A simple C++ interface for TCP/IP

by

Debayan Gupta

Dec-10, 2010

Term Project for
CPSC 527a: Object Oriented Programming, Fall 2010
Yale University

Instructor: Prof. M. Fischer

Abstract:

Sockets provide a mechanism for processes to exchange data. I present a collection of classes in C++ that provide an easy and intuitive interface for TCP/IP [DC09] socket programming. These classes allow a programmer to create and use sockets for various purposes without the jargon required to do so in C [Vic98], while providing type-safety at the same time.

Unlike the many C++ libraries already available for socket programming (such as Boost.ASIO [Koh08], Practical C++ Sockets [Don10], etc.), my interface is IP version agnostic, ie., IPv4 and IPv6 addresses [Ber10] can be used interchangeably. Thus, applications that use this interface will be able to disregard any problems caused by the transition from IPv4 to IPv6. Further, this interface allows sockets to be treated like streams, and data can be input/output using the standard "<<" and ">>" operators [Cpl10].

I have also provided a number of simple classes implementing common servers and clients (web, echo, chat, etc.) which can be easily extended.

Table of Contents

1. Introduction

1.1. Purpose

Socket programming in C is very difficult, to put it lightly. There are a number of functions with very complicated calling sequences [Vic98] and no type safety to speak of. With the sheer number of "void*"s used, unexpected and non-isolatable errors can happen very easily.

A number of people have written C++ wrappers for the Berkeley Sockets (also known as the BSD Socket API [Hal09]).

One of the best libraries available for this is boost.asio [Koh08]. I had intended to use some parts of boost.asio in this project; however, upon exploring the library, I found one glaring defect – it was not IP version agnostic. Classes written for IPv4 had to have completely separate methods for IPv6. The two types of addresses had different method names (with different calling sequences). In short, any application that needed to work on IPv4 and IPv6 had to have very nearly twice the amount of code.

Therefore, I decided to write a C++ interface for TCP/IP that would be completely IP version agnostic, and then build some common applications that implemented the interface.

1.2. Methodology

I have used a number of common design patterns for this project, eg. the Address class uses the Proxy pattern [GHJV95], and provides an IP agnostic interface. Therefore, other classes can use the class without worrying about the actual address. All of the design patterns used, along with the reasons for their usage, have been described in the list of classes. Overall, however, the system uses the polymorphism and bridge patterns extensively [GHJV95], relying on the interfaces provided by the base classes to allow complex interactions between various derived types.

The prototyping process [Bro95] has been used throughout the development of this project – I developed simple systems that worked under a highly restricted set of conditions, and then recursively added complexity.

I have used a GUI-style event loop [Fis10] for the servers – the base classes (eg. ServerSocket) have virtual functions that act as placeholders for the eventual user functions (the event handlers).

2. Requirements

It is assumed that the C++ Standard Template Library [Cpl10, SL94] is available. For testing the applications created by this code, the following software should be present:

1. A web browser (www.webdevelopersnotes.com/design/browsers_list.php3 has a list of browsers).
2. A telnet client (www.telnet.org provides links to telnet clients for many platforms).

No other libraries have been used.

3. Classes

3.1. Description of overall class structure

There are four main classes: Address, Socket, ServerSocket, and ClientSocket. All the other classes are either helpers, or extensions/variations of these classes.

The Address class acts as a proxy between the other classes and the actual IPv4/IPv6 address. The Socket class uses it (via composition) to store the actual address of a socket. The Socket class further defines the input/output mechanisms for all Sockets.

ServerSocket and ClientSocket, as can be inferred from their names, are the main server and client socket base classes. ServerSocket is a controller class (handles events) that dispatches events to virtual event handler methods, while ClientSocket simply provides an interface for different kinds of clients.

