Using the UNIX sockets and UDP to implement Remote Invocation Middleware

Prof. Amr El-Kadi

Introduction

This project involves 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 *request-reply protocol*. You are to use Linux machines (you do not need administrator privileges to develop these exercises). A good tutorial on UNIX sockets was provided in PDF. You are to use C++ prototypes. Definitions for use with C++ are given as appendices at the end of this document.

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.

The client and server behave as follows:

Client: this takes the name of the server computer as an argument (i.e. at the command line). 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' is first received.

The client and the server programs should be built on top of an OO class hierarchy, which utilizes the sockets and concurrency primitives.

You may use the set of C++ class interfaces presented in the appendix to build your client and server infrastructure, or design your own if you want.

Demonstration

You must demonstrate that your programs work correctly by running two clients and a server on three different computers. 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: A Community Steganography-Based Image Service using Remote Invocation

In this exercise you are required to design and implement an Image Service that is capable of storing a special type of image files. The service should receive an image from a client together with a set of attributes to be embedded seamlessly into the image using steganography techniques. Steganography is a technique of concealing data inside another data, and is extensively being used in media whereby secret data can be hidden into an image files.

The client need to have a GUI interface where by it can initiate requests to the image service using an application layer protocol based on RPC of your design. Each request from the client needs to include authentication credentials where by the user get authenticated against a back end authentication database. A user can store images on the service, each image is coupled with the user credential and a set of usernames who are allowed to view the image as well as the number of times that each user can view the image should be defined in the data sent to the service with the image to be stored. Using steganography, the service should package all the data sent to it and hide it into the image. Please refer to the Useful Resources section for more information about steganography and the recommended tools.

Users can only see their own images that they uploaded, or images that have their usernames hidden into them. Also each time a user views an image, the number of views needs to be updated inside the image and the user is denied view to the image when the number of views gets

consumed. The owner of the image has the right to update the image data by adding or removing users from the image or changing their viewing quota. Moreover, users can retrieve a list of their own images and a list of all the images they can view with the remaining view quota for each image (The use of robots that roam the service to collect such data is encouraged).

You are to design and implement the Server and the Client middleware using C++ and the QT GUI library. You are also required to design the RPC protocol message and implement all the marshaling and unmarshalling of messages between the client and the server, which should take network ordering into account, by using the functions *htonl* and *ntohl*. You will need to extend the Message class to define an RPC messages as suggested in the corresponding Appendices with updated message types to accommodate the new functionality. Also extra fields might need to be defined such as file name, permission attributes, etc. The server should be able to handle multiple concurrent connections from different clients.

You are required to design and implement all the operations and their corresponding RPC messages in order to fulfill the required functionalities.

Three other matters that should be addressed:

- All packet loss and packet reordering should be handled by your communication infrastructure (see UDPSocket Class in the Appendix)
- Both client and server should make use of the *messageType* field of the M*essage* class after unmarshaling to test that Requests and Replies are not confused;
- 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.

Demonstration

You must demonstrate that your programs work correctly by running three users on three different computers. The result should be an image sharing and storing service that performs image repository and display, and which behaves sensibly when dealing with exceptional conditions, such as shared resources, invalid/corrupted messages, unauthorized access, etc.

Exercise 3 Adding fault-tolerance

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.

Demonstration

Design demonstrations that would show fault-tolerance capabilities of your system.

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;
- Sources for client and server functionalities.
- For exercise 2, a short report (1 to 2 pages) should be written to describe the experiment conducted to demonstrate how clients and the server benefited from the use of the *RPCId* to match a reply with its request. Cross-references to the source code files are expected.
- For exercise 3, a short report (1 to 2 pages) should be written to describe the experiments conducted to demonstrate fault-tolerance capabilities of your system. Cross-references to the source code files are expected.

