| RC 19/20 | COMPUTER NETWORKS | Project #: | 1 |
|---|---|---|---|
| Programming with Sockets | | Issue Date: | 14/Oct/2019 |
| Chat Client & Server | | Due Date: | 31/Oct/2019 |
| Author: Prof. Luis Pedrosa | | Revision: | 1.0 |

## 1 Overview

In this project you will learn the basics of socket programming in the C language, including the basics of connection set up and tear-down, sending and receiving data, and managing multiple sockets at the same time. Towards this end we will be developing a simple chat system using connection-oriented TCP sockets. This system will include a server that multiple clients connect to to participate in a single common chatroom. Like any chatroom style service, any messages sent by one client can be seen by all.

To achieve this goal, the clients simply forward text messages from standard input to the server and display messages sent by the server on standard output. The server, in turn, receives these text messages from the clients and forwards them to all other clients. The server also notifies all when a new user joins the chat room. This involves handling multiple clients and juggling multiple sockets accordingly.

To keep things simple, the clients are simply identified by their IP address and port rather than more descriptive nicknames. The server is responsible for managing these identifiers and prepending them to each users text messages. This way each client sees who sent what message.

Though it is possible to implement the chat server using multiple threads or processes, doing so brings its own challenges of locking and synchronization that are not the focus of this class. As such, no multiprocessing will be allowed within this project, i.e. while the clients and server run concurrently, each can only run in one process/thread. The idea is to use the tools provided by the sockets API, specifically the `select` call, to manage multiple sockets at the same time.

Finally, this project will be partially evaluated with automated test scripts so it is important to follow the specification precisely. Points may be deducted if the project is not submitted in the expected format or if the build process does not work out of the box as described. It is also important to process inputs and outputs exactly as shown in the specification below.

## 2 Specification

### 2.1 The Client

The client executable must be called `chat-client` and takes exactly two arguments from the command line: the server host name and port number. Invoking the client will thus look something like this:

```
:~$ ./chat-client localhost 1234
```

Given these two arguments, the client application should create a TCP socket and connect to the server at the designated host and port. Any error in parsing the arguments or connecting to the server should result in an error, with an exit status of -1.

Assuming the connection is successful the client should subsequently scan both standard input and the server socket for messages. Messages should be seen as a sequence of characters ending in a newline (`'\n'`). We will not test with messages longer than 4096 characters.

The client will scan both standard input and the server socket for new text and upon completion of a new line (i.e. receiving `'\n'`) from either, the message should be forwarded onward (including the `'\n'`). In effect, the contents of standard input are forwarded to the server and any text received from the server is displayed on standard output. The contents of standard error will be ignored during grading and can thus be used for debug purposes. The application should not block while reading input.

The client should process inputs continuously until either the user closes standard input (e.g. by pressing Ctrl-D) or the server closes the connection. When this happens, the client should close the socket, cleanup any resources used, and exit with status 0.

## 2.2 The Server

The server executable must be called `chat-server` and takes exactly one argument from the command line: the server port number. Invoking the server will thus look something like this:

```
:~$ ./chat-server 1234
```

Given this argument, the server application should create a TCP socket and listen for client connections at the designated port. Any error in parsing the arguments or preparing the socket should result in an error, with an exit status of -1.

Assuming all went well, the server should listen for incoming connections from new clients. We will not use more than 1000 clients during testing. Once a new client is accepted, a message should be sent to all clients (including the new one), indicating that a new user has joined the chat room. The message should follow the following format: `"<client-IP>:<client-port> joined.\n"`, for example: `"192.168.56.2:12345 joined.\n"`.

In addition to scanning for new clients, the server should also check for new text from the pool of existing clients. If a client sends a new message (terminated with a newline, see above), the message should be prepended with the client's identity and forwarded to all other clients (but not back to the one that sent the message). The broadcasted message should look like: `"<client-IP>:<client-port> <text>\n"`, for example: `"192.168.56.2:12345 Hi!\n"`. The server should not block while receiving messages from clients and messages can arrive from any client in any order.

When a client closes its connection to the server, the server should close the connection as well and free any associated resources. The server should then broadcast a message to all of the remaining clients, indicating the departure. The message should follow the following format: `"<client-IP>:<client-port> left.\n"`, for example: `"192.168.56.2:12345 left.\n"`.

The server itself should serve clients perpetually, until interrupted by SIGINT (e.g. by pressing Ctrl-C). The contents of standard output and error will be ignored during grading and can thus be used for debug purposes.

