Sockets++: Safe and Secure Network Programming

Undergraduate Honors Thesis

December 5, 2006

By Ehren A. Kret
Department of Computer Sciences
The University of Texas at Austin
kret@mail.utexas.edu

Supervising Professor: Dr. Greg Lavender

Table of Contents

Abstract

This research provides a more reliable and secure methodology and programming environment for developing network-enabled applications. The goal was achieved by designing and developing an Object-Oriented framework and APIs using C++. The contributions of this thesis are:

1. A new open source ANSI C++ standard Sockets++ *iostreams* framework and implementation that ensures that all network communication is not subject to common buffer overflow attacks. The framework utilizes robust *iostreams* buffering abstractions, consistency checking, and is compatible with OpenSSL, POSIX threads, and the C++ Standard Template Library.
2. A new open source ANSI C++ framework and implementation of the Tor onion-routing client that uses encryption and Sockets++ to enable application-level privacy and enhanced security when used with Sockets++.
3. A simple Web Proxy Server that is built from the previous components that enables additional privacy and anonymity when configured for use with standard web browsers. In particular, the proxy protects against snooping of a user's settings, etc.

1. Introduction

This research provides a more reliable and secure methodology and programming environment for developing network-enabled applications. The goal was achieved by designing and developing an Object-Oriented framework and APIs using ANSI C++ [5]. The contributions of this thesis are:

1. A new open source ANSI C++ standard Object-Oriented Sockets++ *iostreams* framework and implementation that ensures that all network communication is not subject to common buffer overflow attacks. The framework utilizes robust *iostreams* buffering abstractions, consistency checking, and is compatible with OpenSSL [7], POSIX threads [6], and the C++ Standard Template Library.
2. A new open source ANSI C++ framework and implementation of the Tor onion-routing client that uses encryption and Sockets++ to enable application-level privacy and enhanced security when used with Sockets++.
3. A simple Web Proxy Server that is built from the previous components that enables additional privacy and anonymity when configured for use with standard web browsers. In particular, the proxy protects against snooping of a user's settings, etc.

Figure 1 illustrates the connection between all three of these contributions.

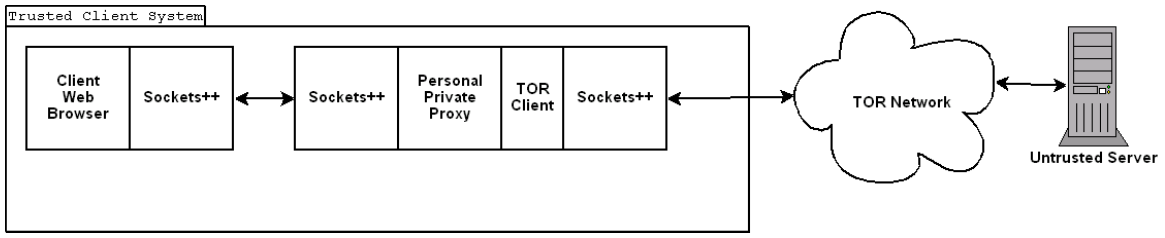


Figure 1: Anonymizing Proxy High-Level Overview

Many technologies exist to address issues related to creating safe and secure networking. Some such applications include IPSec [4] and Virtual Private Networking (VPN) using Layer 2 Tunneling Protocol (LT2P) [8] network services. They provide security at the OSI network and data link layers between a trusted client and a trusted server through an un-trusted local or wide-area network. Most modern operating systems provide built-in security mechanisms to protect against spoofing and other common TCP/IP attacks. However, exploits still occur, usually due to programmer sloppiness and error (e.g., stack smashing). This type of exploit usually results in privilege level escalation by compromising the vulnerable program’s stack and allowing execution of code introduced to the system from the network. These common problems can be overcome by providing safe and secure programming abstractions and APIs in the session layer and/or higher layers, but few exist. The standard way for programmers to utilize the transport layer is through raw memory buffers, which is the source of many of the network software exploits. By providing a more robust programming interface to the transport layer, many of the vulnerabilities can be eliminated. Our approach is Sockets++, a library that provides a safe “socket stream” interface to the transport layer. The library can be reused to provide presentation and application layer protection for network-enabled software applications.

2. Design

The first part of the solution is to create a portable, secure, thread-safe library that transforms the interface provided by the operating system to the transport layer into an easier to use and more intuitive C++ *iostreams* compatible abstraction and interface that complies with the latest C++ standards and co-exists with the Standard Template Library and POSIX threads. This library is called Sockets++ and allows networked application input and output to be programmed using the same methods as reading and writing to a file or an I/O device. By providing safe buffered streams through the library that only allow indirect access to the buffers through the well-understood *iostreams* interface, more secure programs can be written. In addition, the library utilizes the OpenSSL open source security library to provide SSL support to applications. SSL support provides the applications with confidentiality through end-to-end encryption and authentication using digital certificates. Using the Sockets++ library, programs can be securely written, safe from buffer overflow and other stack-smashing attacks, and have a confidential stream from the client to the authenticated server system.