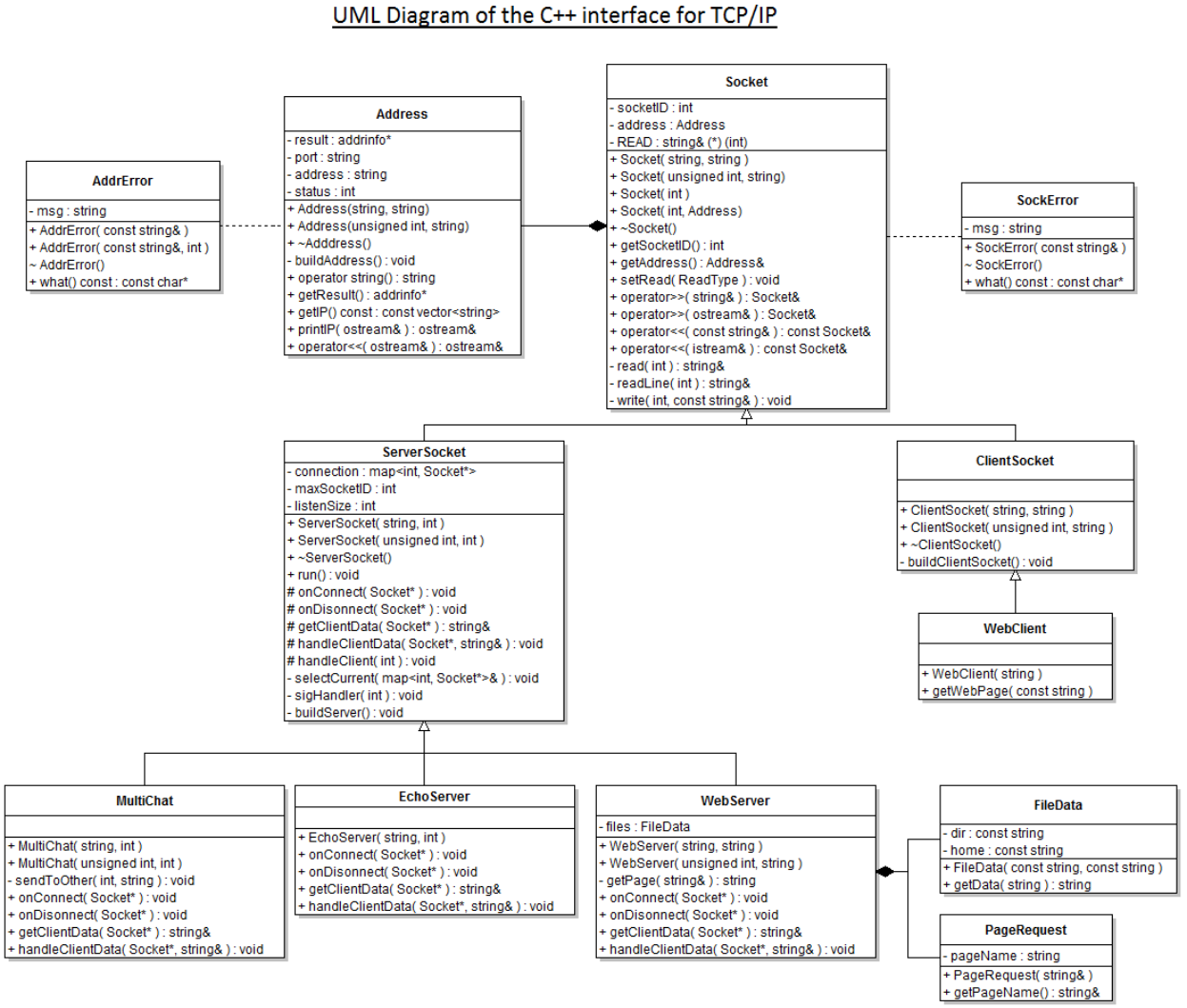

Figure 1 (Note: A better resolution is available in the accompanying file "uml.png")

3.2. List of classes

3.2.1. Address

This class acts as a separator between the actual network addresses and the other parts of the program. It implements the proxy pattern [GHJV95] to allow seamless functioning regardless of whether the underlying network uses IPv4 or IPv6.

I will now describe the various structures usually used, as well as the approach that I have taken to make my address system IP version agnostic. Information all of the structures and functions described below can be found in the linux man pages. For more information, see [Man10].

Most old applications use the following C structures:

- struct sockaddr_in
- struct sockaddr
- struct in_addr

With the advent of IPv6, the following structures came into use:

- struct sockaddr_inet6
- struct sockaddr_storage
- struct in6_addr

Unfortunately, these two sets of classes, except for "struct sockaddr_storage", were incompatible in every sense of the word (they had different variable names, types and sizes – even the constants for initializing the status variables were different). This meant that programmers had to write completely separate methods to deal with IPv4 and IPv6 addresses.

