SOCKETLIB

SocketLib is an event based, semi-asynchronous socket stream. It derives from standard C++ sockets, therefore, all extractors (>>) and inserters (<<) can be used. Semi-asynchronous method allows programmer to define event handler to handle incoming data asynchronously without taking the ability to read blocking data. The main aims of this system is to reduce difficulty of sockets programming and make socket systems a lot more C++ friendly. The second aim is to make the system small enough to allow it to be integrated with any project. In fact, there are many frameworks that allow easy to use sockets. However, as far as I have seen, none work in semi-asynchronous mode. Moreover, most socket libraries are a part of a larger framework requiring you to add hundreds of files to your project.

Normally working with sockets in C or C++ requires knowledge on BSD sockets API or Windows Sockets (WinSock) API (probably both to make your program cross platform), system calls, threads or processes. This has two implications, the code written will require additional work to be cross platform; and you have to do a lot of learning. An intuitive C++ socket stream will take minutes to understand opposed to hours of reading for the other option.

Second improvement over BSD or WinSock alternative is using C++ classes and namespaces. Especially WinSock heavily uses macros which disturbs C++ coding and might cause problems. For instance WinSock has the macro which replaces `errno` with `*errno()`, effectively disallowing you to use a variable named `errno`.

My socket library uses socketlib as namespace, every class, type and enumeration resides in this namespace. However, there is another namespace (networking) contains network related information, this namespace is also defined in socketlib namespace. Networking contains three enumerations and their respective types; `Protocol` and `ProtocolType`, `Port` and `PortNumber`, `Family` and `FamilyType`. These types are used in other functions as input parameters or results.

Requirements

SocketLib requires GGE/Utils package and pThread library to work. GGE/Utils package is included in the project as well as pThread headers. However, you may need to copy pthread32.dll to Windows directory Currently its only tested on Windows XP, however, it is written and designed to work on *nix class operating systems as well. In fact, pThread library is native to POSIX compatible operating systems such as Linux, Unix, or Mac-OS; however, there is an compatibility

system for Windows too. Currently system is compiled on Microsoft C++ Compiler 14.0 (which is shipped with Visual Studio 2005).

HostInfo and AddressInfo

Our first two classes are **HostInfo** and **AddressInfo**. HostInfo resolves and contains all address information that a host has. It is basically a collection of AddressInfo, where each AddressInfo holds network related information about a specific address. AddressInfo allows easy access to IP address and family (IP v4, IP v6). However, other information can be accessed by obtaining raw **addrinfo** pointer.

Resolve function of HostInfo class can be used to resolve a domain name (or an IP address). There is also **StartResolve** function which can be used for asynchronous checking; whenever resolve finishes, **ResolveComplete** event is called. HostInfo can be used as a Boolean value to check if the resolve is succeeded. The following example illustrates the use of this system. It can print more than one IP address for a server.

```
#include "SocketLib/HostInfo.h"
#include <iostream>

using namespace socketlib;
using namespace std;

void resolved(HostInfo &info) {
  if(!info) { //HostInfo can be converted to bool to check result
        cout<<"Cannot resolve host"<<endl;
        return;
  }

  foreach(AddressInfo, ai, info) { //Collection iteration
        cout<<endl<<"IP address: "<<ai->IPAddress()<<endl;
  }
}

void main() {
  HostInfo h;
  h.ResolveComplete.Register(&resolved);
  h.StartResolve("cmpe.emu.edu.tr");

  cin.sync();
  cin.ignore(1);

  return 0;
}
```

TCPServer

Currently only TCP side of the system is complete, which is enough to implement required functionality in this assignment. **`TCPServer`** is the class that listens and accepts incoming connections. First, Listen function should be called to bind the server to a specific port. Accept procedure can work synchronously or asynchronously. Asynchronous mode fires **`ConnectionReceived`** event. If desired, this event can be fired in a different thread. **`CallConnRcvedEvtInNThrd`** property controls this behavior. ConnectionReceived event uses **`TCPServer::accept_params`** for parameter object which contains the accepted **`TCPSocketStream`**. ConnectionLost event is fired when one of the clients looses connection. ConnectionLost event uses **`TCPServer::connlost_params`** for parameter object which contains the disconnected TCPSocketStream. This system also provides safe resource allocation, i.e. whenever server object is destroyed, all the connections will be disconnected (calling ConnectionLost events), accept thread is terminated, port is released, and all resources are freed.

The following is the list of the TCPServer methods.

