BSE 3202 – Distributed Systems Development

Project 2: Using sockets to implement a simple RPC framework in C or C++

Introduction

This project consolidates the basic knowledge about RPC implementation in Chapters 4 and 5 and gives students experience in constructing programs that use Unix sockets with UDP. This exercise should be done in groups of four or six. You are required to demonstrate working programs and to submit well-commented code.

Outline

This document contains a set of exercises that are intended to lead you through the steps necessary to master the use of UNIX datagram sockets to build a form of remote procedure call. The attached notes on Sockets in Unix may be also be useful (socketnotes.pdf). You should use either C++ or ANSI standard C function prototypes. Definitions for use with for C++ and definitions for use with C are given as appendices at the end.

Preamble

Before working on the coursework itself, you should perform the following exercise, which is not examined. It involves compiling and running an existing sockets program in *UDPsock.c* . Study it carefully. Like the subsequent programs you are asked to write for this exercise, it uses UDP (not TCP) sockets. You may borrow any code you feel necessary from this program, such as `MakeLocalSA`, `MakeDestSA`, `MakeReceiverSA`, `printSA`, `anyThingThere` and use them as utilities in the following exercises. Also note the 'include' files needed and the function prototype for `gethostbyname`.

Warm-up exercise. Take a copy of the file *UDPsock.c*. This file comprises a C program containing the examples given in the Notes on Sockets in Unix. It can be run either as a "sender" – a process that calls the sender procedure or as a "receiver" – a process that calls the receiver procedure. You should compile it and then run it as follows:
Log in to two computers. On one of them, run the program as a "receiver" by giving it the argument "r". On the other one, run the program as a sender by giving it the four arguments: "s", the name of the computer where you are running the receiver and two messages. The program prints out the socket address used and the messages received.

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`. 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` (such as the procedure identifier) 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 out.

Server: repeatedly receives a string using `GetRequest`, prints it on the screen and replies with `SendReply`. The server exits when the string consists of the single character 'q'.

Use the following (or equivalent C) definitions for `Status` and `SocketAddress`, which are in C++.

```
enum Status {
     Ok,         // operation successful
     Bad,        // unrecoverable error
     Wronglength           // bad message length supplied
};
typedef sockaddr_in SocketAddress;
```

Implement `DoOperation`, `GetRequest` and `SendReply`. The recommended prototypes are given in the definitions in the appendices. In the C++ definitions they are included in the classes `Client` and `Server`. The Status value returned reflects the values returned by `UDPsend` and `UDPreceive`.

`DoOperation`	sends a given request message to a given socket address and blocks until it returns with a reply message
`GetRequest`	receives a request message and the client's socket address
`SendReply`	sends 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 system calls `sendto` and `recvfrom`.

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

`UDPsend`	sends a given message through a socket to a given socket address
`UDPreceive`	receives a message and the socket address of the sender into two arguments

Use the recommended definitions for `SocketAddress` and `Message`, and the recommended prototypes for `UDPsend` and `UDPreceive`. In the C++ definitions, the latter are to be found in class `Socket`.

Choosing a server port

You will want to run server processes that can coexist with other people's processes in the same computer. You need to select an agreed port number for the server to receive messages from clients. Two servers on the same computer cannot use the same local port number. You will therefore want to choose a port number that is sure to be different from other people's port numbers. If everybody takes the first unreserved port number and adds their *uid*, there should be no such clashes – i.e.:

```
aPort =  IPPORT_RESERVED + getuid();
```

How unreliable are datagrams?

As part of this exercise, you should design an experiment to find out whether you can cause datagrams to be dropped. Describe the experiment and discuss its results in a comment in your client program.

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 by RPC - implemented by you. The two operations `Stop` and `Ping` are for all services.

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

Client and server should provide the ability to use `Stop` and `Ping` and in addition do arithmetic

Client: Each line 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.

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

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

The `Status` value should be extended with extra values required for this service, e.g DivZero. The code supporting the specific service (The Arithmetic Service) should be implemented as a separate part so that it could be replaced by the code for another service. You will need to define a new class (in C++) or a struct in C for RPC messages as suggested in the corresponding Appendices.

This exercise requires you to write marshalling and unmarshalling function members, which take network ordering into account, by using the functions `htonl` and `ntohl`. The recommended prototypes for marshal and unmarshal are given in the definitions. In C++ they are function members of the class `RPCMessage`.

Two other matters that 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 *requestId* 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 your client program. 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 a small, fixed number of 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 time out.

Timing-out 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 shows how to do this. You can test your time-out by running the client when the server is not running.

Presentations

Exercise 1

You must demonstrate that your programs work correctly by running two clients and a server on three different computers. You should also report whether you managed to make the server drop any messages, and what experiment you performed to test this.

