, ONC RPC includes a compiler system that helps programmers build distributed programs automatically.
Remote Programs and Procedures

ONC RPC extends the RPC model by defining a remote execution environment.

It defines a *remote program* as the basic unit of software that executes on a remote machine.

As Fig. 21.5 illustrates, all remote procedures inside the remote program can share access to the single database.
Remote Programs and Procedures (cont.)

For example, one can implement a single remote database by constructing a single remote program that includes data structures to hold shared information and three remote procedures to manipulate it: *insert*, *delete*, and *lookup*. 
Figure 21.5 Conceptual organization of three remote procedures in a single remote program. All three procedures share access to global data in the program, just as conventional procedures share access to global data in a conventional program.
Reducing the Number of Arguments

Using a structure instead of multiple arguments make the program more readable because the structure field names serve as keywords that tell the readers how each argument will be used.

If all programs using RPC collect their arguments into a structure, each remote procedure will need only a single argument.
Identifying Remote Programs and Procedures

A specific remote procedure on a given remote program can be identified by a pair: 

\[(prog, proc)\]

- \(prog\): identifies the remote program, which is a unique 32-bit integer

- \(proc\): identifies a remote procedure within the remote program, which is an integer between 1 to \(N\); 0 is reserved for an echo procedure for testing reachability
Identifying Remote Programs and Procedures

To avoid conflicting, RPC has divided the set of programs into 8 groups as Fig. 21.6 shows.

Of the $2^{29}$ program numbers available in the first group, Sun has only assigned a handful of numbers as shown in Fig. 21.7 (some of the assignments)
### Figure 21.6

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Values Assigned By</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>0x1fffffff</td>
<td>Sun Microsystems, Inc.</td>
</tr>
<tr>
<td>0x20000000</td>
<td>0x3fffffff</td>
<td>The system manager at a user’s site</td>
</tr>
<tr>
<td>0x40000000</td>
<td>0x5fffffff</td>
<td>Transient (temporary)</td>
</tr>
<tr>
<td>0x60000000</td>
<td>0x7fffffff</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x80000000</td>
<td>0x9fffffff</td>
<td>Reserved</td>
</tr>
<tr>
<td>0xa0000000</td>
<td>0xbfffffff</td>
<td>Reserved</td>
</tr>
<tr>
<td>0xc0000000</td>
<td>0xdfffffff</td>
<td>Reserved</td>
</tr>
<tr>
<td>0xe0000000</td>
<td>0xffffffff</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The division into eight groups of 32-bit numbers that ONC RPC uses to identify remote programs. Each remote program is assigned a unique number.
<table>
<thead>
<tr>
<th>Name</th>
<th>assigned number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>portmap</td>
<td>100000</td>
<td>port mapper</td>
</tr>
<tr>
<td>rstatd</td>
<td>100001</td>
<td>rstat, rup, and perfmeter</td>
</tr>
<tr>
<td>rusersd</td>
<td>100002</td>
<td>remote users</td>
</tr>
<tr>
<td>nfs</td>
<td>100003</td>
<td>network file system</td>
</tr>
<tr>
<td>ypserv</td>
<td>100004</td>
<td>yp (now called NIS)</td>
</tr>
<tr>
<td>mountd</td>
<td>100005</td>
<td>mount, showmount</td>
</tr>
<tr>
<td>dbxid</td>
<td>100006</td>
<td>DBXprog (debugger)</td>
</tr>
<tr>
<td>ypbind</td>
<td>100007</td>
<td>NIS binder</td>
</tr>
<tr>
<td>wallid</td>
<td>100008</td>
<td>rwall, shutdown</td>
</tr>
<tr>
<td>yppasswdd</td>
<td>100009</td>
<td>yppasswd</td>
</tr>
<tr>
<td>etherstatd</td>
<td>100010</td>
<td>ethernet statistics</td>
</tr>
<tr>
<td>rquotad</td>
<td>100011</td>
<td>rquotaprog, quota, rquota</td>
</tr>
<tr>
<td>sprayd</td>
<td>100012</td>
<td>spray</td>
</tr>
<tr>
<td>selection_svc</td>
<td>100015</td>
<td>selection service</td>
</tr>
<tr>
<td>dbsessionmgr</td>
<td>100016</td>
<td>unify, netdbms, dbms</td>
</tr>
<tr>
<td>rexd</td>
<td>100017</td>
<td>rex, remote exec</td>
</tr>
<tr>
<td>office_auto</td>
<td>100018</td>
<td>alice</td>
</tr>
<tr>
<td>lockd</td>
<td>100020</td>
<td>klmprog</td>
</tr>
<tr>
<td>lockd</td>
<td>100021</td>
<td>nimprog</td>
</tr>
<tr>
<td>statd</td>
<td>100024</td>
<td>status monitor</td>
</tr>
<tr>
<td>bootparamd</td>
<td>100026</td>
<td>bootstrap</td>
</tr>
<tr>
<td>pcnfsd</td>
<td>150001</td>
<td>NFS for PC</td>
</tr>
</tbody>
</table>

