CS360 Lab #A -- Threaded Chat Server

- James S. Plank
- CS360
- Url: http://web.eecs.utk.edu/~jplank/plank/classes/cs360/360/labs/Lab-9-Chat/index.html
- Directory: **/home/jplank/cs360/labs/Lab-9-Chat**

This lab is a very powerful one -- you are going to write a chat server using pthreads that allows clients to chat with each other using **nc** (or **jtelnet**). The syntax of your server is:

```
UNIX> ./chat-server port Chat-Room-Names ...
```

So, for example, if you'd like to serve chat rooms for football, bridge and politics on **hydra3** port 8005, you would do:

```
UNIX> ./chat_server hydra3.eecs.utk.edu 8005 Football Bridge Politics
```

Clients attach to the server through **nc**. Suppose, for example, we have a clients on **hydra2** and **hydra4**:

On **hydra2**:

```
UNIX> nc hydra3.eecs.utk.edu 8005
Chat Rooms:

Bridge:
Football:
Politics:

Enter your chat name (no spaces):
Dr-Plank
Enter chat room:
Bridge
Dr-Plank has joined
There's no one here...
Dr-Plank: There's no one here...









Goofus has joined
Hi Goofus -- do you like bridge?
Dr-Plank: Hi Goofus -- do you like bridge?

Goofus: Bridge? You mean that card game my gramma plays?
Indeed
Dr-Plank: Indeed

Goofus: Loser.  Bye.

Goofus has left
Can't say I liked him.
Dr-Plank: Can't say I liked him.









Gallant has joined

Gallant: Hi Dr. P
Greetings, Gallant
Dr-Plank: Greetings, Gallant

Gallant: After memorizing your lecture notes,

Gallant: I like to read books on bridge.
I will recommend you for many jobs & scholarships.
```

On **hydra4**:

```
UNIX> nc hydra3.eecs.utk.edu 8005
Chat Rooms:

Bridge: Dr-Plank
Football:
Politics:

Enter your chat name (no spaces):
Goofus
Enter chat room:
Bridge
Goofus has joined

Dr-Plank: Hi Goofus -- do you like bridge?
Bridge? You mean that card game my gramma plays?
Goofus: Bridge? You mean that card game my gramma plays?

Dr-Plank: Indeed
Loser.  Bye.
Goofus: Loser.  Bye.
<CNTL-D>
UNIX>

UNIX> nc  hydra3.eecs.utk.edu 8005
Chat Rooms:

Bridge: Dr-Plank
Football:
Politics:

Enter your chat name (no spaces):
Gallant
Enter chat room:
Bridge
Gallant has joined
Hi Dr. P
Gallant: Hi Dr. P

Dr-Plank: Greetings, Gallant
After memorizing your lecture notes,
Gallant: After memorizing your lecture notes,
I like to read books on bridge.
Gallant: I like to read books on bridge.
```

```
Dr-Plank: I will recommend you for many jobs & scholarships.
<CNTL-D>
UNIX>
```

```
Dr-Plank: I will recommend you for many jobs & scholarships.

Dr-Plank has left
I didn't get a chance to be more sycophantic!
Gallant: I didn't get a chance to be more sycophantic!
<CNTL-D>
UNIX>
```

To be descriptive, when a client joins, the server sends it information about the current chat rooms. The chat room names will be listed lexicographically, and the name of each person chatting should be listed with each chat room, separated by a space. The order of that listing should be the order in which the chatters joined.

The server then prompts the client for a name and then a chat room. Obviously it should error check (including premature EOF). Once the person joins the chat room, a line is sent to all others in the chat room that the person has joined. Lines entered by the clients are sent to all the clients in the chat room.

The server should support *any* number of clients, and should work seamlessly when clients leave, as Goofus, and later Dr-Plank did above. The server does not have to print any output, but it may -- I will not be testing what the server prints -- I will only test the behavior of the clients.

Structure

This lab is involved, and will use pthreads, mutexes and condition variables. Suppose there are *r* chat rooms and *c* clients. Then your **chat_server** will have *r+c+1* threads. Specifically:

- There will be one thread that is spinning on a **while()** loop, waiting for clients to attach to the socket. When it detects a client, it will create a client thread.
- There will be one thread for each chat room. That thread will typically be blocked on a condition variable (unique to that chat room). When the condition variable is unblocked, that is the indication that the server has received input from a client. That input will be on a list. For each string on the list, the server thread should traverse all the clients and send the string to each client. When it is done processing the list, it should wait on the condition variable again.
- There will be one thread for each client. That thread will typically be blocked reading from the socket. When it receives a line of text from the socket, it will construct the proper string from it, put that string onto the chat room's list, and signal the chat room server.

Remember to protect data structures when you have to. For example, the clients and servers share the chat room's lists. When the clients update the list and when the servers read the list and delete entries, those operations must be protected by a mutex.

