COMPUTER SCIENCE & ENGINEERING DEPARTMENT
TEXAS A&M UNIVERSITY

DESIGN DOCUMENT
ON

SOCKET++

A Simple Socket Library in C++

SUPERVISED BY:

Professor Bjarne Stroustrup

Submitted by:

Ankit Gupta Garima Agarwal Swati Singh

TABLE OF CONTENTS

Socket++
A Simple Socket Library in C++

Design Documentation

Ankit Gupta, Garima Agarwal, Swati Singh

1. Introduction:

Sockets are an application programming interface for interprocess communication, either locally or across a network. Socket application program interfaces (APIs) are the network standard for TCP/IP and are supported by a wide range of operating systems. TCP/IP provides an interface for full duplex communication through socket programming which was first introduced with BSD version Unix operating system. The socket programming interface provides three basic communication services:

- TCP Stream Communication
- UDP Datagram Communication
- raw datagram communication to IP layer

TCP basically follows client-server model in which a master process listens for connection requests from clients. After a client is connected, the master process creates a child process to handle the data request from the client. Multiple clients can connect to a server at a time. A Server can be synchronous or an asynchronous server.

But Socket programming in C is very difficult, to put it lightly. There are a number of functions with very complicated calling sequences 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). One of the best libraries available for this is boost.asio. We had intended to use some parts of boost.asio in this project; however, upon exploring the library, we 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. Moreover learning curve is too high and it would take 2 days to just learn how to write a simple server.
Therefore, we 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 in easy and

user friendly manner such that user has not to worry about handling errors, multithreading, type safety, event handling and all.
We have used a number of common design patterns for this project, eg. the Address class uses the Proxy pattern and provides an IP agnostic interface. Therefore, other classes can use the class without worrying about the actual address. Similarly, Reactor Pattern is used for event handling.

However socket programming in C is very complicated with various function calls and various void pointers used in so many function calls. It does not provide type safety and unexpected errors can occur easily which are very hard to debug.

2. Existing Libraries:

As difficult it is to work with sockets in C, so is it to use the existing libraries in C++ for socket programming. While some of the libraries make the users suffer due to bad design, others are just overly complicated. Some the existing C++ libraries are listed below:

- ADAPTIVE Communication Environment (ACE)
- Boost
- Netlink Socket C++
- Practical C++ Sockets
- Giallo: C++ Network Library

2.1 Issues with Existing Libraries:

- **Boost:** Boost ASIO manages to get most of the things on the right track, especially the Asynchronous Input/Output part. However there are several design issues with the network part of the library. And most of all, it is overly complicated and lacks sufficient documentation. A new user may take days to understand how the library is supposed to be used.

- Non-copyable Sockets: A user may want to copy a socket or use it in multiple threads. The way to do this is to transfer a file descriptor to another thread or by storing a pointer to the socket in a shared_ptr<> which means dynamic allocation is required to use a socket in multiple threads. A socket should at least be movable (which can be done using the move constructors in C++1x).

- Bad Interface Design: Both synchronous and asynchronous behaviours have been clubbed into the same interface which is a terrible design. A user would either perform a synchronous operation over a socket or an asynchronous one, not both.

Two different behaviors in the same interface is a clear violation of the single responsibility principle

- Name reuse: ASIO reuses the standard names used in C library to do different tasks. For example, the acceptor class defines a function accept() to accept stream sockets.

- Not IP-version agnostic: Different methods have been written for IPv4 and IPv6 addresses and hence any application that uses both these addresses uses twice the amount of code.

3. Design of Socket++ Library

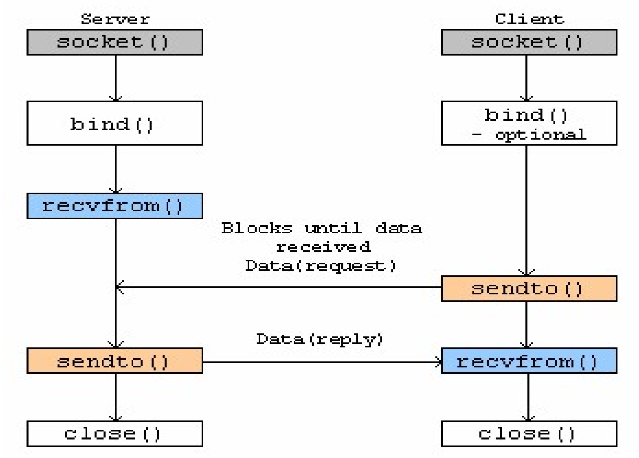
1) Separate interface for address and socket:

It is important to differentiate an address and a socket as two separate entities instead of binding them as one. A socket address is the combination of an IP address and a port number. While a TCP server always binds a socket to some address, this binding of address may not be a must precondition for all the sockets. For example, binding a client socket to an address is optional in case of a UDP session. Hence, a socket address is represented by a Address class and the socket is not inherently bound to any socket address.