APPENDIX 1: Definitions for C++ Programs

In these exercises you are advised to define the following:

```
#ifndef MESSAGE_H
#define MESSAGE_H

enum MessageType { Request, Reply};

class Message
{
     private:
          MessageType message_type;
          int operation;
          void * message;
          size_t message_size;
          int rpc_id;
     public:
          Message(int operation, void * p_message, size_t p_message_size,int p_rpc_id);
          Message(char * marshalled_base64);
          char * marshal ();
          int getOperation ();
          ing getRPCId();
          void * getMessage();
          size_t getMessageSize();
          MessageType getMessageType();
          void setOperation (int _operation);
          void setMessage (void * message, size_t message_size);
          void setMessageType (MessageType message_type);
          ~Message();
};
#endif // MESSAGE_H
```

This class encapsulates the state of the message to be sent between a client and a server. Notice the method "marshal" which converts the data members of an object instantiated from this class to a data stream. I recommend using a data stream format that avoids the null character as you may send binary data; a good candidate for that is the base64 format, and you can read about the base64 encode and decode here http://en.wikipedia.org/wiki/Base64. Of course there are other alternatives that you can use but you need to be very careful and have very good reasons for your choice. The "unmarshalling" is carried out via the second parameterized constructor, which receives a base64 formatted character array. In exercise 1 you will not need to utilize the "operation" and "rpc_id" data members but it will be needed in the next phases of the project. The message type should be set by the client and the server objects whose class definitions are presented below. For exercise 2 you will definitely need to extend this class either through inheritance of through amendment to account for other data attributes of the media file to be transferred such as file name, file attributes, storage location, etc.

```
#ifndef UDPSOCKET_H
#define UDPSOCKET_H

class UDPSocket
{
     protected:
          int sock;
          sockaddr_in myAddr;
          sockaddr_in peerAddr;
          char * myAddress;
          char * peerAddress;
          int myPort;
          int peerPort;
          bool enabled;
          pthread_mutex_t mutex;
     public:
          UDPSocket ();
          void setFilterAddress (char * _filterAddress);
          char * getFilterAddress ();
          bool initializeServer (char * _myAddr, int _myPort);
          bool initializeClient (char * _peerAddr, int _peerPort);
          int writeToSocket (char * buffer,  int maxBytes );
          int writeToSocketAndWait (char * buffer, int  maxBytes,int microSec );
          int readFromSocketWithNoBlock (char * buffer, int  maxBytes );
          int readFromSocketWithTimeout (char * buffer, int maxBytes, int timeoutSec,
int timeoutMilli);
          int readFromSocketWithBlock (char * buffer,  int maxBytes );
          int readSocketWithNoBlock (char * buffer, int  maxBytes );
          int readSocketWithTimeout (char * buffer, int maxBytes, int timeoutSec, int
timeoutMilli);
          int readSocketWithBlock (char * buffer,  int maxBytes );
          int getMyPort ();
          int getPeerPort ();
          void enable();
          void disable();
          bool isEnabled();
          void lock();
          void unlock();
          int getSocketHandler();
          ~UDPSocket ( );
};

#endif // UDPSOCKET_H
```

The UDPSocket class encapsulates all the data and methods necessary to establish communication between two entities. This class acts as a parent class to the UDPClientSocket and the UDPServerSocket. Notice the wealth of methods for reading and writing from a UDP socket with different timeout options.

```
#ifndef UDPCLIENTSOCKET_H
#define UDPCLIENTSOCKET_H

class UDPClientSocket : public UDPSocket
{

     public:
          UDPClientSocket ();
          bool initializeClient (char * _peerAddr, int _peerPort);
          ~UDPClientSocket ( );
};

#endif // UDPCLIENTSOCKET_H
```

This class represents a client UDP socket and it inherits from theUDPSocket class. It has an initialization method that initializes the UDPSocket in a client mode

```
#ifndef UDPSERVERSOCKET_H
#define UDPSERVERSOCKET_H

class UDPClientSocket : public UDPSocket
{

     public:
          UDPServerSocket ();
          bool initializeServer (char * _myAddr, int _myPort);
          ~UDPServerSocket ( );
};

#endif // UDPSERVERSOCKET_H
```

