CIS 5050: Software Systems

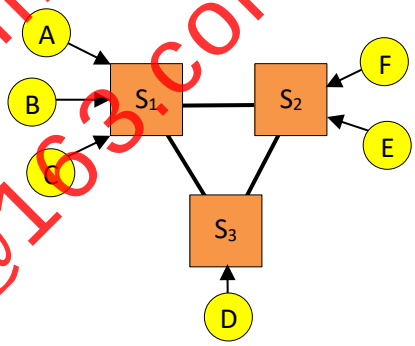
Spring 2024

Assignment 3: Chat server

Due on April 2, 2024, at 10:00pm EDT

1 Overview

For this assignment, you will implement a simple replicated chat server that uses multicast to distribute the chat messages to the different replicas. The framework code is available from the course website at: https://www.cis.upenn.edu/~cis5050/handouts/hw3.zip



The figure on the right shows a sketch of the system you will build. There are two kinds of processes: servers (represented by orange squares) and clients (represented by yellow circles). Each client is connected to *one* of the servers, and each server is connected to *all* the other servers. The clients and servers can all run on different machines, or (e.g., for testing) they can all run on the same machine, e.g., in your VM.

The system will provide different chat rooms. Client can join one of the chat rooms and then post messages, which should be delivered to all the other clients that are currently in the same chat room. If there was only a single server, this would be easy: the server would simply send each message it receives to all the clients who are currently in the relevant room. However, since there can be multiple servers, this simple approach does not work. Instead, the servers will internally use multicast to deliver the messages.

To make this a little more concrete, consider the following example. In the figure, client D is connected to server S_3. Suppose D joins chat room #3, which already contains A and B, and posts a message there (say, "Hello"). S_3 then multicasts D's message to multicast group #3, so it will be delivered to the other servers. (All servers are members of all multicast groups.) The servers then forward D's message to all of the clients that are currently in chat room #3. In this case, S_3 will echo it back to D, and S_1 will send it to A and B. S_2 does not currently have any clients that are in chat room #3, so it ignores the message.

The servers should support three different ordering modes that we studied in class:

- **Unordered multicast:** Each server delivers the messages to its clients in whatever order the server happens to receive it;
- **FIFO multicast:** The messages from a given client C should be delivered to the other clients in the order they were sent by C; and
- **Totally ordered multicast:** The messages in a given chat room should be delivered to all the clients in that room in the same order.

Your implementation should be fully distributed; in other words, there should not be any central 'sequencing nodes'. All communication, both between clients and servers and among the servers, should be via *unicast* UDP using datagram sockets (and not using the OS's built-in multicast primitive). You may assume that there is no message loss, and that the messages between servers and clients are not reordered (but messages between servers are typically out-of-order).

2 Specification

2.1 The client

Your first task is to implement a simple chat client. The user should be able to invoke the client from the command line with the IP address and port (i.e., the bind address) of a particular server as a parameter. Note that each client is connected to exactly one server. The user should then be able to type lines of text (commands or chat messages), and, whenever the user has entered a new line, the chat client should send the line to the server in a single UDP packet (without any trailing <CR> or <LF> or <CRLF>). The client should also listen for UDP packets from the server; whenever it receives such a packet, it should print its contents to the terminal. You may also want to add a -v option to the client program for your own debugging purposes (what to print for the client's debug output is entirely up to you).

2.2 Commands

When the server receives a line of text from the user, it should check whether it begins with a forward slash (`/`). If not, the server should consider the line to be a chat message; otherwise, it should treat it as a command. The following commands should be supported, and all of them should be case sensitive.

- `/join <roomID>`: Used to join a particular chat room (Example: `/join 7`). The roomID should be an integer in the set {1, 2, …, N}, where N is the total number of chat rooms. Your implementation should support *at least* N = 10 chat rooms (i.e., with room IDs 1,2,3,...,10). If the command succeeds, the server should respond with a line that starts with `+OK` (Example: `+OK You are now in chat room #7`). If the user tries to join more than one chat room at a time, or if the specified chat room does not exist (e.g., the specified room is outside your supported range of room IDs), the server should respond with a line that starts with `-ERR` (Example: `-ERR You are already in room #9`).
- `/part`: Used to leave the current chat room. If successful, the server sends a response that starts with `+OK` (Example: `+OK You have left chat room #7`). If the user is not currently in a chat room, the server sends a response that starts with `-ERR`.
- `/nick <nickname>`: Used to set the user's nickname (Example: `/nick Linh`). The server should respond with a line that starts with `+OK`. If the user does not explicitly set a nickname, the server should use its IP and port (Example: `127.0.0.1:5005`). If the `/nick` command is used more than once, the server should use the latest nickname. The `/nick` command can be issued at any time (including even before the user joins a chatroom), and it should persist across chatrooms. You may assume that `<nickname>` can be any non-empty sequence of ASCII alphanumeric characters. It is case sensitive, and, for ease of implementation, does not need to be unique (e.g., it is okay to have two users with the same nickname).

