Designing and Implementing an Application Layer Network Protocol Using UNIX Sockets and TCP

Ryan Lattrel
ECE 480 Design Team 2

11/16/2012

Abstract

Using Sockets with TCP is a way to reliably send data between two computers over a standard Ethernet device. Many different methods and libraries exist in order to accomplish this. These notes will focus on using Berkeley UNIX sockets and the TCP transport layer protocol using C++ and Linux. Data is formatted and transmitted in the application layer, which operates on top of TCP. An application specific design is necessary in order to effectively relay data between devices, and these notes will illustrate creating and using a custom protocol.

Keywords: TCP, Berkeley Sockets, Application Layer, Linux

Contents

1 Introduction **3**

2 Using TCP **3**
2.1 Overview 3
2.2 Server-side Configuration 4
2.3 Client-side Configuration 5
2.4 Common Functions 6

3 Designing an Application Protocol **7**
3.1 Overview 7
3.2 Packet Design 7
3.3 Message types 7
3.4 Payload design 8

4 Implementing an Application Protocol **8**
4.1 Overview 8
4.2 Sending a Message 8
4.3 Receiving a Message 9
4.4 Handling Message Types 9

5 Conclusions **10**

6 Appendix **10**

1 Introduction

The ability to send data across a network or between multiple computers is often needed when developing applications. These notes describe a technique to abstract low level socket code, as well as a method to develop an application layer protocol on top of this abstracted code. Being able to send relevant and reliable messages between computers is important and the method described is reusable and applicable in many different applications. Free and open source software is also often needed and these notes explain how to create networked programs in C++ using Linux.

2 Using TCP

2.1 Overview

TCP is a standard network protocol used to reliably transmit data between two computers. TCP is the foundation for the modern internet, and without it - or without a similar protocol - there would be no way to send large amounts of data over a network accurately. The protocol can easily be implemented and used by any programmer for any application. It is the best way to ensure reliable data transfer between an applications running on two separate devices or computers. There are two different configurations that have to be considered: server-side and client-side. The server is the device that will wait for an incoming connection request from a client. Once the connection has been established both the server and the client can send and receive data. The following section will define the details needed in order to set up a connection to send data over.

The best way to use Berkeley UNIX sockets is by implementing a Socket class in order to take advantage of the lower level functionality of the library. This prevents having to write confusing function calls numerous times, and instead all of the unnecessary detail can be abstracted away in the functions of the custom socket class. The following sections will give an example of how to separate the functionality of the socket class, and the appendix will give an example of using this these functions in a class. To better describe the functions in the following examples, the variables used are shown in the code section below.

Socket class member variables

```
int socket_fd;
sockaddr_in remote;
sockaddr_in local;
char buf[BUFLEN];
char retbuf[BUFLEN];
socklen_t len;
socklen_t rlen;
hostent *hp;
```

2.2 Server-side Configuration

A server has two steps in order to begin the server and wait for connections. The first is setting up the socket to listen on a specific port, and the second is to actively wait for incoming connections. The first step is in the example below. In the code bind() and listen() are the function calls that begin the server. The bind command attaches the socket to a port, and the listen command tells the interface to prepare to queue incoming connection requests.

TCP Server: initialization example

```
int CSocket::StartTCPServer(int port, int address)
{
      //create socket
      socket_fd = socket(AF_INET, SOCK_STREAM, 0);

      //setup socket
      local.sin_family = AF_INET;
      local.sin_addr.s_addr = address;
      local.sin_port = port;

      //bind socket to addr
      bind(socket_fd, (struct sockaddr *)&local, len);
      listen(socket_fd, LISTEN_QUEUE_NUMBER);

      //return port number
      getsockname(socket_fd, (struct sockaddr *)&local, &len);
      return local.sin_port;
}
```

After the server has been set to listen, a call must be made which waits for a client to try to connect. This call is blocking, and it is ideal to call this in a thread to allow the program to complete other tasks while waiting for an incoming connection. When the connection is accepted, a new file descriptor is returned. In order to read or write to this connection the new file descriptor is to be used, and not the file descriptor that is listening for incoming connections. The socket waiting for connections can be closed if only one connection is expected, or left open to allow multiple clients to connect simultaneously.