A more recent structure for IP addresses is:

```
struct addrinfo {
    int              ai_flags;     // AI_PASSIVE, AI_CANONNAME, etc.
    int              ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
    int              ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
    int              ai_protocol;  // use 0 for "any"
    size_t           ai_addrlen;   // size of ai_addr in bytes
    struct sockaddr *ai_addr;      // struct sockaddr_in or _in6
    char            *ai_canonname; // full canonical hostname

    struct addrinfo *ai_next;      // linked list, next node
};
```

I have used this structure to store addresses for two main reasons:

1. The `ai_family` field can hold `AF_INET` (for IPv4), `AF_INET6` (for IPv6), `AF_UNSPEC` (which allows the use of both kinds of addresses).

2. The structure can store multiple addresses in the form of a linked list – so, this structure will work perfectly even if something has multiple IPs (eg., in most modern computers, “localhost:80” has two IP addresses – 127.0.0.1 and ::1).

I have used the pre-defined function getaddrinfo() [Man10] in conjunction with this structure to store the appropriate IP addresses into it. The member function Address::getIP() provides a vector of strings with the IP addresses stored within it in a human readable format.

I have also defined multiple constructors to allow creation of an Address in many different ways (string, int, sockaddr_storage, etc.).

The class has “operator string()” defined, which allows it to return the entire set of IP addresses as a single string in a human readable format (it just prints to a stringstream [Cpl10] and returns the string).

3.2.2. ClientSocket

The ClientSocket class is derived from the class Socket. It creates a TCP/IP socket using the IP family detected by the Socket class, and attempts to connect to it. It tries to create & connect to all the addresses available in the addrinfo linked list.

It has a built in exception handler that handles the errors appropriately and prints messages to cerr.

3.2.3. EchoServer

The EchoServer class is derived from the class ServerSocket. It creates a server on the specified port on the current computer (*localhost*).

The server can handle multiple clients simultaneously, and simply echoes back whatever a client sends.

In the current implementation, I have called “setRead(Line)” so that it reads (and echoes) only after a client presses “return”.

To test the server, one can simply telnet (see requirements) into it.

3.2.4. FileData

This class is used by the WebServer class to serve up data from local files. In order to function, it needs a target directory (be default, the current directory) where it searches for the files requested by the client.

I have also added a default “homepage” filename – “index.html”, but this only works for the top level directory.

3.2.5. MultiChat

The MultiChat class is derived from the class ServerSocket, and creates a chat server capable of catering to multiple clients simultaneously.

Anything typed by any one of the clients will appear on the screens of all the others. Clients can also see when other clients log in/out.

3.2.6. PageRequest

The PageRequest class is used to parse inputs from clients, specifically web browsers. The request is parsed, and the name of the file being requested is stored in a private variable, which can be retrieved by using an access function, getPageName().

Detailed information on the structure of the data sent by a web browser requesting a page can be found at [Fie99].

3.2.7. ServerSocket

This class is derived from Socket. It implements the "controller" design pattern [GHJV95].

It uses a map of pair<int, Socket*> (available from the Standard Template Library [SL94]) to keep track of all the connections.

When constructed, the class immediately creates and binds to a socket (appropriate error handling mechanisms have been implemented). It is also capable of reusing sockets.

A common problem with many servers is the creation of "zombie" processes. These may appear when fork()ed processes terminate – they are simply processes that have terminated, but still have an entry in the process table. The UNIX faq (http://www.faqs.org/faqs/unix-faq/faq/part3/section-13.html) has detailed information on this subject.

Even though the current implementation does not use fork()ing, it sets up a mechanism for "reaping zombies", or removing these unnecessary entries from the process table, using the sigaction structure [Ope97].

The class provides a number of virtual functions for derived classes to modify. These fall into two main groups: pure virtual functions that must be defined in derived classes, and virtual functions that will work for most purposes, but may be redefined in derived classes in some special cases.

The run() function is the core of the server. It uses a GUI-style event loop to run through all the clients currently connected to it, and when some event occurs, dispatches it to an appropriate event handler (these are the pure virtual methods).

The current implementation uses the select() function to do synchronous I/O multiplexing (more details on the function are available in the linux man pages [Man10]).

This, while consuming far less CPU time than using nonblocking sockets [DC09], is still not optimal. Libraries such as libevent [Pro04] provide far better performance, but given the restrictions on time, I decided to go with the (comparitively) simpler select() function, which looks like this:

```
int select(int numfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
struct timeval *timeout);
```

Because select() requires file descriptor sets (type fd_set [Man10]), I have written a function that acts as an interface, allowing the use of map<int, Socket*> for the same purpose.