This class represents a server UDP socket and it inherits from the UDPSocket class. It has an initialization method that initializes the UDPSocket in a server mode. You will need to couple that with concurrency and multithreading constructs to be able to serve different clients simultaneousloy.

```
#ifndef CLIENT_H
#define CLIENT_H

class Client
{
     private:
          UDPClientSocket * udpSocket;
     public:
          Client(char * _hostname, int _port);
          Message * execute(Message * _message);
          ~Client();
};
#endif // CLIENT_H
```

The Client class encapsulates the client functionalities. It has an instance of the UDPClientSocket class, which implements the UDP client. The constructor receives the host name that the server process resides on as well as the port number. The execute method receives a message which is transmitted for remote execution, and the reply message is returned. Prior to sending the request, the message type should be set as a “Request”. The type of the message received as a reply should be checked and its type should be a “Reply”. All the network operations details should be encapsulated inside the execute method, and the calling routine should not be aware of the remote execution.

```
#ifndef SERVER_H
#define SERVER_H

class Server
{
     private:
          UDPServerSocket * udpServerSocket;
          Message * getRequest();
          Message * doOperation();
          void sendReply (Message * _message);
     public:
          Server(char * _listen_hostname, int _listen_port);
          void serveRequest();
          ~server();
};
#endif // SERVER_H
```

The server class encapsulates all the server operation. The constructor receives two parameters, which indicate the interface and the port that the server should listen on. The public method “serveRequest” should block upon calling it until a message is received from a client. The message type needs to be checked and verified that it is of type “Request” message. The method will use the three private methods “getRequest”, “doOperation”, and “sendReply” to execute the received request and send the results back to the client. All deserialization/unmarshalling and serialization/marshalling should take place inside “getRequest” and “sendReply” methods respectively. The “handleRequest” method acts as a dispatcher, which might need to instantiate more objects from different classes based on the message operation id to be able to perform the request tasks. For exercise 1, the logic of the echo server should be implemented inside the “doOperation” method without the need for the dispatcher. Prior to sending the reply, the message type should be set as a “Reply”.

Useful Resources

To help you start with Linux, we have compiled a list of Linux resources that would help take you from novice stage to Kernel expert (as needed). The list is the work of the very large community of people who have worked with the Linux kernel.

For Starters

- Kernel Newbies: Intended to help new kernel hackers
- Linux Questions: A civil forum asking linux questions
- Ubuntu Support Forums: From the best-known consumer distribution of Linux, including a section for absolute beginners

Kernel News

- Linux Weekly News (LWN.net)

Kernel Hacking

- Here is where you can get any kernel you want: Linux Kernel Archives at kernel.org
- The Linux Kernel Documentation Project
- The Linux Kernel by David A Rusling
- Linux Kernel Internals by Tigran Aivazian
- The Linux Kernel Hackers' Guide: HTML

Linux Kernel Debuggers

- KDB: built-in kernel debugger with multiprocessor support
- Kgdb: kernel debugger used through serial line

Source Navigators

The source navigators provide browse-able kernel code, making it easier to read than your ssh window or emacs. However, the kernel is not necessarily the same version as the kernel you will be working with, so watch out for subtle differences.

- Linux Source Navigator
- Another Linux Source Navigator

Steganography

- http://en.wikipedia.org/wiki/Steganography
- http://easybmp.sourceforge.net/steganography.html
- http://manytools.org/hacker-tools/steganography-encode-text-into-image/
- http://www.maketecheasier.com/hide-confidential-data-inside-images-in-linux/
- http://www.findbestopensource.com/tagged/Steganography
- http://punkroy.drque.net/PNG_Steganography/SteganographyC.php

C Language Resources

As Linux is written mostly in C, if you are unfamiliar with the language, we provide you with few links that will jump-start your C capabilities in no time.

- C programming.com
- ANSI C On Unix Systems
- Common C Problems
- C Frequently Asked Questions
- This includes a tutorial and full reference to C: Programming In C: Unix System Calls and Subroutines in C