2.1. Sockets++ Core Design

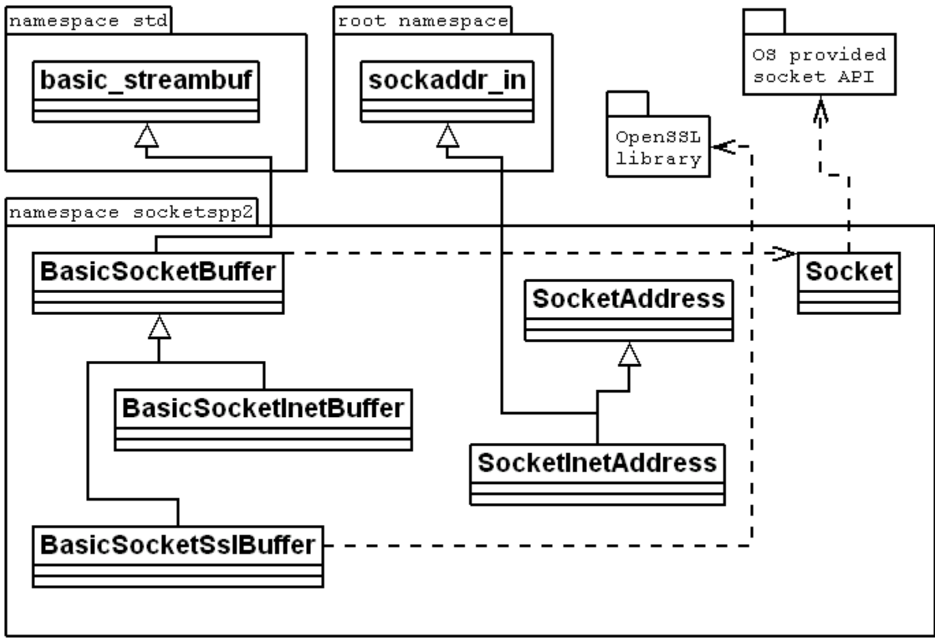


Figure 2: Sockets++ Core Library Design

The first portion of design involved creating an abstraction for everything specific to the operating system in order to isolate the later parts of the design from having to worry about operating system specific differences. The first abstraction that was designed represents socket addresses. Socket addresses represent where the socket can bind to or connect to. For example, with standard IPv4 sockets, the socket address represents the IP address and the port number. A base class named SocketAddress was designed with the goal of representing all possible current and future sockaddr types. sockaddr is a structure which is 'overloaded' by the operating system to provide socket address types for many various address families. Address families define classes of sockets which share the same address type. Both UDP and TCP sockets use AF_INET as their address family types which is the address family for IPv4 sockets. The SocketAddress class contains several pure virtual functions designed to retrieve the operating system specific sockaddr structures. The class is then inherited by subclasses which take responsibility for a specific address family type. The subclass in control of IPv4 socket addresses is called SocketInetAddress. It inherits from both SocketAddress and the operating system's sockaddr_in structure. The virtual functions from SocketAddress are then implemented to return various details from the sockaddr_in structure which the operating system will ask for as parameters to socket functions such as connect and bind.

The next step was to create an abstraction for the socket descriptor. This abstraction was designed in the form of another class called Socket. Many functions exist that are common to all types of sockets such as accept, bind, close, connect, listen, and shutdown. Another functionality that is common to all sockets is the ability to send and receive data. All of these common functions were designed into the Socket class as member functions so that no code outside the Socket class need access the operating system's methods. Other sections of the library which need to make socket function calls can simply call the appropriate member function on the Socket class. This effectively hides any intricacies of dealing with the operating system's API inside one class which can be easily reconfigured to support multiple operating systems.

Now that multiple abstractions have been designed to deal with the specifics of each operating system the library needs to deal with, the design for the actual *iostreams* implementation can begin to take shape. Luckily, the *iostreams* interface provides a very extensible mechanism to create new stream implementations. This extensibility is provided by the stream buffer interface, or basic_streambuf as it is referred to in §27.5 of the 2003 C++ standard (ISO/IEC 14882:2003). By creating a new class which extends the basic_streambuf class, it is possible to create a new stream type which sends and receives its data from a socket instead of from the console or from a file. This class is called BasicSocketBuffer. It provides the essential functionality required to implement a basic_streambuf interface. However, since some methods of sending and receiving data require different methods to be called, two `protected` member functions were designed into the `BasicSocketBuffer` class to deal with this case. These methods, `Read` and `Write`, are designed as pure virtual functions in this class allowing any new socket buffer implementation to be creating easily by just overriding the `Read` and `Write` functions. The remaining functions in `BasicSocketBuffer` are generic enough to work regardless of the underlying data read and write methodologies.

The safety and security guarantees of the Sockets++ library are provided by the `basic_streambuf` class. This class requires only that the `BasicSocketBuffer` class override the buffer underflow and buffer overflow methods to refill the get buffer when it runs dry (underflow) or to flush the put buffer when it fills up (overflow). The methods are named, appropriately enough, `underflow` and `overflow`. Given that the `basic_streambuf` class is secure from buffer overrun attacks, it is only necessary to verify the correct functionality of the overridden `underflow` and `overflow` functions in `BasicSocketBuffer` in order to assert an increase in safety and security against buffer overrun attacks. Barring a purposeful attempt by the library user to cause a buffer overrun error inside the `BasicSocketBuffer` class, this series of abstractions provides protection from stack smashing attacks.

The first class designed to extend the `BasicSocketBuffer` class is the `BasicSocketInetBuffer`. As indicated by the name, this class provides the implementation of the `Read` and `Write` methods that were pure virtual in `BasicSocketBuffer` for reading and writing to IPv4 sockets.

The next class designed to extend the `BasicSocketBuffer` class is the `BasicSocketSslBuffer`. It provides an implementation of the `Read` and `Write` methods that send and receive data through an SSL connection constructed on top of a normal TCP/IPv4 socket. In order to support all the various configuration options available through SSL, it permits an addition argument to the constructor not seen in the `BasicSocketInetBuffer`. This argument is the SSL context, a structure provided by the OpenSSL library that defines most of the various configurable features of an SSL connection.

2.2. Sockets++ Threading Design

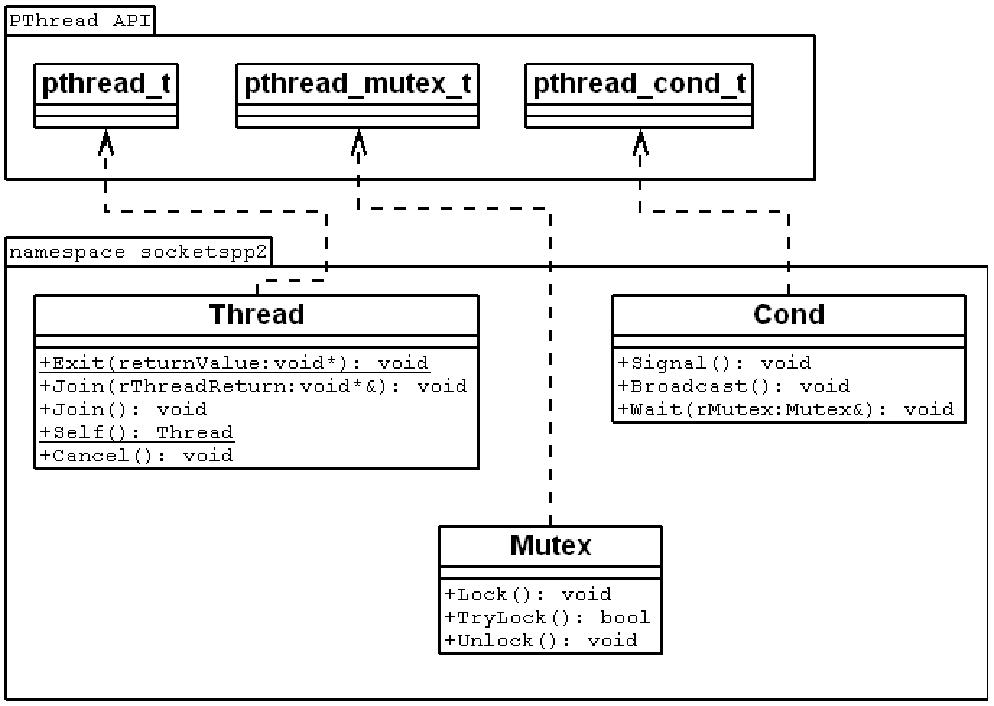


Figure 3: Sockets++ Threading Library Design

The design up to this point is sufficient to allow users to write single threaded client applications using sockets through the Sockets++ library. However, there is no mechanism yet provided by the library to enable easy server creation. Though it is possible to create a socket that acts as a server side socket, the library provides no mechanism for multithreading and is thus limited in potential as a server side tool.