- `/quit`: When the user enters this special command, the chat client should send the command to the server and then terminate. The server should remove the chat client from its list of active clients. The `/quit` command can be sent by the user at any time.

If the user sends a text message without having joined a chat room, the server should respond with a line that starts with `-ERR`. To make your implementation easier, each client cannot be in more than one chat room at a time. You may assume that each text message sent by the user is at most 1000 bytes. If the user terminates the client using CTRL+C, the client should behave in the same way as if it has received a `/quit` command. If the user enters an invalid command, i.e., the text begins with "/" but does not fall into the above commands, the server should respond with a line that starts with `-ERR`.

Below is an example session (the lines in bold are typed by the user):

```
$ ./chatclient 127.0.0.1:5000
/join 3
+OK You are now in chat room #3
Hello everyone!
<127.0.0.1:57326> Hello everyone!
<Alice> Hi there! Do you want to set a nickname?
/nick Bob
+OK Nickname set to 'Bob'
Is this better?
<Bob> Is this better?
<Alice> Yes, much better!
/join 5
-ERR You are already in room #3
/part
+OK You have left chat room #3
/quit
```

2.3 The server

Often, multicast systems support dynamic membership: nodes can join and leave the system at runtime. However, to simplify the assignment a bit, we will implement only static membership: each server will have access to a configuration file that contains the addresses of all the servers in the system.

Each server will have two – potentially different – addresses. The first is the *forwarding address*: this is the IP and port number that other servers should use as the destination address when they want to send a message to this server. The second is the *bind address*: this is the IP and port number on which the server listens for UDP messages from clients and from other servers; it is the address that the server should bind to (and that its clients should connect to). Usually, the two addresses are the same. However, for debugging, it is useful to have the ability to make them different; for more details, see Section 3.2.

Your server program should require two command-line arguments. The first is the name of a configuration file that contains the forwarding address and the bind address of *all* the server instances, and the second is

the index of the current server instance in this list. The file should contain one line for each server instance. The configuration file follows one of the two formats, depending on whether the proxy is used or not (the proxy will be described later in Section 3.2):

- If the proxy is used: Each line should contain two IPs and port numbers, separated by a comma (Example: `127.0.0.1:8001,127.0.0.1:5001`); the first address is the forwarding address and the second is the bind address.
- If the proxy is not used: Each line of the configuration file should contain only one IP address and port number (Example: `127.0.0.1:5000`); this address serves as both the forwarding address and the bind address for that server.

The index of the first server should be 1. For instance, suppose the file `foo.txt` contains the following lines (notice that the proxy is used in this example because each line contains two addresses):

```
127.0.0.1:8000,127.0.0.1:5000
127.0.0.1:8001,127.0.0.1:5001
127.0.0.1:8002,127.0.0.1:5002
```

and the server is invoked like this: `chatserver foo.txt 2`, then the server would listen for (all) messages on its bind address, i.e., on `127.0.0.1:5001`. When it has a message for the third server, it would send the message to the third server's forwarding address, which is `127.0.0.1:8002`; when it has a message for the first server, it sends the message to the first server's forwarding address, which is `127.0.0.1:8000`. In addition, the user would invoke a client of the server using the server's bind address – e.g., a client of the second server can be invoked like this: `chatclient 127.0.0.1:5001`.

The server should also accept two options (`-v` and `-o`) on the command line:

- When the `-v` option is present, the server should print a line to `stdout` when something important happens. (This is meant for debugging, so it is up to you to decide what qualifies as important.) Each such line should start with a prefix of the form `hh:mm:ss.uuuuuu Snn`, where `hh:mm:ss.uuuuuu` is the current UTC time in hours, minutes, seconds, and microseconds, and `nn` is the server index from the command line. For instance, the server might print `03:48:22.004328 S02 Client 17 posts "Hi" to chat room #3`. To obtain the current wall-clock time, you can use functions such as gettimeofday() or clock_gettime(), and convert the obtained values to the correct format as shown above.
- The `-o` option is used to select an ordering mode: choices are `-o unordered` (the default), `-o fifo`, and `-o total`. You may assume that, when `-o` option is present, all servers will always use the same ordering mode. In addition, you may assume that *all* the servers will be running at the same time (before invoking the clients) and that they do not fail.

