"Linux Gazette...*making Linux just a little more fun!*"

Linux Socket Programming In C++

By Rob Tougher

Contents

1. Introduction

Sockets are a mechanism for exchanging data between processes. These processes can either be on the same machine, or on different machines connected via a network. Once a socket connection is established, data can be sent in both directions until one of the endpoints closes the connection.

I needed to use sockets for a project I was working on, so I developed and refined a few C++ classes to encapsulate the raw socket API calls. Generally, the application requesting the data is called the client, and the application servicing the request is called the server. I created two primary classes, **ClientSocket** and **ServerSocket**, that the client and server could use to exchange data.

The goal of this article is to teach you how to use the **ClientSocket** and **ServerSocket** classes in your own applications. We will first briefly discuss client-server communications, and then we will develop a simple example server and client that utilize these two classes.

2. Overview of Client-Server Communications

Before we go jumping into code, we should briefly go over the set of steps in a typical client-server connection. The following table outlines these steps:

Server	Client
1. Establish a listening socket and wait for connections from clients.	
	2. Create a client socket and attempt to connect to server.
3. Accept the client's connection attempt.	
4. Send and receive data.	4. Send and receive data.
5. Close the connection.	5. Close the connection.

That's basically it. First, the server creates a listening socket, and waits for connection attempts from clients. The client creates a socket on its side, and attempts to connect with the server. The server then accepts the connection, and data exchange can begin. Once all data has been passed through the socket connection, either endpoint can close the connection.

3. Implementing a Simple Server and Client

Now its time to dig into the code. In the following section we will create both a client and a server that perform all of the steps outlined above in the overview. We will implement these operations in the order they typically happen - i.e. first we'll create the server portion that listens to the socket, next we'll create the client portion that connects to the server, and so on. All of the code in this section can be found in simple_server_main.cpp and simple_client_main.cpp.

If you would rather just examine and experiment with the source code yourself, jump to this section. It lists the files in the project, and discusses how to compile and test them.

3.1 Server - establishing a listening socket

The first thing we need to do is create a simple server that listens for incoming requests from clients. Here is the code required to establish a server socket:

listing 1 : creating a server socket (part of simple_server_main.cpp)

```
#include "ServerSocket.h"
#include "SocketException.h"
#include <string>

int main ( int argc, int argv[] )
{
  try
    {
      // Create the server socket
      ServerSocket server ( 30000 );

      // rest of code -
      // accept connection, handle request, etc...

    }
  catch ( SocketException& e )
    {
      std::cout << "Exception was caught:" << e.description() << "\nExiting.\n";
    }
```

```
  return 0;
}
```

That's all there is to it. The constructor for the **ServerSocket** class calls the necessary socket APIs to set up the listener socket. It hides the details from you, so all you have to do is create an instance of this class to begin listening on a local port.

Notice the try/catch block. The **ServerSocket** and **ClientSocket** classes use the exception-handling feature of C++. If a class method fails for any reason, it throws an exception of type **SocketException**, which is defined in SocketException.h. Not handling this exception results in program termination, so it is best to handle it. You can get the text of the error by calling **SocketException**'s **description()** method as shown above.

3.2 Client - connecting to the server

The second step in a typical client-server connection is the client's responsibility - to attempt to connect to the server. This code is similar to the server code you just saw:

listing 2 : creating a client socket (part of simple_client_main.cpp)

```
#include "ClientSocket.h"
#include "SocketException.h"
#include <iostream>
#include <string>

int main ( int argc, int argv[] )
{
  try
    {
      // Create the client socket
      ClientSocket client_socket ( "localhost", 30000 );

      // rest of code -
      // send request, retrieve reply, etc...

    }
  catch ( SocketException& e )
    {
      std::cout << "Exception was caught:" << e.description() << "\n";
    }

  return 0;
}
```

By simply creating an instance of the **ClientSocket** class, you create a linux socket, and connect it to the host and port you pass to the constructor. Like the **ServerSocket** class, if the constructor fails for any reason, an exception is thrown.

3.3 Server - accepting the client's connection attempt

The next step of the client-server connection occurs within the server. It is the responsibility of the server to accept the client's connection attempt, which opens up a channel of communication between the two socket endpoints.