You should use **fdopen()** twice on each connection. The client threads will call **fgets()** and **fputs()** on these stdio buffers initially until the client's name/chat-room have been obtained. After that, the client threads only call **fgets()** and the chat room threads call **fputs()** (and **fflush()**).

A subtle part of this lab is to deal with clients exiting *at any point*. That means you have to test the return values of all **fputs()**, **fgets()** and **fflush()** calls and deal with them appropriately. I dealt with them as follows:

- If I catch an EOF while reading the client's info, I simply close the buffers and kill the client thread.
- After the client has joined the chat room, then if I detect EOF on a **fgets()** call, then I close the input buffer. If the output buffer is still open and if the chat room thread is not currently trying to write to it, then I close the output buffer and remove the client from the chat room's list. Then I kill the client thread.
- If the chat room thread detects a problem on **fputs()** or **fflush()**, then I remove the client from the chat room's list and close the output buffer. I don't mess with the client thread, since it should detect EOF on the **fgets()** call and exit on its own.

You may want to draw yourself some pictures to help visualize the interactions between the client threads and the chat room threads.

Chatty_chat_server

In the lab directory, there is an executable called **chatty_chat_server**. It is identical to **chat_server**, except it prints out thread creating and exiting, plus mutex and cv actions. It gives a little more information as well. You may find it helpful to see how it works when you implement your own synchronization.

The Gradescript

The gradescript here is different. It assumes that you are running your **chat_server** on another machine. You should run the chat server as follows:

```
chat_server port Bridge Baseball Politics Video-Games Art Music Movies Food Woodworking American-Idol
```

Make sure to use a port number that is greater than or equal to 8000. And please make sure that the client and server are both on our lab machines (hydra, tesla).

You run the gradescript with three arguments:

```
gradescript number host port
```

The host and port are of your **chat_server**. Gradescript will run the program **laba-tester**, which opens a number of client connections, sends lines and tests the output. Since your server should be able to handle clients coming and going, you shouldn't have to start and stop your client between runs of **gradescript** -- just start it once and that should suffice for all **gradescript** runs.

Now, a little detail on the internals. **Gradescript** runs a program called **laba-tester**, which should be called with the same arguments as **gradescript**. You should use **laba-tester** to help develop your server. Let's take an easy example. My server is running on **hydra3**, port 8008:

```
UNIX> ./laba-tester 1 hydra3.eecs.utk.edu 8008
Event in Chat Room Art: Fiona has joined
Read Event From Client Fiona: Fiona has joined
Event in Chat Room Art: Mercutio has joined
Read Event From Client Fiona: Mercutio has joined
Read Event From Client Mercutio: Mercutio has joined
Event in Chat Room Art: Fiona has left
Read Event From Client Mercutio: Fiona has left
Event in Chat Room Art: Mercutio has left
Events correctly processed
UNIX>
```

There are three kinds of events that **laba-tester** will generate:

- Client joins a room.
- Client leaves a room.
- Client writes a string.

When a client *c* joins a room *r*, you see the string "Event in Chat Room *r*: *c* has joined". Each client *c2* attached to that room should receive a string saying the client has joined. **laba-tester** tests each of these, and prints out the string "Read Event From Client *c2*: *c* has joined".

The printout when clients leave is similar -- when they leave, you get "Event in Chat Room *r*: *c* has left", and then each client still attached to that room should get "Read Event From Client *c2*: *c* has left".

As you can see, in the above example, clients Fiona and Mercutio join the chat room "Art." Then Fiona leaves, and then Mercutio leaves. Test cases 1-5 just test entering and leaving.

Let's look at a more complicated one:

```
UNIX> ./laba-tester 7 hydra3.eecs.utk.edu 8008
Event in Chat Room American-Idol: Waluigi has joined
Read Event From Client Waluigi: Waluigi has joined
Event in Chat Room American-Idol: Tito has joined
Read Event From Client Waluigi: Tito has joined
Read Event From Client Tito: Tito has joined
Write Event in Chat Room American-Idol: Waluigi: Papa's on the corner waitin' for the bus
Read Event Client Tito, line: Waluigi: Papa's on the corner waitin' for the bus
Read Event Client Waluigi, line: Waluigi: Papa's on the corner waitin' for the bus
Event in Chat Room American-Idol: Waluigi has left
Read Event From Client Tito: Waluigi has left
Event in Chat Room American-Idol: Tito has left
Events correctly processed
UNIX>
```

Again we have two clients, Waluigi and Tito, and we are using one chat room: "American-Idol." After the two clients join, Waluigi writes "Papa's on the corner waitin' for the bus". Both clients read the line successfully, and then they exit.