Exercise 2

The result should be an arithmetic server that performs arithmetic operations for several clients, and which behaves sensibly when dealing with exceptional conditions.

Exercise 3

Demonstrate that after a time out, your client program re-tries a few times and then reports that it cannot contact the server.

Print-outs to hand in

The client and server programs should have separate code (unlike the demonstration program *UDPsock.c*).

Exercise 1

Provide the main program for the client and the server. The code for the client should include a comment containing a short write-up describing the experiment for testing the reliability of datagrams.

Exercises 2 and 3

Please supply the print outs in the following order:

- Header files with definitions for `SocketAddress`, `Status`, `Message`, `RPCMessage`. C++ programs will provide class interfaces, C programs will provide function prototypes;
- Implementation files for the classes or functions;
- Separate header and implementations for the Arithmetic service;
- A main for the client and the server (only one of each, Exercise 3 if possible).

APPENDIX 1: Definitions for C++ Programs

In these exercises you are advised to define the following:

```
class Message {
public:
      Message(unsigned char *, unsigned int ); // message and length supplied
      Message(unsigned int );                  // length supplied
private:
      unsigned char * data;
      unsigned int length;
};
```

The class Message is for holding the data and the length of a message. If you prefer you can make data a fixed length array with size 1000. The first constructor is used to create a message with given contents. The second constructor is used to create a message with a given length for putting the data into later. You may find that you need other function members, e.g. to return the data or length.

```
class Socket {
public:
      Socket();
      Socket(int);                // port given as argument
      status UDPsend(UDPMessage *m, SocketAddress * destination);
      status UDPreceive(UDPMessage **m, SocketAddress * origin);
private:
      int s;
      SocketAddress * socketAddress;
};
```

The class Socket to represents a datagram socket. It has data members for the socket descriptor and for the socket address to which it is bound. There are two constructors i) no arguments: opens the socket and binds it to any local port, ii) one argument: opens a socket and binds it to the given port. The function members send and receive messages via the given port.

```
class Client: public Socket  {
public:
        Client();           // calls constructor Socket()
status DoOperation (UDPMessage *callMessage, UDPMessage *replyMessage, SocketAddress * server);
};
class Server: public Socket {
public:
        Server(int);       // calls constructor Socket(int);
        status GetRequest (UDPMessage *callMessage, SocketAddress * client);
        status SendReply (UDPMessage *replyMessage, SocketAddress * client);
};
```

You can define another class or classes whose instances have the operations `DoOperation`, `GetRequest` and `SendReply`. One way to do this is to define two new classes called `Client` and `Server`, both of which are subclasses of the Socket class, as shown above.

C++ definitions for Exercise 2.

A class for RPC messages:

```
enum MessageType { Request, Reply};
class RPCMessage{
public:
      RPCMessage(MessageType);
      RPCMessage(MessageType, int, int, int);
      void marshall( Message ** ); // marshalls self to Message argument
      void unmarshall(Message *);   // unmarshalls from given message to self
private:
```

```
      MessageType type;
      unsigned int requestId;
      unsigned int procedureId;        // e.g.(1,2,3,4) for (+, -, *, /)
      int arg1, arg2;                  // arguments/ return parameters
};
```

This class has data members corresponding to the structure given in Figure 4.13, but without the remote object reference which is not needed for RPC. The fields *arg1* and *arg2* can be used for operation arguments or returned value and status. RPCMessage has two constructors: the first takes a message type as argument and the second takes values of all 4 data members as arguments. The two function members deal with marshalling and unmarshalling. You may add other function members, e.g. a function that executes a dispatcher given as a function argument. You could consider using a static data member for the next *requestId*.

Note also that you will need to include a definition of the prototypes of the socket system calls in an extern "C" statement in any ".c" file that uses them. This can be achieved by an include line such as the following:

```
#include "/import/GCC/lib/g++-include/sys/socket.h"
```

You may also need the following:

```
extern "C" {
     char * inet_ntoa(struct in_addr);
}
```

APPENDIX 2: Definitions for C Programs

In these exercises 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 */
     WRONGLENGTH            /* bad message length supplied */
} Status;
typedef struct sockaddr_in SocketAddress ;
```

You may alternatively define data as a pointer. The prototypes for DoOperation, GetRequest and SendReply (to which you must adhere) are as follows:

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

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

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

The prototypes for UDPsend and UDPreceive are as follows:

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

Status UDPreceive(int s, 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 */
} RPCMessage;    /* each int (and unsigned int) is 32 bits = 4 bytes */
```

The fields *arg1* and *arg2* can be used for operation arguments or returned value and status.

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

```
void marshal(RPCmessage *rm, Message *message);

void unMarshal(RPCmessage *rm, Message *message);
```

This assignment is due 4 weeks from the 20th of February, 2018.