TCP Server: wait for incoming connection

```
CSocket* CSocket::AcceptTCPConnection()
{
      CSocket * new_socket = NULL;
      int accept_connection_fd = accept(socket_fd, (struct
          sockaddr *)0, (socklen_t*) 0);
      if(accept_connection_fd != -1)
```

```
	{
		new_socket = new CSocket(accept_connection_fd);
	}
	return new_socket;
}
```

The function written here returns a new socket when the connection is accepted, created from the file descriptor returned by the accept function. In order for this to work a constructor must be created that will accept a file descriptor. An example is below.

TCP Server:alternative constructor

```
CSocket::CSocket(int fd)
{
	socket_fd = fd;
	len = sizeof(local);
	rlen = sizeof(remote);
}
```

After the connection has been accepted it can be written to and read to. These commands will be detailed in the Common Functions section.

2.3 Client-side Configuration

Configuring the client is significantly simpler than setting up the server. Only one call will need to be made before the socket can use read and write. The command is called connect, and it will require the IP address and port number of the server it needs to connect to. The configuration below will allow the client to connect to a server using both an IP address or a hostname.

TCP Client: Connect to server example

```
int CSocket::ConnectToTCPServer(int port, char* name)
{
	//create socket
	socket_fd = socket(AF_INET, SOCK_STREAM, 0);

	//setup socket
	remote.sin_family = AF_INET;
	hp = gethostbyname(name);
	memcpy(&remote.sin_addr, hp->h_addr, hp->h_length);
	remote.sin_port = port;

	//connect to server
```

```
        if(connect(socket_fd, (struct sockaddr *)&remote,
            sizeof(remote)) < 0)
        {
                return -1;
                close(socket_fd);
        }
        return 0;
}
```

In the above example, if the server does not respond correctly the socket is closed. If it is successful the read and write commands detailed in the next section can then be used to transmit data.

2.4 Common Functions

After configuring the server or the client, the sockets will act identically and it does not matter which side either of the command functions are called from. The two common functions are read, and write. The read function will be detailed in the example below. The recv() command will copy data from the socket to a buffer. The second parameter of the receive function is the amount of data to read from the socket so it is necessary to know how much data is expected to be sent from the client.

TCP: Receive message

```
int CSocket::ReceiveTCPMessage(char *buffer, int buffer_size)
{
        int msglen = recv(socket_fd, buffer, buffer_size, 0);
        return msglen;
}
```

The write command is very simple, and is similar to the recv command. Instead of copy data from the socket to the buffer, it simply copies data from the buffer to the socket.

TCP: Send message

```
int CSocket::SendTCPMessage(char* msg, int msg_size)
{
        return send(socket_fd, msg, msg_size, 0);
}
```

3 Designing an Application Protocol

3.1 Overview

In the above sections, it is shown how to read and write data to a char array. This is not the easiest way to pass data between computers, and it is better to design a protocol that abstracts these arrays. A way to do this is by casting a well structured packet into a char array when using the functions in the previous section. As long as both the client and server are aware of the size and structure of the packet there is no risk when applying the cast.

3.2 Packet Design

For this tutorial there are two pieces of data needed in a packet. The first is the message type, and the second is the attached data buffer. The message type will describe what type of data to expect in the attached buffer. This type can be used by the sending program and the receiving program to determine what type of data to attach to the message, and what type of data can be expected to be found in the message received. The structure below represents a packet that will be sent over the socket. As needed fields can be added to the message struct, as different header values might be required depending on the application type. This example only requires a type in the header. Other fields could be the size of the message, an id or key from the sender, as well as a packet or block number.

Packet structure

```
struct message
{
      message_type type;
      char payload[PAYLOAD_SIZE];
};
```

3.3 Message types

Defining message types is a fundamental part of the protocol design. The message type should be short and descriptive, typically an integer at the beginning of the packet. A good way to implement this is by using an enumerator in order to avoid hard-coding integers into the code. Each message can be handled differently based on the type, how to do this is described in a later section. An example of defining message types is below. The integers used to describe a type are arbitrary and can be determined by the programmer. This example protocol is based on a program to track and detect targets. Track is a command which begins the tracking process, stop ends it, and success relays information about a target.