To remedy this lack of server-side usability, the design once again focused on abstracting existing operating system specific concepts so that cross system porting would be limited in scope to a few classes rather than a library wide process. In this new round of design, the multithreading API, *pthreads*, was abstracted into several classes. These classes are `Cond`, `Mutex`, and `Thread`. They represent, respectively, the *pthread* concepts of a conditional wait variable, a mutual exclusion lock, and, of course, a thread.

The `Mutex` class was designed first as it is the most basic of the three. It is simple: The only three methods a `Mutex` instance provides are `Lock`, `TryLock`, and `Unlock`. `Lock` and `Unlock` naturally provide the basic ability to take the `Mutex` and release the `Mutex`. `TryLock`, unlike `Lock`, is guaranteed to return immediately after attempting to take the `Mutex` while returning whether or not it successfully acquired the lock.

Following the `Mutex` class, the `Cond` class was designed to abstract a system dependent conditional wait variable type. Like the `Mutex` class, the `Cond` class provides a simple interface. The `Wait` method takes as an argument a `Mutex` which it releases, and then the thread which called `Wait` will not get control back from the `Wait` method until the `Cond` instance gets a signal. The `Signal` method indicates that one thread currently waiting on this `Cond` instance should receive control back from the `Wait` method. Similarly, the `Broadcast` method indicates that all threads currently waiting on this `Cond` instance should receive control back from the `Wait` method.

Figure 3 illustrates the UML design for the threading abstraction layer provided by the library. Of course, this UML design indicates the one-to-one correspondence between Sockets++ classes and *pthreads* concepts. In other operating systems which do not provide a *pthreads* interface, the code for the Sockets++ library will be more involved than simple delegates. As long as the Sockets++ library provides the same interface on the library user end of the threading API, the library can worry about the intricacies of what the underlying operating system is. The application developer should not need to be concerned with these details.

3. Implementation

The IPv4 specific implementation of the interface provided by the `SocketAddress` class is written into `SocketInetAddress`. As mentioned in the design section, this class is primarily an abstraction to ease porting the library. The `SocketAddress` class defines an enum which matches up to a group of preprocessor constants defined by the operating system's socket interface. This enum represents the socket address family. These indicate to the operating system what type of socket address it has received. The most widely used are `AF_INET` (for IPv4 socket addresses), `AF_UNIX` (for UNIX socket addresses), and `AF_INET6` (for IPv6 [1] socket addresses). All of the functions present in `SocketInetAddress` simply delegate to operating system methods. The two routines of interest in `SocketInetAddress` are the two constructors. One takes an `in_addr` structure and a port number and simply copies them into the appropriate place in the `sockaddr_in` structure. The other constructor takes a hostname or IPv4 address in a human-readable string and a port number. It then performs a DNS lookup if necessary to resolve the hostname and fills in the `in_addr` structure and port number inside the `sockaddr_in` structure. Both constructors assume arguments are in host byte order and convert the fields to network byte order as necessary.

The `Socket` class is once again an abstraction of operating system methods. The operating system methods have return values which indicate different failure conditions. Instead of passing these return values back to the user program to interpret, the `Socket` class will instead throw an exception indicating a failure. This way, the library user need not be concerned with the intricacies of the operating system call under each method. Instead, any error will result in a SocketException being thrown with a message to explain what went wrong. Similar to the SocketInetAddress class, the Socket class defines a few enums to represent the operating system constants which are grouped together. These enums are the protocol family of the socket, the type of the socket, and finally the three modes which are available for calling Shutdown. The protocol family represents such values as PF_UNIX, PF_INET (IPv4), and PF_INET6 (IPv6). The two commonly used types are SOCK_STREAM (TCP) and SOCK_DGRAM (UDP). The Socket class does not inherent from anything and the only data member is an integer which identifies the socket file descriptor. All of its methods, except for one, simply delegate their work to the operating system methods that perform the appropriate task. The one method that is not just a delegate is conditionally compiled only if the OpenSSL library is present. This method, getBio, returns a BIO object which is capable of being used by the OpenSSL library to establish an SSL/TLS connection on this socket.

The BasicSocketBuffer class is templated in exactly the same way as the basic_streambuf class provided by the standard template library. It inherits from the basic_streambuf and passes through the template types to the basic_streambuf class. To comply with the C++ standard for creating a new instance of basic_streambuf, the BasicSocketBuffer overrides the underflow, overflow, and sync methods. These methods are called from code inside the basic_streambuf. The basic_streambuf deals with all the normal buffering activities, and the BasicSocketBuffer only needs to concern itself with buffer underflow and overflow conditions. Additionally, the library user can force the buffer to be flushed to the output sink. This case is handled by the sync function. The BasicSocketBuffer class calls some virtual functions, Read and Write, to handle getting data from the input source and sending data to the output sink. This allows the creation of different types of BasicSocketBuffers by simply overriding the Read and Write methods. Currently, the two subclasses of BasicSocketBuffer are BasicSocketInetBuffer and BasicSocketSslBuffer. These override the Read and Write methods to receive and send data from IPv4 sockets and SSL/TLS connections respectively.

At this point in the implementation, sufficient structure exists to allow creation of socket-enabled single-threaded applications. For example, a simple HTTP retrieval program can be written using the Sockets++ library in 15 lines of code.

```
Step 1:     socketspp2::Socket s(socketspp2::Socket::DOMAIN_INET,
                  socketspp2::Socket::TYPE_STREAM);
```

Step 1: Create a Socket object for IPv4 (`DOMAIN_INET`) and TCP (`TYPE_STREAM`).

```
Step 2:     socketspp2::SocketInetAddress sia(“www.google.com”, 80);
```

Step 2: Create an address object representing the IPv4 address of www.google.com on port 80 (the standard HTTP port).

```
Step 3:     s.Connect(sia);
```

Step 3: Connect the socket to the address.

```
Step 4:     socketspp2::SocketInetBuffer sib(s);
            std::iostream sockio(&sib);
```

Step 4: Create a socket buffer object from the socket object and then create an iostream object from that buffer.

```
Step 5:     std::string line;
            sockio << “GET / HTTP/1.1\r\n”
                   << “Host: www.google.com\r\n”
                   << “Connection: Close\r\n”
                   << “\r\n”
                   << std::flush;
```

Step 5: Send the HTTP request to get http://www.google.com/.

```
Step 6:     while(std::getline(sockio, line))
            {
                std::cout << line << std::endl;
            }
```

Step 6: Read in lines from the socket and echo them to `STDOUT`.