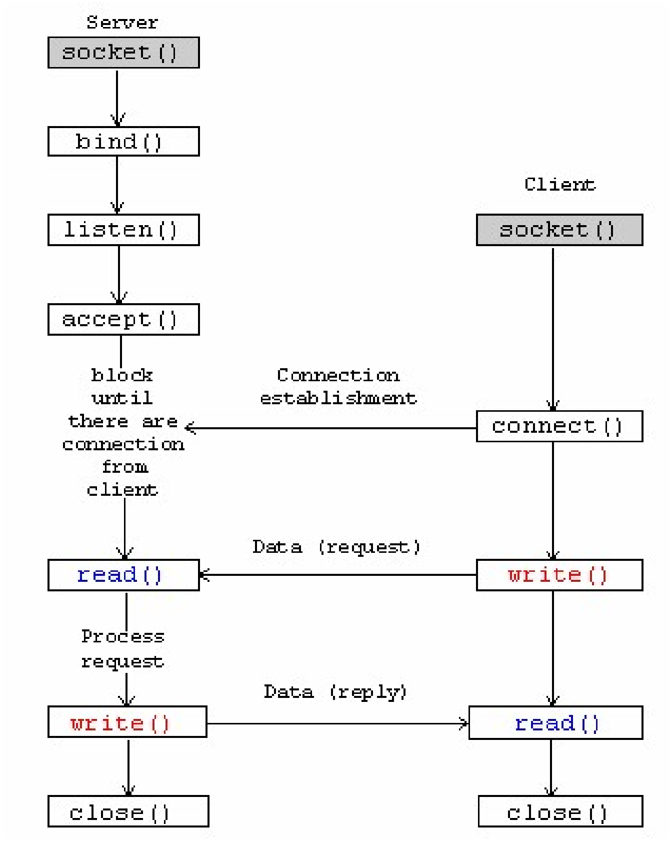
.

2) Separate the interface of TCP and UDP sockets

The following diagram illustrates a client/server relationship of the socket APIs for TCP.



The following diagram illustrates a client/server relationship of the socket APIs for TCP:



As can be seen from the flowcharts, the functions of TCP and UDP sockets are different. A connection needs to be established in a TCP session before any data transfer can occur. This requires the client socket to send a connect request to the server through a connect () function call and the server to accept the request through a accept() function call. This establishment of connection is not a requirement for UDP sessions in which a client socket can direct sends a request to the server. Providing a common interface for both TCP and UDP sockets would imply providing UDP sockets the flexibility to implement connection establishment which is not at all an acceptable design.

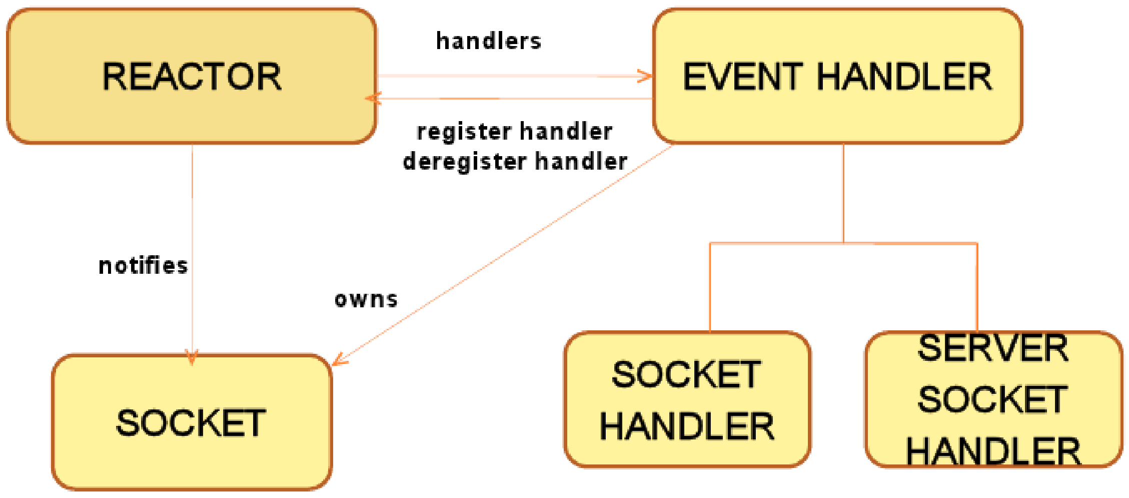
3) Separate Interface for TCP Client and TCP Server

