A Guide to Libsocketpp

A tutorial to C++ streambased sockets

Charlie Sale

This manual is for libsocketpp, 0.2

Published by GNU Manual

Table of Contents

1 Overview

Welcome to libsocketpp, the C++ library for networking sockets. If you don't know already, a socket is a connection between two networking hubs like computers and modems. Before now, the standard socket system for C++ was the C socket system. Although excellent for C, the C socket system was designed for use in the C programming language, not C++. One of the key differences between C++ and C is that C++ is object oriented, which means that it uses classes and objects. One of the objectives of libsocketpp was to integrate the C socket system into an object oriented system built for C++.

Another objective of libsocketpp is to integrate the C socket system into the C++ standard I/O system. The standard C++ I/O system is built around buffer and stream classes. A buffer is a container of data to be moved over a stream, and a stream is a connection between two points that sends and recieves data between the two points. The stream writes data into and reads data from a buffer. Libsocketpp works on this system because the socket sends and recieves data across the internet and stores the data into the buffer.

2 A Basic Tutorial

Now that you have libsocketpp installed and configured, let's cover the basics on usage.

2.1 TCP

The primary socket set used in libsocketpp is the TCP socket. If you don't know already, TCP sockets are streambased, which means it fits perfectly into this streambased library. The TCP socket process goes as such:

SERVER:

1. A socket descriptor is created
2. The socket descriptor is bound to a port
3. The socket descriptor then calls a blocking process to listen for incoming socket connections.
4. Once a connection is found, a socket descriptor representing the accepted client is then returned for use.
5. With that socket descriptor, a stream is set up between the server and now connected. Now, data can be sent to and from each connected member via blocking read and write calls.
6. Eventually, either the server or client will disconnect, terminating the stream.

CLIENT:

1. A socket descriptor is created
2. The socket descriptor then connectes to the host and port on which a server is bound.
3. Once the client is connected, it can now send and recieve data with the server via the same blocking calls implemented by the server.
4. Eventually, the socket will terminate it's connection with the server, and the stream is closed.

The classes in libsocketpp work the same way as this. Here is an example of a client program.

```
#include <socketpp/tcp/socket.h>
#include <iostream>

using namespace std;
using namespace tcp;

int main(int argc, char** argv)
{
  Socket sock("192.168.1.1", 8888);
  sock.connects();

  sock << "Hello World" << endl;

  sock.closes();
```

```
  return 0;
}
```

Here is a breakdown of what each line does:

1.
```
#include <iostream>
#include <socketpp/tcp/socket.h>
```
The first include statement includes the `tcp::Socket` class for use in the program. The second includes I/O header files for C++.

2.
```
using namespace std;
using namespace tcp;
```
These lines declare the usage of namespaces in our program. All of the TCP socket classes are found in the namespace `tcp`. While neither of these lines are necessary, they allow you to write `Socket sock` instead of `tcp::Socket sock` every time you want to declare a Socket class

3.
```
int main(int argc, char** argv) {
```
This line is the main entry point to your C function.

4.
```
Socket sock("192.168.1.1", 8888);
sock.connects();
```
The first one declares a `Socket` object called `sock`. The constructor parameters are the connection values. The string being the host to connect to and the integer being the port on which you are going to connect. The next line, `sock.connects()` connects your socket object to the server with the supplied data. The `connects` method can also be used to set connection data in the same way as done in the constructor.

5.
```
sock << "Hello World" << endl;
```
This line writes the string `"Hello World"` to the server that it connected to. This is done in the same manor as you would do when printing text to `stdout` via cout, meaning you use the formatted output operator, `<<`, to write text. Take special note of the `endl` at the end of the statement. **An `endl` is required to send any data using the formatted output operator.** I will explain why later.

6.
```
sock.closes();
return 0;
```
The line `sock.closes();` terminates the connection between the socket and server, and the line `return 0;` is the standard successful return value from main.

3 An Indepth Tutorial

4 Extending

One of the goals of this project is to make the library extendable for your own usage. By this, I mean that the classes are accessable to the user for their own derivatives of them. Here is an overview of what each base class does and tips/hints on how to extend them for your own purposes.

4.1 tcp::basic_socket : socketpp/tcp/basesocket.h

The most important base class in this project is the class `tcp::basic_socket`. This class contains all basic socket functionality. This includes connection, getting and setting options, sending and recieving data, getting internal data, and closing the connection. Keep in mind that this class is a true base class, so all methods except for the constructors are `protected`. If you want to use this class, it needs to be derived. For example:

```
#include <socketpp/tcp/basesocket.h>

class MyClass : public tcp::basic_socket
{
private:
    int var;

public:
    MyClass() : tcp::basic_socket()
    { }

    int foobar(){ return 1; }

    int closes(){ cout << "I am closed"; close(this->socketfd); }
};
```

In this mini example, a class called `MyClass` that inherites `tcp::basic_socket` is created. It creates it's own private data, calls the parent constructor, adds its own method (`foobar`), and overrides a method (`closes`). By extending this class, you get all basic socket functionality in your own class.

4.2 tcp::base_sock_buf : socketpp/tcp/basesockbuf.h

This base class inherits from the `std::streambuf` class, which is the class that tells a stream how to handle the I/O buffer. Inside of this class, there is an internal `basic_socket` object that is used for reading and writing internal data, which will be talked about later. Another important thing to remember is that the `base_sock_buf` class's copy and assignment constructors are protected, so using them as a standalone class will probably be a pain.

The `base_sock_buf` class has three important methods: `underflow`, `overflow`, and `sync`. These methods are the basics for reading and writing data over sockets.

Lets start with the `underflow` method. This method is used in reading data from a socket. When an `istream` calls a read call and the internal `sockbuf` is empty, then the `istream` calls the underflow command to repopulate the buffer with data. This will make the interal `basic_socket` object read more data and place it into the interal buffer so that there is data to be read by the user. In short, the internal buffer "underflows", so the `streambuf` must refill the buffer with more data.

The next method is `overflow`. This function can be seen as the opposite as underflow. An `ostream` calls this method when the internal write buffer is full, or it "overflows". This causes the stream to empty the data and write it to a location. In this case, the internal socket object sends all of the written data over the connection.

To call an overflow call without waiting to fill up the buffer, the `flush` method can be called, which empties what ever is in the buffer at the moment. The flush functionality is defined in the function `sync`. This is used very often in this class.

These three functions are the majority of this base class. If you have any development to do on this class, these function are where it will happen.

Index

(Index is nonexistent)