The same program written in the normal socket methodology would require close to 50 lines with multiple opportunities for errors. These opportunities include forgetting to check the return codes on the multiple BSD socket API calls that must be made. Additionally, the input/output must be conducted through raw character buffers by the application developer. This introduces the possibility for serious stack overflow issues. Plus, the application is then operating system specific. The Sockets++ approach allows the application to be written system independently allowing it to run anywhere the library has been ported to.

Before showing an example server, it is necessary to discuss the implementation of the multi-threading capabilities provided by the Sockets++ library. As mentioned in the design section, the three classes provided by the library to abstract the operating system's threading methodology are `Thread`, `Mutex`, and `Cond`. This still does not provide a simple methodology to design a Sockets++ server however. To ease the implementation of multi-threaded servers, the Sockets++ library also provides a `ThreadPool` class. This class starts up a specified number of threads (specified by the constructor argument, the default is 32) when an instance of the `ThreadPool` is constructed. These threads end up sitting in a `Cond` instance's `Wait` function waiting for tasks to be added to the `ThreadPool`'s task queue. When a task is added to the task queue, one of the waiting threads is awaken, it is assigned the new task, it executes the task, and then it returns back to the `Wait` function from which it was awaken and repeats the cycle. Tasks are assigned to the `ThreadPool` by the library user. The function `AddTask` takes a function pointer and a void pointer to an argument to give to the function. The `Finish` function on the `ThreadPool` object sleeps while waiting on the `ThreadPool` to complete all its assigned tasks. It then joins into each thread in the pool in order to complete all the threads. The destructor performs the same exact function without waiting for the task queue to empty out.

This allows easy creation of multi-threaded server applications using the Sockets++ library. An echo server makes a perfect example application and demonstration of the library's multi-threading capabilities.

```
Step 1:     int main()
            {
                socketspp2::ThreadPool pool;
```

Step 1: Create a ThreadPool object (default constructor starts 32 threads)

```
Step 2:         socketspp2::Socket s(socketspp2::Socket::DOMAIN_INET,
                  socketspp2::Socket::TYPE_STREAM);
                socketspp2::SocketInetAddress sia("127.0.0.1", 21);
```

Step 2: Create a TCP/IPv4 socket and an address for localhost on port 21

```
Step 3:         s.Bind(sia);
                s.Listen();
```

Step 3: Bind the socket to the address and begin listening on the socket for new connections

```
Step 4:         socketspp2::Socket s2;
```

Step 4: Create a temporary socket object to represent a new connection

```
Step 5:         do {
Step 5.1:           s2.Assign(s.Accept());
Step 5.2:           if(s2)
                    {
```

```
Step 5.3:               pool.AddTask(&EchoClient, new
                  socketspp2::Socket(s2));
                    }
                } while(s2);
                return 0;
            }
```

Step 5: Loop while we do not receive an error from the accept method on our listening socket.

Step 5.1: Assign the new connection's socket to the temporary object we created.

Step 5.2: Check to make sure the socket is a valid connection.

Step 5.3: Add a task to the thread pool to call method `EchoClient` with the given argument.

The `EchoClient` method takes care of processing each connection for as long as the connection exists:

```
Step 1:     void* EchoClient(void* socket)
            {
                socketspp2::Socket *s =
                  reinterpret_cast<socketspp2::Socket*>(socket);
```

Step 1: Cast the void pointer parameter to a socket pointer type.

```
Step 2:         socketspp2::SocketInetBuffer sib(*s);
                std::iostream sock(&sib);
```

Step 2: Create the iostream object from the socket.

```
Step 3:         std::string line;
                while(line != “QUIT” && sock)
                {
                    std::getline(sock, line);
                    sock << line << std::endl;
                }
```

Step 3: Read lines from the connection while it is open and while the line is not the value "QUIT" and echo the lines back to the socket

```
Step 4:         s->Close();
                delete s;
                return NULL;
            }
```

Step 4: Since the socket is on the heap and not the stack, need to make sure to cleanup before returning to prevent a memory leak.

This application, in less than 40 lines, starts 32 threads as the default behavior of the `ThreadPool` object's constructor; it then dispatches incoming connections from localhost on port 21 to a thread to be dealt with for the life of the connection. It sends everything it receives back along the same connection until it receives 'QUIT' on a line by itself. The same application written without benefit of the library requires many times the line count and introduces numerous unnecessary opportunities for the introduction of logic errors, multi-threading issues, and buffer overflow exploits.

4. Onion Routing and The Personal Private Proxy Server

To validate the library and its utility, a practical end-to-end system is constructed that adds additional privacy features to common web browsing. The majority of modern web browsers willingly give away a substantial amount of user and system information to web servers when requesting pages from them. This information may be used by illegitimate websites for nefarious purposes. The browser user should have more control over what information is given away, but technical ignorance and/or laziness is the norm among most web users. Much of the time, enough data is given away to uniquely identify the user without their knowledge or permission. The Sockets++ library is used to create a Personal Private Proxy Server to remove confidential data from the HTTP stream before forwarding the request on to the remote web server. Since some websites only work on a select few browsers but all browsers support proxies, the proxy will provide browser independence as far as security of client data is concerned. However, the proxy server itself is not capable of hiding the logical address of the client machine. To provide anonymity, the Personal Private Proxy Server will communicate with the Internet by "onion routing" using Tor [10]. Tor provides network layer anonymity by forwarding streams through several servers in between the client and end system and protects data confidentiality by repeatedly encrypting the data so that as each server in the route receives it, all it can do is peel off one layer of the encryption and forward the stream on to its next destination. Thus, only the remote end system and the last hop in the Tor network see the original data stream. Figure 4 illustrates how a web browser using Sockets++ and a Tor client built using Sockets++ provides safe and secure networking along with application level privacy.