A basic property of a working TCP client is that it connects to a server. Hence before any data transfer can occur over a client socket, it must be in connected state. On the other hand, a working TCP server must be in listening state and should block until it receives a connection request from a TCP client. Since both Server and Client socket have different function calls and implementations, hence again it is viable to provide different classes for TCP Client Socket and TCP Server Socket.

4) Event Handler Interface separate from the Socket Interface

All event handling is done in a separate Event Handler class. The user will extend this event handler class to redefine his own functions for handling read and write on a socket. This follows from the design principle of Separation of Concerns and Single Responsibility principle.

5) Reactor Pattern for Asynchronous Event Handling:



Whenever a socket is created, it associates itself with a event handler which defines what actions should be taken for a read and write event. The event handler class in turn registers itself with the Reactor which contains the select loop to wait for events on all the sockets registered with it. Any event on a socket triggers the reactor to notify the respective event handler which then takes the respective actions as defined as per the handler methods.
Since a data socket and a control socket for a server handles read event in a different way, hence they should have separate event handlers. Thus Socket Handler and Server Socket Handler class inherit the Event Handler Class. For example, a read event on a server socket denotes incoming connection request from a client whereas on a general socket would denote a request for a data stream t. A server socket would handle a read event by accepting the connection and creating a data socket for the particular connection. This data socket would again be associated with another event handler which would be maintained in the server socket's event handler class in a map structure, and will also be registered with the Reactor. On the other hand, a data socket would respond to a read request by serving the client with the requested data stream.

6) Invariants:

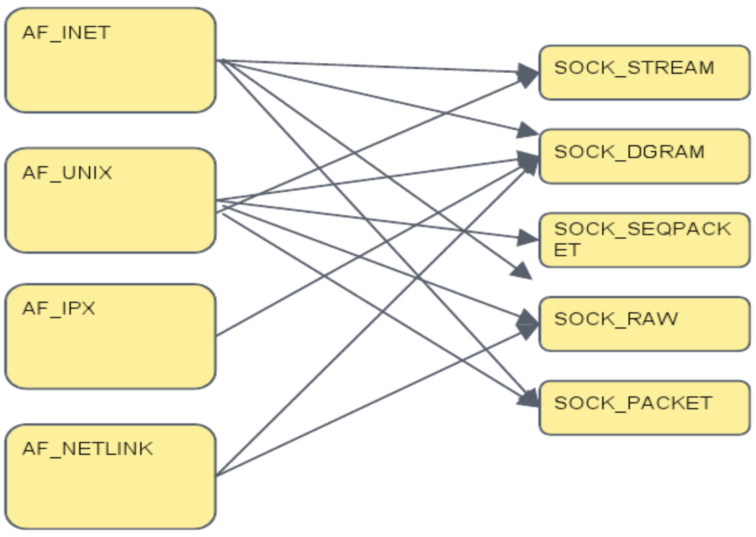
a) Allow Creation of Sockets with Valid Type-Domain Combination:

A socket () function uses Socket Type, Socket Domain and Protocol as its parameters while socket creation. However, not combination of socket type and socket domain is valid.

The following are the socket types and socket domain provided by Linux Socket API:

SOCKET TYPE	**SOCKET DOMAIN**
AF_UNIX	SOCK_STREAM
AF_INET	SOCK_DGRAM
AF_INET6	SOCK_SEQPACKET
AF_IPX	SOCK_RAW
AF_NETLINK	SOCK_RDM
AF_X25	SOCK_PACKET
AF_AX25	
AF_ATMPVC	
AF_APPLETALK	
AF_PACKET	

The figure given below represents a subset of valid combinations that are possible with protocol number as 0.