Message types

```
enum message_type {CONNECT_OK = 1, TRACK = 10, TRACK_CONFIRM =
    15, STOP = 20, STOP_CONFIRM = 25, SUCCESS = 30, FAILURE =
    40, QUIT = 50};
```

3.4 Payload design

Some messages will need to have data attached, instead of being just a simple command. Functions like 'QUIT' and 'TRACK' and 'STOP' in this example do not require and data in the payload, however, messages like 'SUCCESS' and 'FAILURE' do. A payload is any useful data that is sent and received. It can be a string of text or any number of integers. It all depends on the application. In this application the detected target has an ID and set of parameters. This is described in a payload structure which can be copied into the message payload buffer. It is necessary to only use basic types and not to use pointers, because when the data is transferred it will only copy the data contained in the structure, and not data it is pointing to (unless you redefine the sending function to anticipate this format). Be sure that the size of the payload buffer in the packet is large enough to fit the data structure.

Payload structure

```
struct human_data
{
      int human_id;
      int torso_height;
      int arm_length;
      int leg_length;
      int head_diameter;
};
```

4 Implementing an Application Protocol

4.1 Overview

Now that the packet structure has been defined, a way to write it, read it, and handle the different message types and data will be needed. This is done by using the functions described in the Using TCP section, and the data will be cast to fit the functions defined in that section. Switch statements are an ideal way to handle different message types received and this will be documented in the Handling Message Types subsection.

4.2 Sending a Message

Using the functions in the Using TCP section a packet can be sent, as stated, by casting. The code example below shows how this is accomplished using the previously described methods. First a message struct needs to be created. Next a message type needs to be defined, and finally the payload data (if needed) needs to be copied into the outgoing message. The message is then sent over the network. The following code explains how to do this.

Transmitting the message

```
//Create message and payload data
message outgoing_message;
human_data data;

//Define type
outgoing_message.type = SUCCESS;

//Copy data into message payload buffer
memcpy((void*)outgoing_message.payload, (void*)&data,
    sizeof(data));

//Transmit message over TCP with cast
my_socket->SendTCPMessage((char*)&outgoing_message,
    sizeof(outgoing_message));
```

4.3 Receiving a Message

Receiving a message is very similar to sending one. It is important to note that when the ReceiveTCPMessage command is called, the program will be blocked until the bytes requested is received. The data received in the payload can be cast to the type of data expected. Since the type of data can change depending on the message type it is important to handle each message differently, this is described in the next section.

Receiving the message

```
//Create message and payload data
message incoming_message;

//Receive the message
my_socket->ReceiveTCPMessage((char*)&incoming_message,
    sizeof(incoming_message));

//The data will be located in the message
human_data* data = (human_data*)&incoming_message.payload;
```

4.4 Handling Message Types

Since the message data is a cast of the payload, and because different blocks of code will need to be implemented depending on the message type a way is needed to handle each message type. An easy way to do this is using a switch statement in a receive function. Once the packet is received from the socket, a switch can be used based on the incoming message type to determine how to handle the message. An example is below.

Handling the message

```
my_socket->ReceiveTCPMessage((char*)&incoming_message,
    sizeof(incoming_message));

//Handle message based on type
switch(incoming_message.type)
{
      case TRACK:
      {
            //TRACK code block
            break;
      }
      case STOP:
      {
            //STOP code block
            break;
      }
      case SUCCESS:
      {
            human_data* data =
                (human_data*)&incoming_message.payload;
            //SUCCESS code block
            break;
      }

      //Add more cases based on the number of types to be
          received
}
```

5 Conclusions

This method of implementing a protocol is highly versatile and can be modified to suit any program's needs. TCP guarantees the integrity of data sent between computers and devices, and is generally the best transport layer protocol to use. As seen in the example code, a good way to use Berkeley Sockets is by creating a class that simplifies and abstracts the lower level code. In addition to abstracting the low level code, in order to send custom messages a protocol must also be developed and shared between the client and server applications.

6 Appendix

CSocket.h

```
#include <cstring>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define LISTEN_QUEUE_NUMBER 255

/*****************************************************************
*Name: CSocket
*Description: Makes using Berkeley sockets easy.
*****************************************************************/
class CSocket
{
public:
      CSocket();
      CSocket(int fd);
      ~CSocket();

      int SendTCPMessage(char* msg, int msg_size);
      int StartTCPServer(int port, int address);
      CSocket* AcceptTCPConnection();
      int ConnectToTCPServer(int port, char* name);
      int ReceiveTCPMessage(char *buffer, int buffer_size);

      int CloseSocket();
private:
      int socket_fd;
      sockaddr_in remote;
      sockaddr_in local;
      char buf[BUFLEN];
      char retbuf[BUFLEN];
      socklen_t len;
      socklen_t rlen;
      hostent *hp;
};
```