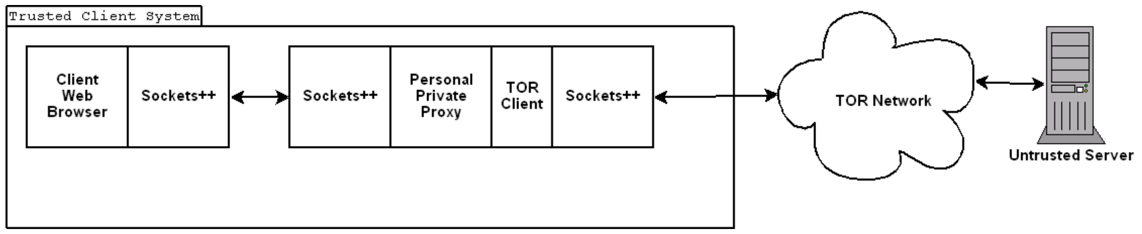


Figure 4: Anonymizing Proxy High-Level Overview

4.1. Tor Overview

In order to establish a Tor connection, the Tor proxy must first know what nodes are out there. The demonstration Tor client included with the library has a hard-coded list of a few nodes and their public keys. Fully featured versions of the Tor proxy have a list of about five nodes that are 'trusted' to provide a list of Tor nodes. The directory list is signed by the node the list is hosted on to verify that no one has tampered with the list in between the Tor proxy and the directory listing. These 'trusted' Tor directory nodes appear to be the weakest point in Tor's anonymity guarantees. If the directory servers were to be compromised, the Tor proxy could be tricked into setting up a Tor connection through Tor nodes which all belong to the same person, thus eroding the anonymity guarantees made by Tor. Figure 5 below illustrates how the Tor directory is retrieved by the Tor proxy.

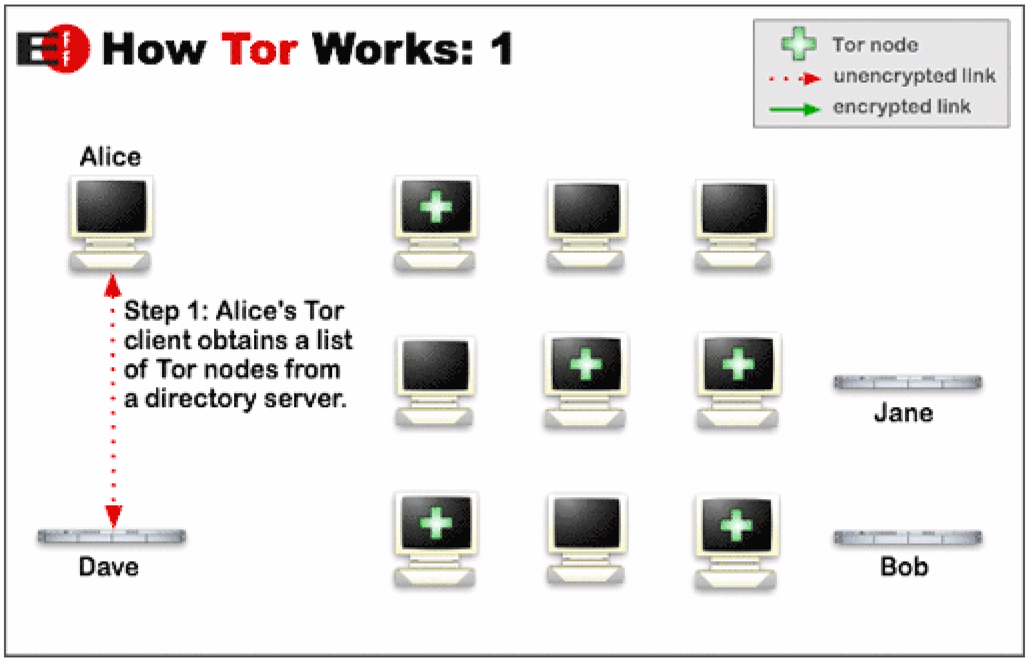

How Tor Works: 1
Tor node
unencrypted link
encrypted link
Alice
Step 1: Alice's Tor client obtains a list of Tor nodes from a directory server.
Jane
Dave
Bob

Figure 5: Step 1: Establishing a Tor connection [10]

Once the Tor proxy has the list of Tor nodes provided by the directory, it will randomly choose two of them for the first two nodes in a circuit. Tor nodes are allowed to establish 'exit policies' which specify IP addresses and ports that can be connected to from that Tor node. Some nodes establish a policy in which no exit is allowed that can only be used for node 1 or node 2 in the circuit. The third node must allow the connection that the client wishes to make. For example, to prevent misuse, almost all the Tor exit nodes do not allow connections to be made to port 25 (SMTP's port) in order to prevent anonymous email spam from coming out of the Tor network. Once the three nodes are selected, the Tor proxy creates a connection one node at a time starting from node 1 going to node 3. The only node that the client system connects to directly is node 1. In order to set up the session keys, Tor uses Diffie-Hellman key exchange [2]. However, since Diffie-Hellman key exchange is subject to man-in-the-middle attack, Tor wraps the publicly traded parts of the Diffie-Hellman key in RSA public key encryption so that only the intended recipient will know the public portion (g^a mod p) of the Diffie-Hellman key. The remote node must then verify with the proxy that it has received the Diffie-Hellman key and sends a hashed value of the key back to the Tor proxy so the Tor proxy can check to see if they agree on the key. If the value received matches the expected value, then the Tor proxy knows the remote system has the same value for the Diffie-Hellman key and the connection has been successfully established. The key is then expanded via hashing it together with some constants to create a forward and backward symmetric key and a forward and backward data digest value to be used to initialize a hash function which will keep a running hash of everything that has been destined for and originated from this Tor node. Once these values are established, the connection between the two nodes is ready to be used.