While, a user may make an error by using invalid combinations, the TCP Socket class creates a socket of either AF_INET and SOCK_STREAM which is a valid combination.

b) TCP Client Socket must be in Connected State:

The constructor for TCP client socket would ensure that client is in connected state. The connect() call immediately accompanies the socket() function call. A TCP Client socket is of no use until it connects to a server.

c) TCP Server Socket must be in Listening State:

The constructor of TCP server socket would ensure that the socket is in listening state. A TCP server must enter into listening state as soon as it is created and binds to a specific IP address and port number.

d) A socket is closed when it goes out of scope:

The destructor ensures this by closing the socket, so that there aren't any dangling sockets in the network.

4. Alternative Designs Considered:

1) Integrate the interface of Client Socket and Server Socket into a single Socket class. The idea was to implement socket as a separate independent object rather than associating it with a client or server session. However with this design, there were faults with resource acquisition while the constructor is called to create the socket object. For example, while a TCP session is established, ideally, the client socket must be connected to a server socket while the server socket must be listening to a specific port number to accept connections. On the other hand, in a UDP session, the client socket need not connect to the server socket before a data transfer could occur. In order to implement all these concepts in practice, we decided to separate the interfaces for TCP Server Socket, TCP Client Socket and UDP Socket.

2) Integrate the Address and Socket Class as the TCP socket typically binds to an IP address and a port number. However, this is not true for UDP sockets in which the socket sends data to addresses directly without binding to it

5. Exception Handling

Two Exception classes are used in our implementation for handling different types of error and handling them.

addr_error

This class is derived from std::exception, and can handle errors caused by malformed addresses, unrecognized IP families, etc.

sock_error

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

- If socket can't be created
- The port to which socket try to listen is busy.
- If other end closes the connection.
- If client is unable to connect to the server.

6 .UNIT TESTING:

- Created a server socket in setup, close it in the tear down and checked whether that all the resources acquired by it are freed or not.
- Set timeouts on sockets. If test fails for some unexpected reason, we don't let it hanging waiting for a connection forever.
- We need to be able to interleave the client/server behavior. We tested what happens if the server shuts down its side of the connection half way through a client read/write and throw proper exception.
- Disconnecting the network wire from the back of computer and ensure that proper error should be thrown.
- Using ipconfig /releases the port which are already bind. Server should throw an error stating proper error.
- If a client uses the same address which is already connected, we are rejection the new connection.
- Check there should not be any open socket after gracefully destroy the socket object.
- Socket connection should be closed whenever server/client reads < 1 bytes from the other.
- Server should not accept client when it reaches it max_listen value.

7. C++11 Features in Socket++

- ### *Move Semantics*

The idea behind to use a move assignment is that instead of making a copy of socket object, we want to use the same existing socket, so move helps us simply to takes the representation from its source and replaces it. There are classes and functions that are taking argument as socket, but we don't want to create a new socket as there are mainly two reasons for not doing that:

1. Copying a socket leaves two socket open for the same socket fd and then create a inconsistency in the system.
2. Whenever socket got created, while copying temporary socket object will be created and destroyed and if there is some connection already on that socket, would get break.
   e.g.

client_socket_handler<T> (std::move(sock));

client_handler(std::move(client_sock));

sock_fd = std::move(sock);

- ### *UNIQUE_PTR*

Concept of unique ptr really helps us a lot and allows us not to give responsibility of freeing the resources by the user. , if an exception is thrown, the unique_ptr will (implicitly) destroy the object pointed to and user is not supposed to free it.

Moreover it also has move semantics and can act as a pointer container so we don't really had to take care of multiple sockets for the same value as before.

e.g.

client_socket_handler(unique_ptr<client_socket<T>> sock)

unique_ptr<Socket<T>> accept_conn(unique_ptr<Socket<T>> newSock);

- ### *Constexpr:*

It helps in providing a way to guarantee that an initialization is done at compile time for userdefined type. We have used it while template conversion.

e.g

template <typename T, typename U>

constexpr bool Convertible() { return std::is_convertible<T, U>::value; }

- ***USING IN TEMPLATES***

In our project socket, client_socket, server socket, address is taking template parameters and can be of any type depending upon the class family , socket domain. So We have use the new typedef of C++11 called using for defining template type parameters.