The class provides 4 pure virtual functions which are the primary event handlers:

- onConnect
- onDisconnect
- getClientData
- handleClientData

Examples of their usage are provided in some of the classes, as well as in the usage and examples section of this document.

3.2.8. Socket

The Socket class creates a generic, IP agnostic socket. It has a data member *address* of type Address that it uses to do this.

The class also sets up the << and >> operators for use with Socket variables, allowing direct I/O with strings and other streams.

In addition to this, it has a number of constructors for initializing sockets in various ways, as well as access functions for returning the socket number and the address.

3.2.9. WebClient

The WebClient class is a very simple class, derived from ClientSocket, that allows access to web pages. It assumes the port number to be 80/http, and sends the appropriate header and page request [Fie99] to a web server.

3.2.10. WebServer

This class is derived from the class ServerSocket. It creates a simple web server that can serve up pages/files from a given directory.

Upon receiving a request from a client, it uses the PageRequest class to parse the data and get the name of the file to be served up.

It then sends this filename to the member variable *files* (of type FileData) that searches for the file in the specified directory, and sends back the contents of the file. If the file is not found, it returns a 404 error [Fie99].

3.3. Classes for handling exceptions

3.3.1. AddrError

The AddrError class is derived from std::exception [Cpl10, SL94], and can handle errors caused by malformed addresses, unrecognized IP families, etc.

3.3.2. SockError

This class is also derived from std::exception, and handles errors caused by the Socket class.

4. Usage and Examples

In this section, I have outlined some basic usages of the interface. This list is by no means comprehensive, and is simply meant to demonstrate some of the capabilities of the interface.

Getting IP addresses

IP addresses can be found by using the Address class.

Table 1: Addresses

Code	Output
`Address a("http", "google.com");` `cout << a;`	`[173.194.33.104]`
`Address a("http", "localhost");` `cout << a;`	`[::1]  [127.0.0.1]`
`Address a("http", "ipv6.google.com");` `cout << a;`	`[2001:4860:800f::63]`

Of course, instead of "http", one can write the port number as 80 or "80".

Getting data from the internet

This can be done using the ClientSocket class. The WebClient class provides a pre-built mechanism for viewing web pages.

Table 2: Clients

Code	Output

`ClientSocket c("http", "google.com");`
`c << "GET / HTTP/1.0\n";`
`c << "Host: www.google.com\n\n";`
`c >> cout;`
	The contents of the google.com home page.

`WebClient w("google.com");` `cout << w.getWebPage("imghp");`	***The contents of the google images home page.***

Using the WebClient class is a lot easier, especially when requesting specific pages, because one can simply give the name of the page to getWebPage(), and it forms and sends the appropriate request, and returns the server's response.

Creating a Web Server

The WebServer class provides a simple way to create webservers:

```
// create a web server at port 5000
// documents to be served up from the directory "www"
WebServer w("5000", "www");

// run the server
w.run();
```

(I have provided a directory "www" with some sample web pages, as well as a photograph)

I ran this on the lion computer in the zoo (more information available at http://zoo.cs.yale.edu/) – to access the "website", all one had to do was type `lion.zoo.cs.yale.edu:5000` into their web browser, and the "index.html" page in the "www" directory was served up.

Note: Other kinds of servers, such as EchoServer and MultiChat can be created using essentially the same code, just replacing "WebServer" with the appropriate class name. Of course, in both of these cases, the directory name ("www" in the example above) will not be required. These can be easily tested with ClientSocket, or by using telnet (see requirements).

5. Limitations

Exception Handling

Exception handling in the current implementation is done at an extremely low level, for the most part. Some rudimentary exception types have been defined and used, but there is a lot of scope for improvement.

I have created two basic exception types AddrError and SockError, which should act as base classes for a more varied set of exceptions in later implementations.

Encoded URLs

When implementing the WebServer class, I had to parse the page request sent by the client's browser. This is done by the class PageRequest. Currently, the PageRequest class is unable to handle most encoded URLs [BFM05, BMM94], that is, URLs containing certain special characters [BMM94].

In the current implementation, only "%20", which represents a single space, is recognized by PageRequest. In future expansions, it should be capable of recognizing common URL encodings [Wil05], if not the entire set.