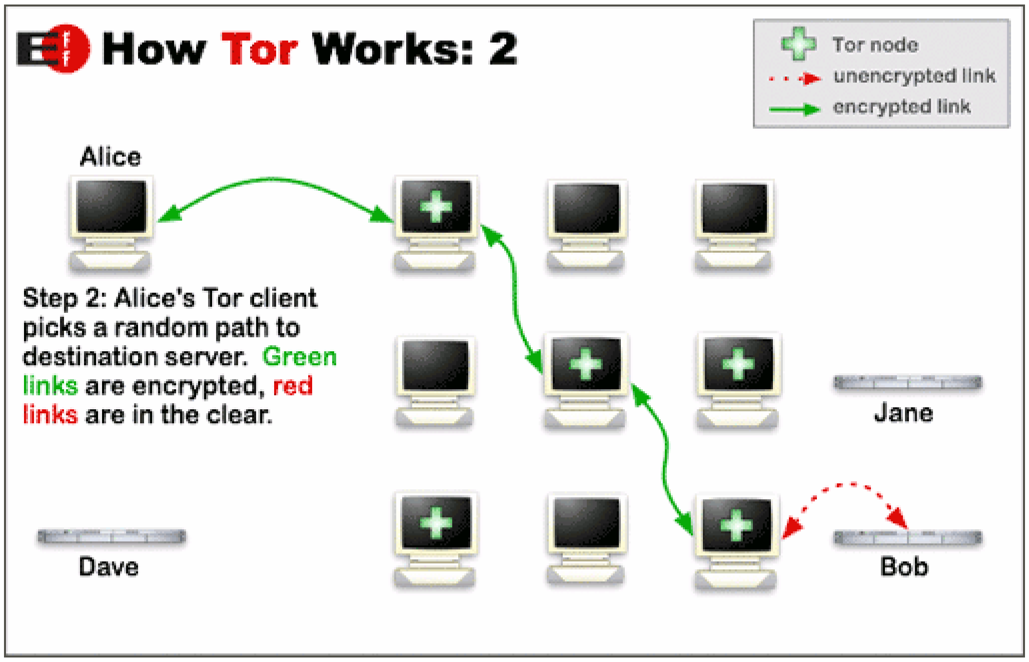

How Tor Works: 2
Tor node
unencrypted link
encrypted link
Alice
Step 2: Alice's Tor client picks a random path to destination server. Green links are encrypted, red links are in the clear.
Jane
Dave
Bob

Figure 6: Step 2: Establishing a Tor connection [10]

As mentioned earlier, the client system does not communicate directly with anyone other than node 1. Thus to extend the connection to node 2, the client system tells node 1 to set up a connection with node 2. However, the key exchange is done via relay between the client and node 2, once again relying on RSA to prevent node 1, or any other man-in-the-middle, from getting the Diffie-Hellman key. The same key verification process takes place and the same key expansion takes place. Thus once the connection to node 2 is established, the client has four symmetric keys and four running digest hashes. Node 1 only knows its two symmetric keys and its two digests. It does not know how to communicate with node 2, even though it is the one that has the actual TCP/IP connection to node 2. Node 2 only knows its two symmetric keys and its two digests. The same process is used to extend the circuit to node 3. Node 1 is asked to relay a message to node 2 which asks node 2 to extend the circuit to node 3. Once the third node is established into the circuit, the client can ask all three to relay along the circuit to node 3 which is then asked to create a connection to the ultimate destination of the circuit. See Figure 6 above and Figure 7 below for examples of what the Tor circuit looks like in its fully established state.

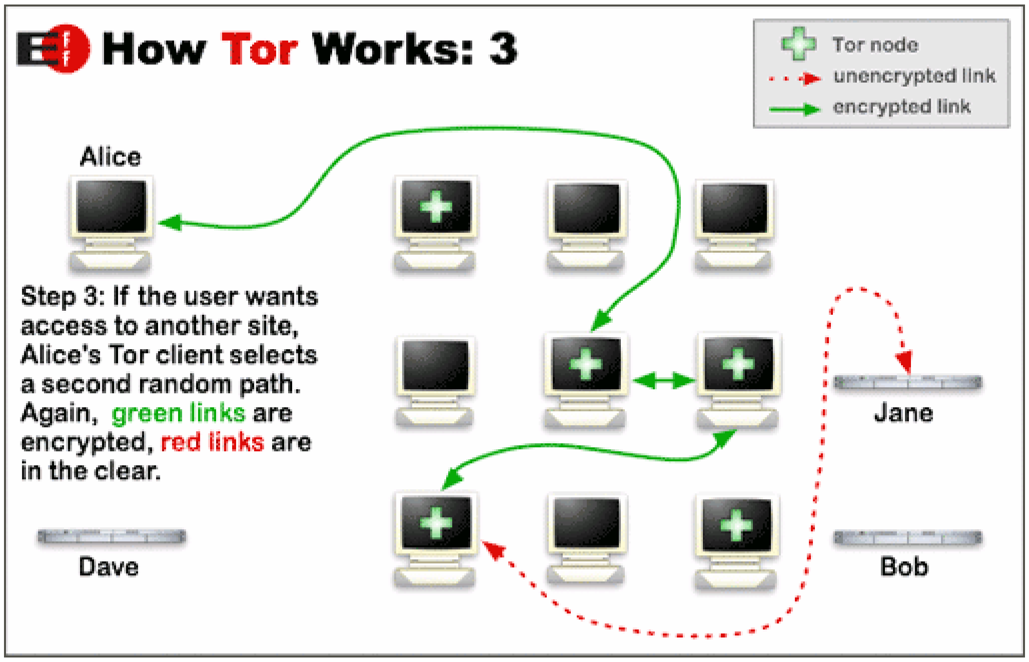

How Tor Works: 3
Tor node
unencrypted link
encrypted link
Alice
Step 3: If the user wants access to another site, Alice's Tor client selects a second random path. Again, green links are encrypted, red links are in the clear.
Jane
Dave
Bob

Figure 7: Step 3: Establishing a second Tor connection [10]

Each link is established using standard SSLv3 or TLSv1 as the first layer of encryption. As discussed earlier, each link also has established a pair of keys and digests with the originating Tor proxy system. Data being sent outbound in the circuit from the Tor proxy to the endpoint is encrypted first using one of the keys setup between node 3 and the Tor proxy. It is encrypted again with one of the keys established between node 2 and the Tor proxy. Finally, it is again encrypted with one of the keys established between node 1 and the Tor proxy. Thus, node 1 and node 2 can not see the data or final destination for the message. Node 1 can not see node 3 and node 3 can not see node 1. Additionally, nodes 1 and 2 are not sure whether they are receiving messages from the originator of the circuit or just another Tor node. The only way node 3 knows that it is not communicating with the circuit originator is because it is the exit point and almost all Tor circuits are created with three nodes in them.

Similarly, when data is returning from the endpoint of the circuit back to the circuit originator, the data is encrypted by each node as it is sent back along the circuit. Once the message reaches the Tor proxy at the circuit origin, it must decrypt the message three times to get to the original content. This sequencing of encryption and decryption is where the name onion routing comes from. The encryption is added on and peeled away like layers of an onion.