CSocket.cpp

```
#include "csocket.h"
#include <cstdio>
#include <cstdlib>

#include <iostream>
#include <sstream>
```

```
#include <cstring>

using namespace std;

CSocket::CSocket()
{
      len = sizeof(local);
      rlen = sizeof(remote);
}

/*********************************
 * Name: CSocket
 * Purpose: Create a socket given a file descriptor
 * Receive: int filedescriptor
*********************************/
CSocket::CSocket(int fd)
{
      socket_fd = fd;
      len = sizeof(local);
      rlen = sizeof(remote);
}

CSocket::~CSocket()
{
      this->CloseSocket();
}

/*********************************
 * Name: SendTCPMessage
 * Purpose: Send a TCP message on socket
 * Receive: char* msg, int msg_size
 * Return: int
*********************************/
int CSocket::SendTCPMessage(char* msg, int msg_size)
{
      return send(socket_fd, msg, msg_size, 0);
}

/*********************************
 * Name: ReceiveTCPMessage
 * Purpose: Receive a TCP message on socket.
 * Receive: char *buffer, int buffer_size
 * Return: message length
*********************************/
int CSocket::ReceiveTCPMessage(char *buffer, int buffer_size)
{
      int msglen = recv(socket_fd, buffer, buffer_size, 0);
      return msglen;
}
```

```
/********************************
 * Name: StartTCPServer
 * Purpose: Start a TCP server on given port and address
 * Receive: int port, int address
 * Return: port TCP server is running on
********************************/
int CSocket::StartTCPServer(int port, int address)
{
      //create socket
      socket_fd = socket(AF_INET, SOCK_STREAM, 0);

      //setup socket
      local.sin_family = AF_INET;
      local.sin_addr.s_addr = address;
      local.sin_port = port;

      //bind socket to addr
      bind(socket_fd, (struct sockaddr *)&local, len);
      listen(socket_fd, LISTEN_QUEUE_NUMBER);

      //return port number
      getsockname(socket_fd, (struct sockaddr *)&local, &len);
      return local.sin_port;
}

/********************************
 * Name: AcceptTCPConnection
 * Purpose: Waits for a TCP connection and creates a new socket
 * Receive: none
 * Return: Socket to new TCP connection
********************************/
CSocket* CSocket::AcceptTCPConnection()
{
      CSocket * new_socket = NULL;
      int accept_connection_fd = accept(socket_fd, (struct
          sockaddr *)0, (socklen_t*) 0);
      if(accept_connection_fd != -1)
      {
            new_socket = new CSocket(accept_connection_fd);
      }
      return new_socket;
}

/********************************
 * Name: ConnectToTCPServer
 * Purpose: Connects to a TCP server on given port and hostname
 * Receive: int port, char* name
 * Return: int
********************************/
int CSocket::ConnectToTCPServer(int port, char* name)
```

```
{
      //create socket
      socket_fd = socket(AF_INET, SOCK_STREAM, 0);

      //setup socket
      remote.sin_family = AF_INET;
      hp = gethostbyname(name);
      memcpy(&remote.sin_addr, hp->h_addr, hp->h_length);
      remote.sin_port = port;

      //connect to server
      if(connect(socket_fd, (struct sockaddr *)&remote,
          sizeof(remote)) < 0)
      {
            return -1;
            close(socket_fd);
      }

      return 0;
}

/**********************************
 * Name: CloseSocket
 * Purpose: closes a socket
 * Receive: none
 * Return: int
**********************************/
int CSocket::CloseSocket()
{
      close(socket_fd);
      return 0;
}
```

Starting a TCP server

```
//Begin listening
TCPServer = new CSocket();
TCPServer->StartTCPServer(port, ip_address);

//Accept connection
CSocket* pc_connection = TCPServer->AcceptTCPConnection();

//Send connect_ok message
message outgoing_message;
outgoing_message.type = CONNECT_OK;
pc_connection->SendTCPMessage((char*)&outgoing_message,
    sizeof(outgoing_message));
```

Starting a TCP client

```
//Connect to localhost on port 9999
char *server_name_str = ``127.0.0.1'';
unsigned short int tcp_server_port = 9999;

//Connect to the server
CSocket* TCPClientSocket = new CSocket();
TCPClientSocket->ConnectoToTCPServer(tcp_server_port,server_name_str);

//Receive a message
message incoming_message;
TCPClientSocket->ReceiveTCPMessage(char*)&incoming_message,
    sizeof(incoming_message);
```