Efficiency

The program currently sends all data by encoding it into strings. This, while comparatively safe from endian issues, is horribly inefficient from the network traffic point of view [Hal09].

Safety Issues

When creating a server of any kind, a number of safety issues [Shi03] need to be taken into consideration. While some of these issues have been taken care of in the current implementation, there are many more that need to be addressed before this interface can be used for any practical purpose.

Syntactic sugar for URLs

While this may not seem to be very important, almost nobody types in something like "www.debayangupta.com/index.htm" – they will expect to be redirected to the correct page (index.htm) automatically. These "shortcuts" are ubiquitous, and any practical application needs to be able to recognize them. Apache has a very powerful module for doing this [Apa10]. Later versions of this interface should be able to handle these kinds of abbreviated requests.

6. Bibliography

[Apa10] The Apache Software Foundation. *Apache Module mod_rewrite*, 2010. "Provides a rule-based rewriting engine to rewrite requested URLs on the fly". http://httpd.apache.org/docs/2.0/mod/mod_rewrite.html

[Ber10] D. J. Bernstein. *The IPv6 mess, djbdns*, cr.yp.to, 2010. http://cr.yp.to/djbdns/ipv6mess.html

[BFM05] T. Berners-Lee, R. Fielding, L. Masinter. *RFC 3986: URI Generic Syntax*, The Internet Society, 2005. http://tools.ietf.org/html/rfc3986#section-2.1

[BMM94] T. Berners-Lee, L. Masinter, M. McCahill. *RFC 1738: Uniform Resource Locators (URL)*, the Internet Engineering Task Force, 1994. http://www.ietf.org/rfc/rfc1738.txt

[Bro95] Frederick P. Brooks Jr. *The Mythical Man-Month, Essays on Software Engineering, Anniversary Edition*, Addison-Wesley Professional. ISBN-13: 978-0-201-83595-3.

[Cpl10] CPlusPlus.com. *Library Reference*, The C++ Resources Network, 2010. http://www.cplusplus.com/reference/

[DC09] Michael J. Donahoo, Kenneth L. Calvert. *TCP/IP Sockets in C: Practical Guide for Programmers*, Morgan Kaufmann, 2009. ISBN-13: 9780123745408

[Don10] Michael J. Donahoo. *Practical C++ Sockets* provides wrapper classes for a subset of the Berkeley C Socket API for TCP and UDP sockets. http://cs.baylor.edu/~donahoo/practical/CSockets/practical/

[Fie99] R. Fielding. *RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1*, The Internet Society, 1999. http://www.w3.org/Protocols/rfc2616/rfc2616.html

[Fis10] Michael J. Fischer. *Lecture 22, CPSC 427a: Object-Oriented Programming*, Yale University, Nov-30, 2010. http://zoo.cs.yale.edu/classes/cs427/2010a/attach/ln22.html

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN 0-201-63361-2.

[Hal09] Brian Hall. *Beej's Guide to Network Programming*, Version 3.0.14, 2009. http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html#serialization

[Koh08] Christopher M. Kohlhoff. *Boost.Asio* is a cross-platform C++ library for network and low-level I/O programming that provides developers with a consistent asynchronous model using a modern C++ approach. http://www.boost.org/doc/libs/1_39_0/doc/html/boost_asio.html

[Man10] ManPagez.com. *BSD Library Functions Manual*, manpagez, 2010. http://www.manpagez.com/man/3/

[Ope97] The Open Group. *The Single UNIX ® Specification, Version 2*, The Open Group, 1997. http://www.opengroup.org/onlinepubs/007908799/xsh/sigaction.html

[Pro04] Niels Provos. *libevent - an event notification library*, rc2.0.9, 2004. http://monkey.org/~provos/libevent/

[Shi03] Thomas W. Shinder. *Best Damn Firewall Book Period*, Syngress Publishing, 2003. ISBN-13: 9781931836906

[SL94] Alexander Stepanov and Meng Lee, *The Standard Template Library*. HP Laboratories Technical Report 95-11(R.1), November 14, 1995.

[Vic98] Vic Metcalfe. *Programming UNIX Sockets in C - Frequently Asked Questions* created by Vic Metcalfe, Andrew Gierth and other contributors. http://www.faqs.org/faqs/unix-faq/socket/

[Wil05] Brian Wilson. *Index DOT Html – URL Encoding*, 2005. http://www.blooberry.com/indexdot/html/topics/urlencoding.htm