4.2. Sockets++ Tor Client

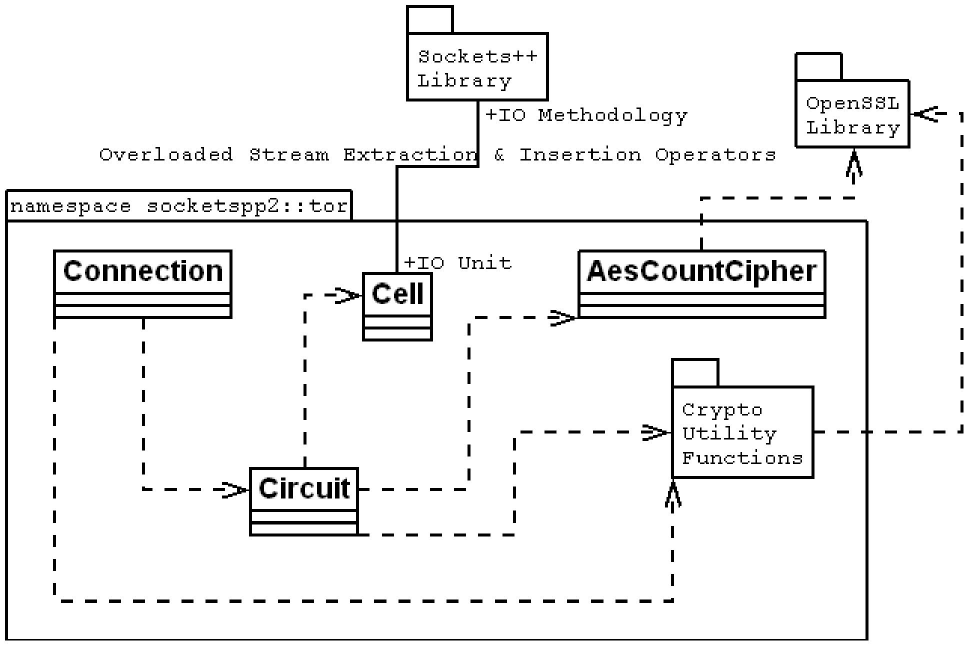


Figure 8: Tor Design Using Sockets++

The original Tor client is extremely interconnected with the operating system’s socket library and is very difficult to follow. Many functions have parameters which are no longer used and most functions also include ‘goto’ statements.

As a test and demonstration of the Sockets++ library, the Sockets++ library includes a rewritten Tor client. The design for this Tor client is illustrated in Figure 8. All the data sent back and forth through a Tor connection is sent via Tor cells. These cells are 512 bytes in length and have a standardized format which depends on the type of cell. The Sockets++ Tor client has a class named `Cell` which overloads the stream extraction and insertion operators so that reading and writing to the Tor circuit is as simple as:

```
>           sockio >> c; // sockio is a socket stream and c is a Cell
>           sockio << c;
```

The cell can be created or parsed by functions in the `Connection` class to send data across a circuit or receive data from the circuit. At some point in the future, it should be possible to include this processing into a stream buffer so that reading and writing to a Tor circuit is as simple as reading and writing across a regular socket connection in Sockets++.

The `Circuit` class keeps track of the series of Tor nodes that are linked together to form an end-to-end connection through the Tor network. It must keep track of the two symmetric keys that have been established between each node in the circuit. Additionally, it must remember the cipher method's state for each of these keys since the cipher used is a stateful cipher.

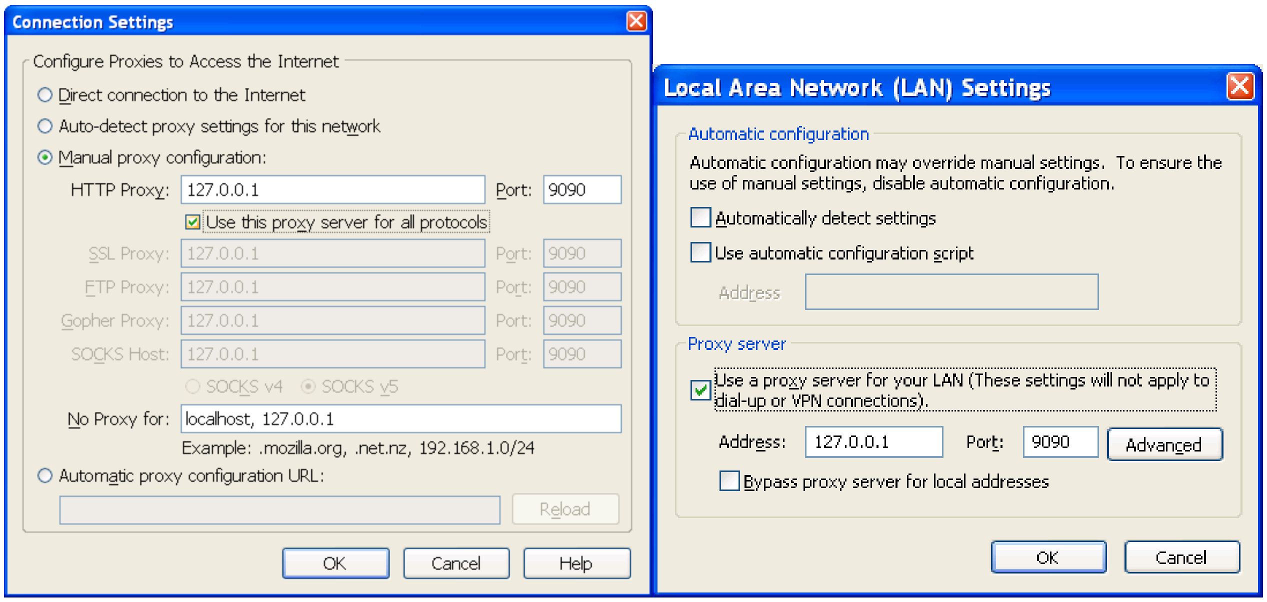
The AES [1] count cipher as used in Tor keeps a sixty-four bit counter which is placed in a 128 bit value with the first sixty-four bits set to zero. It then encodes this value using AES128 and the established symmetric key. The encoded buffer is then XORed with the data that will be sent across the Tor network to produce the encrypted data. The encoded buffer is only used to encrypt 16 bytes after which the counter is incremented and the encoded buffer is recreated. The class `AesCountCipher` keeps track of all of these inner workings and allows other portions of the Tor client to make method calls on it to encrypt and decrypt data.

Thus, the `Circuit` class stores two `AesCountCiphers` for each node (one for data being sent and one for data being received). Additionally, Tor provides data integrity guarantees with a digest which keeps track of all the data which has been sent to a given Tor node and another digest which tracks all of the data which has been received from a given Tor node. The `Circuit` class must keep track of these values as well.

The `Connection` class represents the actual socket based connection between the Tor client and the first node in the circuit. Additionally, it is possible to have multiple circuits on a connection uniquely identified by a 16 bit integer. These circuits naturally share the same first node (otherwise they would not be on the same connection). The `Connection` class provides a place to store the socket, the socket SSL buffer, and the socket stream necessary to communicate with the node at the other side of the socket. Also, it must keep a map from circuit identifiers (the 16 bit integer) to the `Circuit` object so that data coming in from this connection can be dispatched to the appropriate `Circuit` object.