*Figure 21.7* Example RPC program numbers currently assigned by Sun Microsystems, Inc.
Accommodating Multiple Versions of a Remote Program

ONC includes an integer version number for each remote program

\[(prog, vers, proc)\]

`vers` specifies the version of the program to which the message has been sent

Because all ONC RPC messages identify a remote program, the version of that program, and a remote procedure in the program, it is possible to migrate from one version of a remote procedure to another gracefully and to test a new version of the server while an old version continues to operate
Mutual Exclusion

RPC provides automatic mutual exclusion among procedures within a given remote program by permitting at most one remote procedure (e.g., either insert or delete) to execute at a given time
Communication Semantics

When choosing the semantics for ONC RPC, the designers have to choose between two possibilities:

- To make a remote procedure call behave as much like a local procedure, RPC should use a reliable transport like TCP and should guarantee reliability to the programmer.
- To allow programmers to use efficient, connectionless transport protocols, the RPC should support communication through a datagram protocol like UDP.
ONC RPC allows each application to choose TCP or UDP as a transport protocol. It does not enforce reliable semantics.
At Least Once Semantics

The ONC RPC standard uses the *at least once semantics* to describe RPC execution when the caller receives a reply.

It uses *zero or more semantics* to describe the behavior of a remote procedure call when the caller does not receive a reply.

Programmers who choose to use UDP as the transport protocol for an ONC application must build the application to *tolerate zero-or-more execution semantics*. 
At Least Once Semantics (cont.)

Zero-or-more semantics usually means that a programmer makes each RPC *idempotent* (An operation is said to be idempotent if repeated applications of the operation produce the same result).

For example, a remote procedure that appends data to a file is not idempotent, however, a remote procedure that write data to a specified position in a file is idempotent.
RPC Retransmission

The library software supplied with the ONC RPC implementation includes a fixed (nonadaptive) timeout and retransmission strategy.

Programmers can adjust the timeout and retry limits for a given application.

An application cannot interpret failure as a guarantee that remote procedure was never executed (in fact, it may have executed several times).
Mapping a Remote Program to a Protocol Port

UDP and TCP transport protocols use 16-bit protocol numbers to identify communication endpoints.

To make it possible for clients and servers to rendezvous, we assume that each service is assigned a unique protocol port number and that the assignments are well-known.

ONC RPC introduces an interesting problem: because it uses 32-bit numbers to identify remote programs, RPC programs can outnumber protocol ports.
Mapping a Remote Program to a Protocol Port (cont.)

- If an RPC program does not use a reserved, well-known protocol port, clients cannot contact it directly.

- Because the RPC program (server) only obtains a protocol port after it begins execution, the client cannot know which protocol port the server obtained.
Dynamic Port Mapping

To allow clients to contact remote programs, the ONC RPC mechanism includes a dynamic mapping service.

