Project 5, CSC 101, Fall 2009

Simple Messaging System (SMS)

For this final course project you will develop an integral component of a simple messaging system.

In particular, you will implement (complete the implementation of) a server program for this messaging system (details below). This system will allow a user to send a short message (much like a text message on a cellphone) to another user. Users can list the messages they have received and delete the messages in which they are no longer interested.

Background

Networking: What is this Client-Server thing anyhow?

Consider browsing the web. On your machine you launch a web browser such as firefox. Then, to retrieve a webpage, you connect to another machine by specifying that machine's name and, sometimes, the page in which you are interested. For example, you might connect to http://www.google.com.

You already have experience using a client program to access a server program. Your web browser is the client program (the program that the user interacts with). The web server (maintained by Google) is the server. The client program sends requests to the server. The server processes each request and sends a reply back to the client.

The client and the server communicate over the network by sending messages. These messages can be quite simple (which they will be for this assignment), or they can be very complicated. The difference depends on the needs of the application. For this assignment, the messages sent between the client and the server (your server) will be very simple. In fact, they will be just plain text like that seen in a file.

Networking: Host and port?

When writing networked programs (clients and/or servers) one will need to keep in mind the concepts of hosts and ports.

Let's consider an analogy based on a telephone system at a large company. You can call the company at a give phone number. To speak with a specific person at the company you will need their extension. So you might call the company at 1-800-555-1234 and give extension 5678 for the person of interest. This is very similar to the idea of hosts and ports. A host is similar to a phone number and a port is similar to an extension at that number.

Fortunately, you are already familiar with the concept of a host. This is just the address of a machine on the network. We use names such as falcon.csc.calpoly.edu or www.google.com.

You are likely less familiar with ports since they are not often explicitly stated (though some of you may have seen them when connecting to servers for certain games). For example, web servers typically listen on port 80.

Networking Library: Give me a message!

You will not write any networking code for this assignment. All of the networking code is provided for you in a library of functions. You will compile this library into your program (instructions to do so are discussed below) and call the functions provided.

To use the networking code you will need the following files. Download the header file and one of the net.a files (as appropriate for your platform).

net.h net.a (Windows) net.a (Mac OS X Intel) net.a (Mac OS X G5) net.a (linux for falcon/hornet)

As indicated by the header file, this library provides the following five functions.

```
SERVICE_ID create_service(int port);
CONNECTION_ID accept_connection(SERVICE_ID service_id);
void close_connection(CONNECTION_ID connection);

char *read_string(CONNECTION_ID connection, char string[], int size);
void write_string(CONNECTION_ID connection, char string[]);
```

The first three functions are provided to do the networking setup and teardown. You will not need to use these functions yourself, the code is provided for you in a link below. They are used as follows.

```
 SERVICE_ID initialize_service(int port) {
   SERVICE_ID service_id = create_service(port);
   if (service_id == SERVICE_FAILURE)    {
               perror("unable to initialize service");
                exit(-1);
   }
   return service_id;
}

void run_service(int port, struct user *users, int num_users) {
   SERVICE_ID service_id = initialize_service(port);
   CONNECTION_ID connection;
   printf("Listening on port %d ...\n", port);
   while ((connection = accept_connection(service_id)) !=
CONNECTION_FAILURE)
   {
                handle_connection(connection, users, num_users);
               close_connection(connection);
   }
}
```

The run_service function first creates a network service (given a port) and then repeatedly accepts connections from clients. Each connection is handled by the server (in handle_connection, a function that you will implement) and, once complete, closed.

Your code will use the read_string and write_string functions to read a string from a connection and to write a string to a connection.

char *read_string(CONNECTION_ID connection, char string[], int size)

The read_string function is very similar to the fgets function discussed in your textbook on page 451.

This function takes a CONNECTION_ID from which to read, a string into which the data will be copied, and a maximum number of characters to read (counting the terminating character). This function will read up to the maximum number of characters specified or until a newline is read. The newline will not be stored in the string (this differs from the behavior of fgets).

This function will return NULL if there is no more data to read from the connection or if an error occurs.