If you want to see the order of events, look at **tmp-inputfile.txt**:

```
UNIX> cat tmp-inputfile.txt
START Waluigi American-Idol
START Tito American-Idol
Waluigi: Papa's on the corner waitin' for the bus
END Waluigi
END Tito
UNIX>
```

And if you want to see the output of each client as it came from the server, look in **output-*client*.txt**:

```
UNIX> cat output-Waluigi.txt
Chat Rooms:

American-Idol:
Art:
Baseball:
Bridge:
Food:
Movies:
Music:
Politics:
Video-Games:
Woodworking:

Enter your chat name (no spaces):
Enter chat room:
Waluigi has joined
Tito has joined
Waluigi: Papa's on the corner waitin' for the bus
UNIX> cat output-Tito.txt
Chat Rooms:

American-Idol: Waluigi
Art:
Baseball:
Bridge:
Food:
Movies:
Music:
Politics:
Video-Games:
Woodworking:

Enter your chat name (no spaces):
Enter chat room:
Tito has joined
Waluigi: Papa's on the corner waitin' for the bus
Waluigi has left
UNIX>
```

The test cases are as follows:

- 1-5: 2 to 10 clients joining and exiting.
- 6-10: 2 clients, one room, one line of text.
- 11-15: 3 to 12 clients, one room, one line of text.
- 16-20: 2 to 12 clients, one room, two lines of text.
- 21-25: 3 to 12 clients, two rooms, two lines of text.
- 26-30: 3 to 12 clients, one room, one line of text per client.
- 31-35: 3 to 12 clients, two rooms, one line of text per client.
- 36-40: 3 to 12 clients, three rooms, one line of text per client.
- 41-100: 4 to 24 clients, four to ten rooms, 30 to 259 lines of text.

If you call **laba-tester** with a number greater than 100, it will choose a random test case with the same distribution as above.

Finally, when you get to the later test cases, you will see some complex behavior. When the tester writes lines of text to the server, it does not read them from the clients until one of the following:

- A joining event occurs.
- A leaving event occurs.
- More than 10 consecutive talking events have occurred.

At that point, it reads from all the clients, and double-checks the correctness and ordering of the output. It is entirely possible for the output order to differ from the input order, because the order is going to depend on which threads return from their **fgets()** calls in your server. Let's look at an example:

```
UNIX> laba-tester 26 hydra3.eecs.utk.edu 8008
Event in Chat Room Video-Games: Tito has joined
Read Event From Client Tito: Tito has joined
Write Event in Chat Room Video-Games: Tito: Ah, sloppy Sue and Big Bones Billie, they'll be comin' up for air
Read Event Client Tito, line: Tito: Ah, sloppy Sue and Big Bones Billie, they'll be comin' up for air
Event in Chat Room Video-Games: Tinky-Winky has joined
Read Event From Client Tito: Tinky-Winky has joined
Read Event From Client Tinky-Winky: Tinky-Winky has joined
Event in Chat Room Video-Games: Thor has joined
Read Event From Client Tito: Thor has joined
Read Event From Client Tinky-Winky: Thor has joined
Read Event From Client Thor: Thor has joined
```

```
Write Event in Chat Room Video-Games: Tinky-Winky: There's no escape, I can't wait
Write Event in Chat Room Video-Games: Thor: To do what was right
Read Event Client Thor, line: Thor: To do what was right
Read Event Client Thor, line: Tinky-Winky: There's no escape, I can't wait
Read Event Client Tinky-Winky, line: Thor: To do what was right
Read Event Client Tinky-Winky, line: Tinky-Winky: There's no escape, I can't wait
Read Event Client Tito, line: Thor: To do what was right
Read Event Client Tito, line: Tinky-Winky: There's no escape, I can't wait
...
```

Note the order of the writing events: Tinky-Winky writes "There's no escape, I can't wait", and then Thor writes "To do what was right". However, the talk-server's thread for Thor got its line before the thread for Tinky-Winky, and so each of the three clients reads Thor's line before Tinky-Winky's. That is fine, and your output does not have to match mine exactly, since mine may differ from run to run. However, each reading client has to receive the events in the same order relative to each other. The testing program tests to make sure this happens, so it will approve the output above. If, for example, Tito had printed out Tinky-Winky's line first while the other two printed out Thor's line first, the testing program would flag it as an error.

A final word about gradeall

The **gradeall** script runs all 100 **gradescript** tests, all with the same server. So, if your server has a massive memory leak (made, for example, because you don't clean up your threads when they die), then it may start to fail on the later gradescripts. This may confuse you: If you run gradescript 72 with a fresh version of the server, it works fine; however, when you run **gradeall**, it fails on gradescript 72. The reason is that your server has been leaking memory by not cleaning up its threads, and eventually **pthread_create()** starts failing, or maybe even **malloc()** starts failing. Pay attention to this -- make sure you test the return values of your calls so that you can figure out failures when they happen.