For creating INET STREAM(TCP) address socket make its typedef and use the same throughout our code.

e.g

using client_sock_stream = client_socket<inet_stream_addr>;

using sock_stream = Socket<inet_stream_addr>;

- ***Enable_if:***

Since as we have shown in the table above , there are number of Address family type and socket domains correspondingly but all of the possible combinations are not allowed with each other. So we use enable_if., This enables a function only if a condition (passed as a type trait) is true; otherwise, the function is undefined. In this way, you can declare several "overloads" of a socket template and enable or disable them depending on some requirements we need. The nice part of this is that the disabled functions will not be part of our binary code because the compiler simply will ignore them.

e.g *template <class T>*

void Socket<T,typenamestd::enable_if<Convertible<T, base_addr*>()>::type>::set_reuse_address(bool flag)*

8. DESCRIPTION OF OVERALL CLASS STRUCTURE

1) Class: base_addr

Header:base_address.hpp

Description:
It is the common base class for other address classes like inet_stream_addr(for creating TCP/IP sockets).

Types:
1) using sock_stream = Socket<inet_stream_addr>;
It provides an interface to TCP sockets which are created by binding the socket to an inet_stream_addr during initialization.

Constructors:
1) base_addr(int type);
Creates a address of a particular type, where type can be AF_INET, etc.

Member Functions:
1) int get_type(void) const;
Function to get the type of address

2) void set_type(int type);
Function to set the type of address

3) int get_size(void) const;
Function to get the size of address

2) Class: inet_stream_addr

Header:base_address.hpp

Description:
This class represents an IP socket address and derives from base_addr. The address can belong either to the IPv4 or the IPv6 address family. It consists of a host address and a port number.

Constructors:
1) inet_stream_addr(string address, int port);
Creates a inet_stream_addr from an IP address/host name and a port number/service name.

Member Functions:Member Functions:
1) addrinfo* get_result();
Returns pointer to addrinfo structure which stores the socket address.

2) void build_address();
Builds an address by providing details about the family and socket type.

3) Class: Socket

Header:socket.hpp

Description:
Socket is the base class for creating a socket from which Client Socket and ServerSocket are derived.

Constructors:
1) Socket()
Creates an uninitialized socket.

2) Socket(int fd)
Creates a socket with fd as the file descriptor id of the socket.

Destructors:
~Socket()
Closes the socket if it is open.

Member Functions:

1) string& read_bytes();
Reads bytes received at the socket
2) int get_sockfd();
Returns the file descriptor of the socket

3) int write_bytes(const string& data);
Writes a string of data to the socket.

4) void set_non_blocking(bool flag);
Sets socket to non-blocking mode

5) void set_keep_alive(bool flag);
Sets the value of SO_KEEPALIVE flag to TRUE which causes a default packet to be sent to the remote host on the other end if no packet has been sent for a long time.

6) void get_keep_alive();
Returns the value of SO_KEEPALIVE flag

7) void set_no_delay(bool flag);
Sets the value of TCP_NODELAY flag to TRUE. Applications that require lower latency on every packet sent should be run on sockets withTCP_NODELAY enabled.c

8) void set_reuse_address(bool flag);
Sets the value of SO_REUSEADDR option to TRUE. Enabling this flag indicates that the rules used in validating addresses while binding a socket to an address should allow reuse of local addresses. For AF_INET sockets this means that a socket may bind, except when there is an active listening socket bound to the address.

9) void set_broadcast(bool flag);
Sets the broadcast SO_BROADCAST flag. When enabled, datagram sockets receive packets sent to a broadcast address and they are allowed to send packets to a broadcast address. This option has no effect on stream-oriented sockets.

10) void socket_close();
Closes the socket.

3) Class: server_socket

Header: server_socket.hpp

Description:
It provides an interface to server sockets and derives from the base class Socket.

Types:
1) using server_sock_stream = server_socket<inet_stream_addr>;
It provide an interface to TCP server sockets.

Constructors:
1) server_socket(unsigned int listen_port,string ipaddress, int max_listen);
Creates a server socket that listens on a specific port and ip address. max_listen is the maximum number of clients that can attempt to connect to it at a particular time.

2) server_socket(string listen_port,string ipaddress, int max_listen);
Creates a server socket. Accepts the port number it listens to in string format.

Destructors:
~server_socket();
Destroys the server socket.

Member Functions:
1) accept_conn();
Accepts a connection from the client and returns a socket for data transfer.

5) Class: ClientSocket

Header: client_socket.hpp

Description:
This class provides an interface to socket used by a client session and it also derives from the base class Socket.