Fig. 21.8 illustrates that the ONC PRC port mapper operates as a separate server process.
Dynamic Port Mapping (cont.)

Each RPC program registers its program number, protocol port number, and version number with the *port mapper* on the local machine.

A caller contacts the *port mapper* on a machine to find the protocol port to use for a given RPC program on that machine.
Figure 21.8 The ONC RPC port mapper. Each RPC program registers its program number, protocol port number, and version number with the port mapper on the local machine. A caller contacts the port mapper on a machine to find the protocol port to use for a given RPC program on that machine.
Port Mapper Algorithm

Algorithm 21.1 shows the ONC RPC port mapper algorithm.

The port mapper allows clients to reach remote programs even though the remote programs dynamically allocate protocol ports.

Registration requests: The remote program contacts the port mapper on its local machine and adds a triple of integers to the database: (RPC prog no., protocol port no., version no.)
Look-up requests: Callers on other machines (by knowing the address of the machine on which the remote program executes) specify a remote program number and version number, and request the protocol port number that can be used to reach the remote program.

Caller can always reach the port mapper because the port mapper communicates using the well-known protocol port, 111.
Algorithm 21.1

1. Create a passive socket bound to the well-known port assigned to the ONC RPC port mapper service (111).

2. Repeatedly accept requests to register an RPC program number or to look up a protocol port for a specified RPC program number.

   Registration requests come from RPC programs on the same machine as the port mapper. Each registration request specifies a triple consisting of the RPC program number, the protocol port currently used to reach that program, and a version number. When a registration request arrives, the port mapper adds the triple to its database of mappings.

   Look up requests come from arbitrary machines. They each specify a remote program number and version number, and request the number of the protocol port that can be used to reach the remote program. The port mapper looks up the remote program in its database, and responds by returning the corresponding protocol port for that program.

Algorithm 21.1 The ONC RPC port mapper algorithm. One port mapper server runs on each machine that implements the server side of an RPC program.
ONC RPC Message Format

A message type field in the RPC message header distinguishes between messages that a client uses to initiate an RPC and messages that an RPC uses to reply.

Constants used in the message type field can be defined using XDR language, for example, the declaration, as shown in Fig. 21.8a.

For example, once values have been declared for symbolic constants, the XDR language can define the format of an RPC message, as shown in Fig. 21.8b.
Figure 21.8a

```
enum msg_type {
    CALL = 0;
    REPLY = 1;
}; /* RPC message type constants */
```
Figure 21.8b

```c
struct rpc_msg { /* Format of an RPC message */
    unsigned int mesgid; /* used to match reply to call */
    union switch (msg_type mesgt) {
        case CALL:
            call_body cbody;
            case REPLY:
                rply_body rbody;
        } body;
    };
```
ONC RPC Message Format (cont.)

The declarations for `call_body` and `rply_body` must be given elsewhere.

For example, RPC defines a `call_body` to have the form, as shown in Fig. 21.8c.
struct call_body {
    /* format of RPC CALL */
    unsigned int rpcvers;  /* which version of RPC? */
    unsigned int rprog;    /* number of remote program */
    unsigned int rprogvers; /* version number of remote prog*/
    unsigned int rproc;    /* number of remote procedure */
    opaque_auth cred;      /* credentials for called auth. */
    opaque_auth verf;      /* authentication verifier */
    /* ARGS */
    /* arguments for remote proc. */
};
Marshaling Arguments for a Remote Procedure

RPC must represent all arguments in an external form that allows them to be transferred between computers.

If any of the arguments passed to the remote procedure consists of a complex data structure like linked list, it must be encoded into a compact representation that can be sent across the network.

The terms *marshal*, *linearize*, or *serialize* to denote the tasking of encoding arguments.
Marshaling Arguments for a Remote Procedure (cont.)

The client side of RPC marshals arguments into the message and the server side unmarshals them.

Marshaling and unmarshaling large data structures can require significant CPU time and network bandwidth.