Note: Any time a program listens for connections from the network it is susceptible to connections from unknown users. Such users may have malicious intent and may try to "break into" your program. The most common attack is known as a buffer overflow. The read_string function is where such an overflow attack would occur. Be certain that the array passed to read_string has space for at least size characters. If you pass an array that is shorter than the specified size, then the function may write into memory beyond the bounds of the array. You might also consider configuring your firewall to prevent external connections to your server program.

void write_string(CONNECTION_ID connection, char string[])

The write_string function is very similar to the fputs function listed in your textbook on page 843.

This function takes a CONNECTION_ID to which it will write the provided string. This function will automatically write a newline after the given string. As such, the string to write should not include a newline at the end. This is to simplify the code that you will write.

Project Description

This section details what you must do for this project.

Executables: An example of how this all works.

First, download the following pair of programs to see how the client and server work. Your server program, once complete, will behave as this one does. You will use this same client program to interact with your server program. Select the programs as appropriate for your operating system.

- p6-client.exe (Windows) p6-server.exe (Windows)
- p6-client (Mac OS X Intel) p6-server (Mac OS X Intel)
- p6-client (Mac OS X G5) p6-server (Mac OS X G5)
- p6-client (linux for falcon/hornet) p6-server (linux for falcon/hornet)

Server

Start the server by double-clicking its icon. You should see the following prompts and output indicating that it is ready for connections.

```
Please enter the name of the user file: users
Please enter the port on which to listen: 49152
Listening on port 49152 ...
```

The first prompt requests the name of a file listing the users for the system. This file contains one user name (of up to 80 characters) per line. You can download a sample file here. These user names are completely made up. You can modify them as you'd like.

For this assignment, we will restrict the maximum number of users to 10. Your program should work if there are fewer than 10 users. If the file contains more than 10 users, then ignore the remaining users.

The second prompt requests a port number on which to listen for connections. You can select (yes, it is your option) any port between 49152 and 65535. If you get an error message (e.g., unable to initialize service: Address already in use), then the port you selected is already in use. Just choose another.

The server will store messages for users. If the server is closed (which must be done by explicitly killing it, try <ctrl>-c), all messages are lost. This means that messages are only available while the server is running. You might consider extending this program later to save messages to a file so that they persist between executions of the server.

Client

Start the client by double-clicking its icon. The client will prompt you for the name of the machine on which the server is running. In the labs you can find the name on the machine itself (look for the white sticker). If you are running the client and the server on the same machine, you can use localhost (or 127.0.0.1) as the name of the machine. This is a special name that indicates the local machine.

You can also run the server and client on separate machines. Let's consider doing so in the lab environment. Login on two machines (preferably next to each other so that you don't have to run all over lab). Start the server on one of the machine. Now start the client on the other machine and provide the name of the first in response to the prompt. Follow the remaining directions to complete the initialization.

The client will next prompt for the port number. This must match the port selected when you started the server. Finally the client will prompt for a user name. This should match one of the names in the user file read by the server.

The following example shows a short run of the client. User bob sends a message to sue and attempts to send a message to tom. Since tom is not a valid user, an error message is printed.

```
Please enter the name of the machine on which the server is running:
localhost
Please enter the name of the port on which the server is listening: 49152
Enter your user name: bob
1) Send a Message
2) List Messages
3) Delete a Message
4) Quit

Select an action: 1
To whom is this message being sent?: sue
Please enter the message: Hello.

1) Send a Message
2) List Messages
3) Delete a Message
4) Quit

Select an action: 1
To whom is this message being sent?: tom
Please enter the message: Hi, Tom.

User Not Found
```

The next example shows a second execution of the client. This time sue is using the system. She lists her messages (these are retrieved from the server). The only message in sue's list is the one sent just above by bob. After reading the message, sue deletes it. Since this was the only message she had, listing the messages again indicates that there are none.

```
Please enter the name of the machine on which the server is running:
localhost
Please enter the name of the port on which the server is listening: 49152
Enter your user name: sue
1) Send a Message
2) List Messages
3) Delete a Message
4) Quit

Select an action: 2

Message 0
From: bob
Date: Sun Nov 19 12:45:56 2006
Hello.

1) Send a Message
2) List Messages
3) Delete a Message
4) Quit

Select an action: 3
Please enter message number to delete: 0

1) Send a Message
2) List Messages
3) Delete a Message
4) Quit

Select an action: 2

No messages
```

SMS Server: What you must do.