## 2.3 Build Process

The project should be built using GNU Make and the corresponding Makefile should be submitted alongside the code accordingly. Though how you organize your code and build the project is up to you, both the client and the server should be built with a simple call to `make` without any arguments. Once `make` is run in this fashion, both the `chat-client` and `chat-server` executables should be available within the same directory as the Makefile. The following Makefile does a basic build as specified above:

```
TARGETS = chat-client chat-server

CC = gcc
CFLAGS = -Wall -Werror -O3

default: $(TARGETS)

%: %.c
	$(CC) $(CFLAGS) -o $@ $<

clean:
	rm -f $(TARGETS)
```

## 2.4 Submission

Project submission will be handled online via the course Fénix website. Each group should submit a ZIP file called `project1.zip`, containing the C code and the Makefile used to build it. Please do not include any build artifacts including any object files or executables that you built yourself. The submitted files, particularly the Makefile, should be placed in the base directory within the ZIP, not within any subdirectory.

Unpacking and building the submission from the command line should look something like the following:

```
:~$ mkdir test
:~$ cd test/
:~/test$ unzip /path/to/project1.zip
```

```
Archive:  /path/to/project1.zip
  inflating: chat-client.c
  inflating: chat-server.c
  inflating: Makefile
:~/test$ ls
Makefile  chat-client.c  chat-server.c
:~/test$ make
gcc -Wall -Werror -O3 -o chat-server chat-server.c
gcc -Wall -Werror -O3 -o chat-client chat-client.c
:~/test$ ls
Makefile  chat-client.c  chat-client  chat-server.c  chat-server
```

A pre-submission check script is provided alongside this document to help check many of these submission requirements. To check your submission ZIP file, run the script in the following manner:

```
:~$ ./test-submission.sh /path/to/project1.zip
```

The script performs a sequence of basic checks, indicating for each whether the check passes or fails. If everything works out, the script should output ”All basic checks passed.”. The script performs just a small set of very basic checks and does not guarantee that your project complies with the full specification. You should still test your project thoroughly, even if it passes these basic checks.

## 3 Advice

### 3.1 Debugging

Developing a client and server that are supposed to directly interoperate can pose a challenge initially, as you can only effectively test them when both are developed to a sufficient degree. In many cases, a simple shim or mock implementation can take the place of the full fledged thing while debugging and testing. For this project there are a variety of network tools that can partially replace either the client or the server. The `telnet` tool, already introduced in class, functions in a similar fashion to the client and can be used while developing the server:

```
:~$ telnet <host> <port>
```

Another similar tool - netcat (`nc` on the command line) - can also act like a trivial TCP server:

```
:~$ nc -l <port>
```

Though these tools do not implement the full functionality of the chat system in this project, they can offer trivial endpoints to interface with during the initial stages of development.

Another challenge while developing networked software is the need for multiple machines to form a network. One way around this is the copious use of virtualization, as we have already been doing in class with `vagrant`. Although you should still test your project using these tools, this system is simple enough that it can be tested on a single host, using the loopback interface. By simply having the clients connect to `localhost`, you can run the server and multiple clients using just different terminal windows, as opposed to a more elaborate virtualized environment.

## 3.2 Task Breakdown

Although, as a team, each of the members of your group are individually and collectively responsible for the entire project, it is always a good idea to split up the project into independent pieces and allow parallel development when possible. This project lends itself to some degree of parallelism during development. Particularly, the client and server applications can be developed independently up to a point and later tested together to check for interoperability.

Within each implementation, one can break things down a step further and implement simple versions first and then add on more advanced functionality. A simple client can initially not implement non-blocking operation and alternate between blocking on reading standard input and the server socket. A subsequent version could add support for non-blocking operation with `select`. The server can also undergo multiple iterations of increasing complexity. A first prototype can handle just one client. Subsequent versions can handle multiple clients and later use `select` to send and receive data without blocking.

Finally, although quality control often ends up being an afterthought (unfortunately both in class projects and professional development), this need not and should not be the case. Efforts to produce basic testing infrastructure to automate testing your project can largely be done in parallel with the development of the main system. Having a series of automated test scripts can prove invaluable while debugging. Furthermore, having a different person write the test cases than the one who wrote the code being tested can help catch differences in interpretation of the specification and help the project converge to the correct behavior. The pre-submission check script (test-submission.sh) can also be used as the basis to build more test-cases. We encourage you to check out the script source for ideas on how to write your own tests.