Thus, most programmers avoid passing linked structure as arguments.
Authentication

RPC defines several possible forms of authentication, including a simple scheme that relies on functions available in the OS and a more complex scheme that uses the DES (Data Encryption Standard).

Authentication information can have one of the 4 types shown in Fig. 21.8d.
enum auth_type {
    AUTH_NULL = 0;
    AUTH_UNIX  = 1;
    AUTH_SHORT = 2;
    AUTH_DES  = 3;
};

/* possible forms of auth. */
/* no authentication */
/* Machine name authentic. */
/* Used for short form auth. in */
/* messages after the first */
/* NIST's (NBS's) DES standard */
Authentication (cont.)

The declaration of the authentication structure in an RPC message uses the keyword *opaque* to indicate that it appears in the message without any interpretation, as shown in Fig. 21.8e.

UNIX authentication defines the structure of the authentication information to contain several fields, as shown in Fig. 21.8f.
struct opaque_auth {
    auth_type atype;
    opaque body<400>;
};

/* structure for authent. info. */
/* which type of authentication */
/* data for the type specified */
Figure 21.8f

```c
struct auth_unix {
    unsigned int timestamp;  /* integer timestamp */
    string smachine<255>;   /* name of sender's machine */
    unsigned int userid;     /* user id of user making req. */
    unsigned int grpids<10>; /* group id of user making req. */
    unsigned int grpids<10>; /* other group ids for the user */
};
```
An Example of RPC Message Representation

Fig. 21.9 illustrates an RPC CALL message. The size of each field is determined by its RPC definition and the XDR specification of sizes.
Figure 21.9 An example of the external format used for an RPC CALL message. The first fields of the message have a fixed size, but the sizes of later fields vary with their content.
An Example of the UNIX Authentication Field

The size of the authentication field in an RPC message depends on its contents.

Fig. 21.10 illustrates the representation for a UNIX authentication field.
### Figure 21.10

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTHENTICATION TYPE (↑ FOR UNIX)</td>
<td></td>
</tr>
<tr>
<td>LENGTH OF BODY THAT FOLLOWS (48)</td>
<td></td>
</tr>
<tr>
<td>TIMESTAMP (e.g., 0x2b15025C)</td>
<td></td>
</tr>
<tr>
<td>LENGTH OF MACHINE NAME THAT FOLLOWS (20)</td>
<td></td>
</tr>
<tr>
<td>MACHINE NAME (merlin.cs.purdue.edu)</td>
<td></td>
</tr>
<tr>
<td>USERID OF SENDER (30)</td>
<td></td>
</tr>
<tr>
<td>GROUP ID OF SENDER (30)</td>
<td></td>
</tr>
<tr>
<td>LENGTH OF GROUP ID LIST THAT FOLLOWS (2)</td>
<td></td>
</tr>
<tr>
<td>GROUP ID₁ (30)</td>
<td></td>
</tr>
<tr>
<td>GROUP ID₂ (59)</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 21.10** Example representation for UNIX authentication within an ONC RPC message. The example values are taken from a message sent by a user with numeric login identifier 30 on machine merlin.cs.purdue.edu.
Summary

- Using the RPC model helps programmers focus on the application instead of the communication protocol.
- ONC RPC specifies a scheme for identifying remote procedures as well as a standard for the format of RPC messages.
- XDR is used to keep message representations machine independent.
Summary (cont.)

ONC RPC programs do not use well-known protocol ports like conventional clients and servers.

They use a dynamic binding mechanism that allows each RPC program to choose an arbitrary, unused protocol port when it begins.

Called RPC port mapper, the binding mechanism requires each computer that offers RPC programs registers with the port mapper on its local machine after it obtains a protocol port.
Summary (cont.)

When an RPC client wants to contact an RPC program, it first contacts the port mapper on the target machine.

The port mapper responds by telling the client which port the target RPC program is using.

The client contacts the RPC program directly using that port.