Project 1. Using datagram sockets to implement a simple RPC framework

Introduction

This project contains three exercises that lead you through the steps necessary to master the use of Linux datagram sockets to build a form of remote procedure call. Your programs should be written in C.

You programs must compile and run on pyrite. You need to use an SSH client to connect to pyrite. This page talks about how to use SSH for Windows and OS X users.

You may borrow any code you feel necessary from the example program UDPsock.c, such as *MakeLocalSA*, *MakeDestSA*, *MakeReceiverSA*, *printSA*, *anyThingThere* and use them as utilities in the following exercises.

Exercise 1: A server that echoes client input

You are required to produce client and server programs based on the procedures *DoOperation*, *GetRequest* and *SendReply*. The prototypes of these procedures are given in the Appendix. These operations have been simplified so that client and server exchange messages consisting of strings. In Exercise 2, you will put the arguments of *DoOperation* into a request message.

The client and server behave as follows:

Client: this takes the name of the server computer as an argument. It repeatedly requests a string to be entered by the user, and uses *DoOperation* to send the string to the server, awaiting a reply. Each reply should be printed on the screen. The client exits when the string entered is "quit".

Server: repeatedly receives a string using *GetRequest*, prints it on the screen, and echoes the string back with *SendReply*.

You will implement *DoOperation*, *GetRequest* and *SendReply*. The prototypes are given in the appendix. The *Status* value returned reflects the values returned by *UDPsend* and *UDPreceive* (see below).

DoOperation	Send a given request message to a given socket address and blocks until it returns with a reply message
GetRequest	Receive a request message and the client's socket address
SendReply	Send a reply message to the given client's socket address

UDPsend and *UDPreceive*

The procedures *DoOperation*, *GetRequest* and *SendReply* must use two procedures *UDPsend* and *UDPreceive* to be written by you, which respectively send and receive a message over/from a socket. You are to implement these functions using the socket calls *sendto* and *recvfrom*.

Each procedure returns a value of type *Status* which reports on the status of its execution. For example, if the *sendto* or *recvfrom* system calls return negative values, your procedures should return a *Status* value of *Bad*.

You should use the definitions for *SocketAddress* and *Message* and the prototypes for *UDPsend* and *UDPreceive* given in the appendix.

UDPsend	Send a given message through a socket to a given socket address
UDPreceive	Receive a message and the socket address of sender into two arguments

Choosing a server port

Your server may use a port number in the range 10000 to 10100. You will want to run server processes that can coexist with other students' server processes on pyrite. Two server programs running on the same computer cannot use the same local port number. You will therefore want to choose a port number that is different from other students' port numbers. If everybody takes the first open port number (i.e., port 10000) and adds the last two digits of their University ID, there should be no clashes. That is:

```
aPort = 10000 + last 2 digits of your University ID;
```

Exercise 2: An arithmetic server using RPC

In this exercise you will create an adaptation of Exercise 1, so that the strings that users type to the clients are arithmetic expressions; and the server evaluates each arithmetic expression and returns the results. Communication is to be done by RPC implemented by you. Client and server should provide the ability to use *Stop* and *Ping* in addition to doing arithmetic.

Stop	The server returns Ok to the client and then exits
Ping	The server returns Ok to the client and then continues

Client: Each string typed at the client is to be interpreted as a simple arithmetic operation (e.g. 34+67, 89*54, etc). This requires the expression to be separated into an operation (+, -, *, /) and two non-negative integer arguments. The client exits when the string entered by user is “quit”.

Server: It must implement an "Arithmetic Service" consisting of four operations add, subtract, multiply and divide which should have identical prototypes:

```
Status op( int x, int y, int *result) //the last argument is for the result
```

The *Status* value should be extended with extra values required for this service, e.g., *DivZero* for dividing by 0.

You will need to define a struct in C for RPC messages as suggested in the Appendix.