- **`Listen( port )`**: Binds the server to the specified port, it can be a **`PortNumber`** (can be obtained from **`networking::Port::portname`** syntax), integer port number, or a port representation string (like http, ftp, etc... numbers are also accepted)
- **`StartAccept( )`**: Start accepting new connections asynchronously
- **`TCPSocketStream &Accept( Timeout )`**: Accepts a connection, if Timeout is not specified, this function will wait indefinitely until a connection is received or socket closed.
- **`TCPServer::Status getStatus( )`**: This function returns the current status of the server. It can be one of the following:
  - **`Idle`**: server performs no operations
  - **`Listening`**: server is listening to the specified port, but not accepting any connections
  - **`Accepting`**: server is asynchronously accepting connections
  - **`BlockingAccept`**: server is blocked in an Accept function
- **`StopListening( )`**: Closes the server socket, effectively unbinding the port and stopping asynchronous accept thread, if running
- **`CloseAll( )`**: Closes all connections
- **`int LiveConnections( )`**: Returns the total number of live connections

Following is a simple server that sends “Hello” to every connected client and disconnects.

```
#include <iostream>
#include "SocketLib/TCPServer.h"

using namespace std;
using namespace socketlib;

void connect(TCPServer::accept_params params) {
  cout<<"Connection received from "<<params.addrinfo.IPAddress()<<endl;
```

```
  params.socket<<"Hello"<<endl;
  params.socket.Close();
}

int main() {
  TCPServer server;
  server.Listen("444");
  server.StartAccept();

  cin.sync();
  cin.ignore(1);

  return 0;
}
```

TCPSocketStream / TCPClient

This class has two different names, **`TCPSocketStream`** and **`TCPClient`**. Its main purpose is to stream data between two sockets. Its second aim is to be the client, connecting to a server. Therefore, it contains connectivity functions as well. This class is derived from standard I/O stream. This means that any object that can be inserted or extracted from stream can be inserted and extracted from this class. However, sockets does not support seeking, therefore, any seek or position request will fail. If you try to send data while the socket is closed, you will get a **`SocketException`** exception, which might be handled by stream system and translated to failed status. However, if the read operation fails due to loosing connection, you will receive EOF notification. Moreover, connection is closed if the object gets destroyed.

Buffer class of this stream has two different buffers for incoming and outgoing data. So both sending and receiving can be performed at the same time. However, Microsoft headers for stream operations uses a single Mutex for both input and output buffers. Because of this the advantage of using sending and receiving at the same time is lost if Microsoft headers are used.

Standard inserters is the preferred method to send data to recipient. Moreover, **`WriteBinary`** function is added to the system for convenience. For any simple object, you may use this function to send its entire data to the other end. Data will be sent on an explicit flush request or whenever buffer is full. **`endl`** stream modifier also flushes the buffer, so it can be used to terminate commands that you need to send to recipient. Send requests are always synchronous, but after data is transferred to operating system, it is queued for sending; your application will not wait for the entire send operation. There is one important point about TCP sockets, when sending data, data might be needed to break into segments. The size of the segments is controlled by operating system or

underlying hardware; therefore, there is no way to be sure that every packet is sent in one send request.

Receiving data works semi-asynchronous mode. In this mode there is one thread always waiting data to read, another thread is used to fire **`Received`** event. First thread starts whenever connection is established and stopped when its closed. Second thread is started when data is received and no requests are made to receive it. This second thread fires Received event and waits for its termination. Parameter object of Received event contains the size of the receive buffer and a reference type Boolean variable called **`shouldrecall`**. If this variable is set to true inside the event handler and there is still data remaining in the buffer, Received event is fired again. This method can be employed to read only one frame every time Received event is fired, delaying remaining data to the second call. Inside the Received event, programmer should read data using **`extractors`**, **`get`**, **`getline`**, **`read`** or **`ReadBinary`** functions. Extraction operations are synchronous, but, since there is data inside the buffer (whenever Received event is fired, buffer definitely contains data) and the size of the data can be determined using the event parameters this method can be used like asynchronous mechanism. An important note is event thread is separate from the main thread and it might be required to synchronize threads.

The following is the list of all methods and variables of TCPSocketStream.

- **`bool Connect( host, port )`**: Resolves and connects to the given host and port, this function works synchronously, asynchronous version is a future work. If it cannot resolve the host or connection fails, this function will return false; if another error occurs it will throw a SocketException.
- **`bool Connect( addressinfo )`**: Connects to the given host using information from an AddressInfo class. For this system to work, you must specify port parameter of resolve function in HostInfo class.
- **`bool isConnected()`**: Returns whether this socket is connected.
- **`Close()`**: Closes the socket, ending accept thread. This function is safe to be used in the receive event thread, however, you cannot destroy calling socket in receive event thread.
- **`int Available()`**: Amount of data available in the read buffer.
- **`Disconnected`** Event: Fires whenever socket is disconnected. Has no specific parameters.

Following is a simple client that connects to the server and display and received data.

```
#include <iostream>
#include "SocketLib/TCPSocketStream.h"

using namespace std;
using namespace socketlib;

void received(TCPSocketStream::accept_received_params params,
```

```
   TCPSocketStream &socket) {

  int cnt=params.available;
  if(cnt>1024) {
        cnt=1024;
  }
  char data[1024];

  socket.read(data, cnt);
  cout.write (data, cnt);

  params.shouldrecall=true; //If data in the buffer is larger than 1k
                                this event handler will be called again
}

void disconnected() {
  cout<<"Disconnected."<<endl;
}

int main() {
  TCPSocketStream client;

  client.Received.Register(&received);
  client.Disconnected.Register(&disconnected);
  client.Connect("localhost", "444");

  cin.sync();
  cin.ignore(1);

  return 0;
}
```