We have to add this functionality to our simple server. Here is the updated version:

listing 3 : accepting a client connection (part of simple_server_main.cpp)

```
#include "ServerSocket.h"
#include "SocketException.h"
#include <string>

int main ( int argc, int argv[] )
{
  try
    {
      // Create the socket
      ServerSocket server ( 30000 );

      while ( true )
        {
          ServerSocket new_sock;
          server.accept ( new_sock );

          // rest of code -
          // read request, send reply, etc...

        }
    }
  catch ( SocketException& e )
    {
      std::cout << "Exception was caught:" << e.description() << "\nExiting.\n";
    }

  return 0;
}
```

Accepting a connection just requires a call to the **accept** method. This method accepts the connection attempt, and fills **new_sock** with the socket information about the connection. We'll see how **new_sock** is used in the next section.

3.4 Client and Server - sending and receiving data

Now that the server has accepted the client's connection request, it is time to send data back and forth over the socket connection.

An advanced feature of C++ is the ability to overload operators - or simply, to make an operator perform a certain operation. In the **ClientSocket** and **ServerSocket** classes I overloaded the << and >> operators, so that when used, they wrote data to and read data from the socket. Here is the updated version of the simple server:

listing 4 : a simple implementation of a server (simple_server_main.cpp)

```
#include "ServerSocket.h"
#include "SocketException.h"
#include <string>

int main ( int argc, int argv[] )
{
  try
```

```
    {
      // Create the socket
      ServerSocket server ( 30000 );

      while ( true )
        {

          ServerSocket new_sock;
          server.accept ( new_sock );

          try
            {
              while ( true )
                {
                  std::string data;
                  new_sock >> data;
                  new_sock << data;
                }
            }
          catch ( SocketException& ) {}

        }
    }
  catch ( SocketException& e )
    {
      std::cout << "Exception was caught:" << e.description() << "\nExiting.\n";
    }

  return 0;
}
```

The **new_sock** variable contains all of our socket information, so we use it to exchange data with the client. The line "new_sock >> data;" should be read as "read data from new_sock, and place that data in our string variable 'data'." Similarly, the next line sends the data in 'data' back through the socket to the client.

If you're paying attention, you'll notice that what we've created here is an echo server. Every piece of data that gets sent from the client to the server gets sent back to the client as is. We can write the client so that it sends a piece of data, and then prints out the server's response:

listing 5 : a simple implementation of a client (simple_client_main.cpp)

```
#include "ClientSocket.h"
#include "SocketException.h"
#include <iostream>
#include <string>

int main ( int argc, int argv[] )
{
  try
    {

      ClientSocket client_socket ( "localhost", 30000 );

      std::string reply;
```

```
      try
        {
          client_socket << "Test message.";
          client_socket >> reply;
        }
      catch ( SocketException& ) {}

      std::cout << "We received this response from the server:\n\"" << reply << "\"\n";;

    }
  catch ( SocketException& e )
    {
      std::cout << "Exception was caught:" << e.description() << "\n";
    }

  return 0;
}
```

We send the string "Test Message." to the server, read the response from the server, and print out the response to std output.

4. Compiling and Testing Our Client And Server

Now that we've gone over the basic usage of the **ClientSocket** and **ServerSocket** classes, we can build the whole project and test it.

4.1 File list

The following files make up our example:

Miscellaneous:
- Makefile - the Makefile for this project
- Socket.h, Socket.cpp - the Socket class, which implements the raw socket API calls.
- SocketException.h - the SocketException class

Server:
- simple_server_main.cpp - main file
- ServerSocket.h, ServerSocket.cpp - the ServerSocket class

Client:
- simple_client_main.cpp - main file
- ClientSocket.h, ClientSocket.cpp - the ClientSocket class

4.2 Compile and Test

Compiling is simple. First save all of the project files into one subdirectory, then type the following at your command prompt:

```
prompt$ cd directory_you_just_created
prompt$ make
```

This will compile all of the files in the project, and create the simple_server and simple_client output files. To test these two output files, run the server in one command prompt, and then run the client in another command prompt:

first prompt:

```
prompt$ ./simple_server
running....
```

second prompt:

```
prompt$ ./simple_client
We received this response from the server:
"Test message."
prompt$
```

The client will send data to the server, read the response, and print out the response to std output as shown above. You can run the client as many times as you want - the server will respond to each request.

5. Conclusion

Sockets are a simple and efficient way to send data between processes. In this article we've gone over socket communications, and developed an example server and client. You should now be able to add socket communications to your applications!

Rob Tougher

Rob is a C++ software engineer in the NYC area. When not coding on his favorite platform, you can find Rob strolling on the beach with his girlfriend, Nicole, and their dog, Halley.

Published in Issue 74 of *Linux Gazette*, January 2002

TABLE OF CONTENTS | FRONT PAGE | TALKBACK | FAQ