Constructors:
1) client_socket(unsigned int server_port, string ipaddress);
Creates a client socket that is connected to a specific port number and ip address.

2) client_socket(string server_port, string ipaddress);
Creates a client socket. Accepts the port number it listens to in string format.

Destructors:
~client_socket();
Destroys the client socket.

Member Functions:
1) void client_socket::build_client_socket()
Creates a socket and connects it to the IP address and port number the server is listening to. If it is unable to connect, it throws an error.

6) Class: addr_error

Header: addr_error.hpp

Description:
This class can handle exceptions caused due to a malformed addresses, invalid IP families etc.

Constructors:
1) addr_error(const string& _msg)

2) addr_error(const string& _msg, int family_type)

Destructors:
~addr_error();

7) Class: sock_error

Package: CplusSockets
Header: sock_error.hpp

Description:
This class can handle exceptions caused by the Socket class.

Constructors:
1) sock_error(const string& _msg);

Destructors:
~sock_error();

8) Class: ev_handler

Header: event_handler.hpp

Description:
It acts a base class that provides virtual functions to handle read, write and timeout events.

Constructors:
ev_handler();

Member Functions:
1) virtual int handle_read();
2) virtual int handle_write();
3) virtual int handle_timeout();

Destructor:
~ev_handler();

9) Class: socket_handler

Header: event_handler.hpp

Description:

It provides event handling methods for the server socket and derives from the ev_handler class.

Constructors:

socket_handler(unique_ptr<Socket<T>> sock)

Member Functions:

1) int handle_read():

It overrides the handle_read() of the base class and handles read event on the socket by accepting the connection request from the client. It opens a new data socket for handling sream requests from the client socket.

10) Class: client_socket_handler

Header: client_event_handler.hpp

Description:

It provides event handling methods for the client socket and derives from the ev_handler class.

Constructors:

client_socket_handler(unique_ptr<client_socket<T>> sock)

Member Functions:

1) 1) int handle_read():

It overrides the handle_read() of the base class and handles read event on the socket. It reads the data stream received on the socket.

11) Class: Reactor

Header: reactor.hpp

Description:

It is a singleton class which registers all the event handlers of all the existing sockets and waits for event on any socket through a single event loop using select.

Constructors:

Reactor();

Member Functions:

1) bool register_handler(Events ev, int fd, ev_handler *evH);

It registers a event handler object ev_handler of a socket with file descriptor fd whenever the particular socket is created.

.

2) bool deregister_handler(Events ev, int fd);

It de-registers a event handler with which a scoket with file descriptor fd is associated.

3) bool register_timeout_handler(ev_handler *evH);

//run it in a different thread since it is blocking

void Run();

static Reactor *get_instance();

12) Class: async_server

Header: server.hpp

Description:

It provides a interface for creating a simple TCP Asynchronous Server.

Constructors:

1) async_server(int listenPort);

The server is initialized by forcing it to listen to a specific port number given by the integer listenPort.

13) Class: async_client

Header: client.hpp

Description:

It provides a interface for creating a simple TCP Asynchronous Client.

Constructors:

1) async_client(unsigned int port, string address);

The client is initialized by connecting it to a server. The IP address and port number of the server must be specified as the arguments.

Member Functions:

1) write(string data);

Send data or request in the form of string to the server.

9. FUTURE WORK:

- **SOCKET++ V 1.2**
  - Support for UDP
  - Synchronous Multi-threaded Client and Server
  - Documentation using Doxygen
- Extensive Testing using http://sockettest.sourceforge.net/
- Load Testing

10. FURTHER READING:

Code repository –

- https://github.com/ankitgupta29/socket

11 ACKNOWLEDGEMENT:

- Professor Bjarne Stroustrup, CSE Department ,Texas A&M University for Creating and teaching us Wonderful Language C++.
- Andrew Nathan Sutton, Postdoctoral Researcher, Texas A&M University for helping us in various design decisions.
- www.stroustrup.com
- www.stackoverflow.com
- www.wikipedia.com
- Beej guide: http://beej.us/guide/bgnet/
- Boost asio :http://www.boost.org/doc/libs/1_52_0/doc/html/boost_asio.html
- Ace Socket library: http://www.cs.wustl.edu/~schmidt/ACE-overview.html
- Poco C++ Library: http://pocoproject.org/