This exercise requires you to write marshalling and unmarshalling functions. The marshalling function takes an RPC message, copies the fields of the RPC message into a network message after converting the integers in the RPC message to network byte order. The unmarshalling function takes a network message and reconstructs the RPC message from it after converting the integers in the network message back to host order. You should use the functions *htonl* and *ntohl* to do the conversions. The recommended prototypes for marshal and unmarshal are given in the Appendix.

Two other matters should be addressed:

1. Both client and server should make use of the *messageType* field of the *RPCmessage* structure after unmarshalling to test that Requests and Replies are not confused;
2. The client should generate a new *RPCId* for each call; and the server should copy the *RPCId* from the *Request* message to the *Reply* message and the client should test that the replies correspond to the requests.

Exercise 3 Adding a timeout

Add a timeout in the procedure *DoOperation*. This should have the effect that if there is no response from the server for several seconds after sending the request message, the client resends the request for up to 3 times. This is in case a message was dropped or the server has crashed. The client should report on its behavior in these circumstances. Extend *Status* to allow for the timeout.

Timeout can be done by using the *select* system call to test whether there is any outstanding input on the socket before calling *UDPreceive*. The procedure *anyThingThere* in the example program UDPsock.c shows how to do this. You can test your timeout by running the client when the server is not running.

Submitting Your Project

You will submit your source files for the three exercises. Your programs must compile and run without errors on pyrite.cs.iastate.edu. The client and server programs should have separate code (unlike the example program UDPsock.c).

For Exercise 1, name your client and server `EchoClient.c` and `EchoServer.c`, respectively. For Exercise 2 and 3, name your client and server `RPCClient.c` and `RPCServer.c`, respectively.

Put all your source files in a folder, then use command

```
zip -r <your ISU Net-ID> <src_folder>
```

to create a .zip file. For example, if your Net-ID is ksmith and project1 is the name of the folder that contains all your source files, then you will type

```
zip -r ksmith  project1
```

to create a file named ksmith.zip. You should submit your .zip file on Canvas.

Grading

Total 140 points

- Exercise 1: 40 points
- Exercise 2: 80 points
- Exercise 3: 10 points
- Documentation: 10 points (You should include plenty of comments in your source code.)

APPENDIX: Definitions for C Programs

You are to use the following type definitions:

```
#define SIZE 1000
typedef struct {
unsigned int length;
unsigned char data[SIZE];
} Message;
typedef enum {
       OK,                    /* operation successful */
       BAD,                   /* unrecoverable error */
} Status;
typedef struct sockaddr_in SocketAddress ;
```

You may alternatively define *data* as a pointer.

The prototypes for *DoOperation*, *GetRequest* and *SendReply* are as follows:

```
Status DoOperation (Message *message, Message *reply, int
socket, SocketAddress serverSA);

Status GetRequest (Message *callMessage, int socket,
SocketAddress *clientSA);

Status SendReply (Message *replyMessage, int socket,
SocketAddress clientSA);
```

The prototypes for *UDPsend* and *UDPreceive* are as follows:

```
Status UDPsend(int socket, Message *m, SocketAddress
destination);

Status UDPreceive(int socket, Message *m, SocketAddress
*origin);
```

C definitions for Exercise 2

You should use the following definition of an RPC message:

```
typedef struct {

      enum {Request, Reply} messageType;        /* same size as an unsigned int */
      unsigned int RPCId;                       /* unique identifier */
      unsigned int procedureId;                 /* e.g.(1,2,3,4) for (+, -, *, /) */
      int arg1;                                 /* argument/return parameter */
      int arg2;                                 /* argument/return parameter */
                                                /* each int (and unsigned int)is 4
                                                bytes */
} RPCMessage;
```

The fields *arg1* and *arg2* can be used for arguments in request message or returned value and status in reply message.

The prototypes of the marshalling and unmarshalling procedures should be as follows:

```
void marshal(RPCmessage *rm, Message *message);
/* convert an RPC message to a network message. */

void unMarshal(RPCmessage *rm, Message *message);
/* reconstruct the RPC message from a network message. */
```