You are to implement the core functionality of the server. The client program (the one you downloaded above) that interacts with the user is already written by the instructor. This client is specifically designed to communicate with your server using predefined messaging protocols so you must use this client in order to test your server.

Here's a start: An initial framework.

To get you started, download the following files.

p6-server.c p6.h

The provided p6-server.c source file includes the code to initialize the network and the code to begin processing client requests. You can modify this code as you see fit. In particular, you might decide on a different data structure than that suggested by the structure definitions at the top of the file. Moreover, you can add additional functions, rename functions, or completely restructure the code. You are encouraged, however, to leave initialize_service and run_service as given.

The provided p6.h header file includes the common definitions used in both the server and the client. In particular, the message types (SEND, LIST, and DELETE), the maximum string lengths (81 for user names and 251 for user messages; these lengths include the terminating character), and error response messages. You will want to send these messages back to the client when an error is detected as discussed below.

Functionality

You will need to add the following functionality to the server. You should add and test one feature at a time. You can add printf statements to the server to display the contents of the messages received from the client (though you should remove these from your final submission).

Cient to Server Messages

Each message sent from the client to the server will begin with an integer followed by a newline. This integer indicates which of the following operations the client is requesting. The provided framework reads this integer and invokes an appropriate handling function. For completeness, these integers are shown in the messages below.

- Send

The client will send a message of the following form; each line ends with a newline character and can be read using read_string.

```
1
user_to
user_from
user_message
```

This message indicates that user_from is sending a message to user_to. The contents of this message, user_message are given in the last line.

Your program must find the user_to and add this user message to the end of that user's list of messages. Your program will need to store the sender of the message (user_from) the date at which it was received (recall the use of time, localtime, and asctime in Lab 6), and the user message itself.

If user_to is not a valid user, then your program must send the USER_NOT_FOUND_MSG message back to the client using write_string.

If user_to already has a maximum number of messages, then your program must send the TOO_MANY_MESSAGES_MSG message back to the client using write_string.

The system will store a maximum of 10 messages per user at any given time. Each user may, of course, have fewer messages. As such, the error code above is used to indicate that the intended recipient of the current message has a full message list.

- List

The client will send a message of the following form; each line ends with a newline character and can be read using read_string.

```
2
user
```

This message indicates that user wants her messages sent back to the client.

Your program must find the user and send each message for that user back to the client using write_string for each line of the response.

For each of the user messages to be sent back to the client, the fields must be sent in the following order.

```
user_from
date
user_message
```

If user is not a valid user, then your program must send the USER_NOT_FOUND_MSG message back to the client using write_string.

- Delete

The client will send a message of the following form; each line ends with a newline character and can be read using read_string.

```
3
user
message_index
```

This message indicates that user wants to delete message at index message_index from her list of messages.

Your program must find the user and remove this message from her list of messages.

If user is not a valid user, then your program must send the USER_NOT_FOUND_MSG message back to the client using write_string.

If message_index is out of the range of messages that this user current has, then your program must send the INVALID_INDEX_MSG message back to the client using write_string.

Linking the library.

To build your program you will need to compile your p6-server.c file with the appropriate net.a file for your system. You can use one of the following Makefiles. If you do not have make installed, then you can execute the gcc command given in the appropriate Makefile.

- Makefile (Windows) or Makefile.bat (Windows Batch File)
- Makefile (Mac OS X)
- Makefile (linux for falcon/hornet)

Grading

10%	Correct functionality: Handling of user file.
20%	Correct functionality: Send feature.
20%	Correct functionality: List feature.
20%	Correct functionality: Delete feature.
20%	Correct error response messages sent to the client.
5%	Coding style.
5%	Clean compile (no warnings using the required compiler flags).

Remember, code that does not compile will receive a grade of zero.

Handing In Your Files

1. Log on to hornet using the Secure Shell Client program.
2. Execute the following command (substitute your section number for the **x** below):

```
handin zwood csc101p6 p6-server.c
```

You should see messages that indicate handin occurred without error. You can (and should) always verify what has been handed in by executing the following command:

```
handin zwood csc101p6
```

This completes the handin process.