Chatter

Assignment Out:	4/8/09
Code Due:	4/17/09

1 Purpose

This assignment should serve as an introduction to the C++ language and to network programming. This is a pair project.

Your task is to create a sockets library. Sockets let you open connections with other computers (but you've done Newswhere, so you should know this) You will have two different kinds of TCP sockets, a server socket, which sits around waiting for connections, and a client socket that attempts to connect to a server socket. Using these you will then create a peer for a file transfer system.

Your peer will connect to a master server, which will contain information about which files are available for download from which peers. Your peer will then open a connection to a peer containing the desired file and download it. This means your peer will also require its own server sock to accept incoming download requests from other peers.

2 The Sockets Library

Before you get started, we highly suggest you go and read Beej's Guide to network programming (http://beej.us/guide/bgnet/). A lot of what you need to do is described there.

2.1 TCP Socket

We will be giving you a header file for your TCP socket. In TCPSocket you will need to set up the port and get information about the socket, including its file descriptor. Using this file descriptor, you should be able to read and write from a socket in exactly the same way that you read and write from a file. There are two functions to fill out here, init() and close(). The rest are error accessing functions that we've left in there for you. From TCPSocket you will be creating two subclasses, ClientSocket and ServerSocket.

2.2 Client Socket

In ClientSocket you will be setting up your client to connect to a specific server. A client socket is used to represent outbound connections. You will probably need to use getaddrinfo (this is used to convert DNS hostnames and IP addresses to and from human-readable text and binary formats). You will also need to implement a read and a write function, in which you will call recv and send

respectively. For more information on how to use getaddrinfo, send, and recv, we once again direct you to Beej's Guide (and the man pages for basic information).

2.3 Server Socket

Your server socket is used to represent the receiving end of incoming connections. In ServerSocket you will set up a socket to receive incoming connections, which requires setting some server options using setsockopt. You should then bind the socket and tell it to listen, which will cause it to block waiting for connection requests. Additional notes about what options you should set can be found in the comments in the stencil code. In addition to setup, you will also need to write code to create a local client (call the global accept function and initialize a ClientSocket). This local client socket will represent the other end of the connection that was just initiated to the server socket. Creating a client socket for each incoming connection allows you to handle multiple clients simultaneously (in a multithreaded environment), which is required for the next part.

2.4 Details and Testing

If this all seems rather opaque to you now, read Beej's Guide. We can't stress enough how helpful that will be. Also, make sure you perform error checking at every available opportunity; in other words, any time you call a library function that might not succeed, check its return code for errors (or unexpected null values), print the error, and terminate your program. For error checking, you should use the error statuses found in TCPSockets.h, which will tell you in a human-readable manner what broke in your sockets.

Furthermore, we highly suggest that you test your sockets before moving on to the next part. A good way to do this is to have your client write to a testing server that echoes back whatever it received from your client. This will test reading and writing from both types of sockets and will also give you a sense for how to use them in the next part.

3 Client Peer

Now that you have a sockets library, you are obligated to do cool stuff with it! More specifically, you're going to use it to download files from other clients. We will provide you with a master server executable (run it with no arguments). There may also be a TA server running that anyone can connect to (details will go out over email). We will also provide you with a header file to implement the peer with; currently it describes an abstract class, and it is your job to implement it. Your peer will need to be able to connect to the master server as well as other peers. This means that as described above, as soon as you start your peer, you also need it to start a server sock which is ready to accept incoming download requests from other peers.

3.1 Usage

Your Peer should be able to connect to the master server. As a user, you will do this through the user interface provided in the support code. Type **help** for a list of commands and type **help**

<command> for information on how to use a specific command.

3.2 Protocol

It's important that the messages you send to the server be consistent with the given protocol, so that all parties can talk to each other.

All messages should be preceded by the length of the message. The length of the message (including the null terminator) should be sent as a *size_t*, not the ASCII text of the length. For example, for 'hi
0' we send (int)3, not (char)'3'.

3.2.1 Peer-to-Server Communication

- lsfiles This message should be null-terminated string "lsfiles" with no newline. When the server receives this message, it will send a list of all files available for download from other peers. Any files available for download from this peer will not appear in the server's response to an lsfiles message (because this peer already has them and doesn't need to get another copy).

- putfiles This message starts with "putfiles\n" and is followed by a newline-delimitted list of files available for download from this peer in the following form: <file name> @ <server>:<port>. The message should be terminated by a null character. Here is a sample:

```
putfiles\n
movie.avi@192.168.1.101:1234\n
readme.txt@192.168.1.101:1234\n\0
```

3.2.2 Peer-to-Peer Communication

- downloadfile

  This message starts with "downloadfile
  n" followed by the name of the file to download. If Peer 1 received **movie.avi@peer2:1234** in a response to lsfiles from Peer 2 (with hostname *peer2*, Peer 1 would send the following message to Peer 2 on port 1234:

```
downloadfile\n
movie.avi\0
```

  This message should be terminated by null character.

- sendfile This is slightly different than a message because it is not a string. If a peer receives a request from another peer to download a file, the peer should first send the length of the file (as a uint32_t in network byte order equal to the number of chars in the file being sent) and then the file data. No null characters or newlines are required.

4 Support Code

We have provided a Makefile and stencil code for this project. Find it at /course/cs032/pub/chatter. After copying the files over to your personal directory, you can compile and run your program using the provided Makefile. There will be a sockets/api directory within the stencil code. You should finish and test he sockets api before you try to run the peer.

To print to the UI (for status messages, etc.), do the following:

```
ui->statusMessage("yo");
```

The stencil is filled with useful comments to guide you. Please read through them before sending questions.

5 Tips

- Ensure that all of your strings are null-terminated
- Remember to use std:: before using items from that namespace (or type **using namespace std;** at the top of your C file)
- Wherever you new an object, make sure there is a corresponding call to delete
- You probably shouldn't put calls to close() in the destructor of your sockets in case you want to use the same socket again via its file descriptor
- Make sure you convert correctly between network and host byte order (see the manual pages for htonl and ntohl)

6 Handin

Your code should be easy to read.

Make a `README` file which explains any quirks in your code, as well as any special instructions about running your program. You should also mention any major design decisions you made or anything you found troubling or out of the ordinary. Also, please include the login names of both people who worked on the project.

Hand in with the following command, which you should run from the directory which contains everything (and not from your home directory, please!)

```
/course/cs032/bin/cs032_handin chatter
```