4.3. Personal Private Proxy Server

Built on top of the Tor client, the anonymizing proxy server prevents a browser from revealing information about the client system's identity to the other end of the connection. Tor prevents the actual connection from providing information about the client system's identity to the other system. Tor also prevents any middleman from knowing both the source system and the destination system. However, many browsers will willingly send information about what websites you visited before the one you are currently viewing, tracking information sent to the browser by the remote site, the type and version of the browser, and operating system information. The anonymizing proxy server receives proxy connections from any modern browser (as all the recent browsers support proxies). Since HTTP is standardized, rather than writing a plug-in for each browser to remove these potentially identity revealing data items, we can filter them out of the HTTP stream in the proxy before forwarding the HTTP stream through Tor. Thus, in one utility, all modern browsers can be provided with anonymity from both the data revealing one's identity and from the IPv4 connection infrastructure revealing one's identity.


Connection Settings

Configure Proxies to Access the Internet

- ○ Direct connection to the Internet
- ○ Auto-detect proxy settings for this network
- ● Manual proxy configuration:

HTTP Proxy: 127.0.0.1 Port: 9090

☑ Use this proxy server for all protocols

SSL Proxy: 127.0.0.1 Port: 9090

FTP Proxy: 127.0.0.1 Port: 9090

Gopher Proxy: 127.0.0.1 Port: 9090

SOCKS Host: 127.0.0.1 Port: 9090

○ SOCKS v4 ● SOCKS v5

No Proxy for: localhost, 127.0.0.1

Example: .mozilla.org, .net.nz, 192.168.1.0/24

○ Automatic proxy configuration URL:

Local Area Network (LAN) Settings

Automatic configuration

Automatic configuration may override manual settings. To ensure the use of manual settings, disable automatic configuration.

☐ Automatically detect settings

☐ Use automatic configuration script

Address

Proxy server

☑ Use a proxy server for your LAN (These settings will not apply to dial-up or VPN connections).

Address: 127.0.0.1 Port: 9090

☐ Bypass proxy server for local addresses

Figure 9: Firefox Proxy Configuration (left) and Internet Explorer Proxy Configuration (right)

5. Contributions

1. A new open source ANSI C++ standard Sockets++ *iostreams* framework and implementation that ensures that all network communication is not subject to common buffer overflow attacks. The framework utilizes robust *iostreams* buffering abstractions, consistency checking, and is compatible with OpenSSL, POSIX threads, and the C++ Standard Template Library. The Sockets++ library is available at http://www.sourceforge.net/projects/sockets2 and consists of nearly 6000 lines of code.
2. A new open source ANSI C++ framework and implementation of the Tor onion-routing client that uses encryption and Sockets++ to enable application-level privacy and enhanced security when used with Sockets++. The Sockets++ Tor onion routing client is available as a portion of the Sockets++ library.
3. A simple Web Proxy Server that is built from the previous components that enables additional privacy and anonymity when configured for use with standard web browsers.

In particular, the proxy protects against snooping of a user's settings, etc. The Personal Private Proxy Server is also available within the Sockets++ library code base at the same URL.

6. Conclusion

Encapsulating operating specific methods for dealing with threading and networking inside a library allows easy application porting and enhanced stability, safety, and security. The potential for errors is greatly reduced by providing a unified methodology, regardless of underlying operating system, to access networks and utilize threading. Many of the major modern errors discovered in computer applications are caused by buffer overflow attacks causes by programmer error, oftentimes in the networking layer of an application due to its unorthodox method of sending and receiving via raw buffers. By providing a familiar interface, which C++ programmers begin learning from their very first ‘Hello World’ program, for sockets, not only does the library remove barriers for entry into network-enabled applications, but it also extremely reduces the risk of buffer overflow exploits existing in the application. Additionally, using the same framework, additional tools can be provided to assist in the creation of secure channels of communication via OpenSSL and anonymous channels of communication via Tor.

7. Future Work

Support for a wider variety of address families should be built into the Sockets++ library. Given the library’s design, however, this should be as simple as creating more subclasses of the `SocketAddress` class.

The included Tor client requires additional work to make it more stable and do more testing on the Tor network. Currently, it is serves primarily as a test and demonstration of the library’s capabilities. However, with a little additional work, it could become an easy to use and fully functional Tor client library which could be provided for reuse by other applications. The fact that it uses Sockets++, however, significantly adds to its safety and overall security.

The included Tor client needs to have a parsing engine created for parsing the Tor directory server file format. The nodes it uses in creating a circuit are currently hard-coded along with their public keys into the source code which makes the Tor client only useful as a demonstration.

An additional compile time option to enable Tor not just as a stand alone proxy, but as a library element would be an excellent addition to the Sockets++ library. It should be fairly easy to create a stream buffer which uses a Tor connection as its data source and sink instead of an IPv4 socket or an SSL/TLS connection.

Additionally, there are many more applications which could be implemented using the Sockets++ library. The library is openly available from SourceForge under an open source license that allows anyone to use it under the terms of the GNU Public License v2. One application in particular that would be very interesting and useful to implement in terms both of usefulness as an application and as a test for the library would be a web server. This would test the full capabilities of the library from the basics of socket streaming to the more advanced multiple threading capabilities of the library.

Bibliography

1. "Advanced Encryption Standard (AES)." 26 November 2001. *National Institute of Standards and Technology (NIST)*. 1 December 2006. <http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf>
2. Diffie, Whitfield and Martin E. Hellman. "New Directions in Cryptography." *National Science Foundation (NSF)*. 3 June 1976.
3. Huitema, Christian. *IPv6: The New Internet Protocol*. Upper Saddle River, NJ: Prentice-Hall, 1998.
4. "IPSec." *Wikipedia: The Free Encyclopedia*. 16 November 2006. 1 December 2006. <http://en.wikipedia.org/wiki/IPSec>
5. ISO, IEC, ANSI, and ITI. *Programming Languages – C++*. *ISO/IEC 14882*. ANSI, 2003.
6. Kleiman, Steve, Devang Shah, and Bart Smaalders. *Programming with Threads*. Mountain View, CA: Sunsoft Press, 1996.
7. *OpenSSL: The Open Source toolkit for SSL/TLS*. The OpenSSL Project. 30 November 2006. <http://www.openssl.org/>
8. Shea, Richard. *L2TP: Implementation and Operation.* Addison-Wesley Professional, 1999.
9. Stevens, W. Richard. *Unix Network Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1990.
10. "Tor: Overview." *Tor*. 22 August 2006. Electronic Frontier Foundation. 25 November 2006. <http://tor.eff.org/overview.html.en>
11. "Tor Protocol Specification." *Tor*. 20 July 2006. Electronic Frontier Foundation. 25 November 2006. <http://tor.eff.org/svn/trunk/doc/tor-spec-v0.txt>

Appendix A: Source Code and Documentation

Source Code and Documentation are available at <http://sourceforge.net/projects/sockets2/>