When no arguments are given on the command line at all, the server should print your name and SEAS login and then terminate. If the server is terminated using CTRL+C, it should close all connections and exit. When a client posts a message to a chat room, the server should prepend "<xxx> " to the message, where `xxx` is the client's nickname, just like in the example transcript above. Your implementation should be able to support at least 10 servers, with up to 250 clients in total.

3 Suggestions

The following section contains some hints that we thought you might find helpful as you implement, test, and debug your solution. None of this is mandatory – feel free to use your own approach if you prefer!

3.1 Implementation hints

The client is a good place to begin. The slide deck about socket programming already contains some code for a simple UDP client, so you could start with that. However, notice that the chat client needs to listen not just for messages from the server but *also* for user input. One simple way to implement this is with the `select` system call: you can use this to block until the UDP socket or `stdin` is ready for reading. The file descriptor number for `stdin` is `STDIN_FILENO`; once this descriptor becomes ready for reading, you can use `read` from it as usual, and you should receive a line of user input.

You can initially test your client with a simple UDP echo server, similar to the one on the slides. Once the echo is working, you can modify the echo server by adding code for reading the file with the server addresses, and forward incoming client messages to the other servers, who can forward them to their own clients. (Remember: This should be done using unicast UDP via datagram sockets, not using any kernel's built-in multicast primitive!) The result is, effectively, a simple unordered multicast with only one group: a message sent by any client will be forwarded to all the clients.

Once the above is working, a good next step would be to add support for `/join`, `/part`, etc. Notice that, once you have support for groups, the servers cannot simply forward the messages they receive; they need to attach some additional information, such as the group to which the message belongs. We recommend that you use a text format for your messages: for instance, if one server wants to tell another about a `Hello` message in chat room #5, it could send something like `5,Hello`. This makes it easy to print messages into the log (with the `-v` option) and to inspect them when something goes wrong.

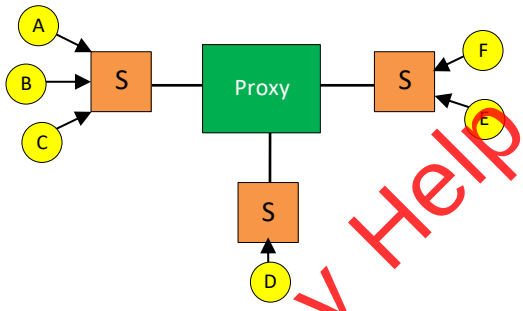
At this point, the big missing pieces are FIFO and total ordering. These will require a holdback queue, as we have discussed in class. Start with FIFO ordering; this will probably require extending your inter-server messages with even more information. Notice that it isn't completely trivial to test whether the ordering is working properly, so you may want to use the proxy for testing (see Section 3.2).

The final piece of the puzzle is total ordering. This will require some more complicated messaging between the servers; the algorithm is on the slides of the Group Communication lecture (the distributed version). Don't forget to include a proper tie breaker! The IP address isn't enough for this: for instance, if you test your server in the VM, all instances will have the same IP address. Using the server's IP + port number for the nodeID (as tie breaker, see the distributed total algorithm on the slides) should work.

3.2 Testing your solution

Testing a system like this is a little tricky. You'll probably want to create a few servers and clients within your VM (they could each run in a separate terminal window, for instance). However, within the VM, network messages are delivered almost instantaneously; hence, by the time you enter the next message, the previous one will already have been delivered to all the clients. Thus, it is hard to produce 'interesting' test cases where several messages are in flight at the same time.

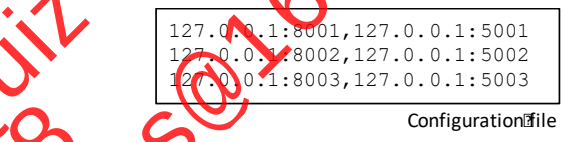
To help you with testing, we have provided two tools in the `test/` subdirectory. The first is called `proxy.cc`. You can insert the proxy between the servers, as shown in the illustration above; it will receive all the messages that the servers send to each other, and insert a random delay, which can be configured (in microseconds) with the `-d` option. If you make the delay large enough (say, a few seconds), the messages should frequently arrive out of order, which allows you to properly test all the interesting cases.

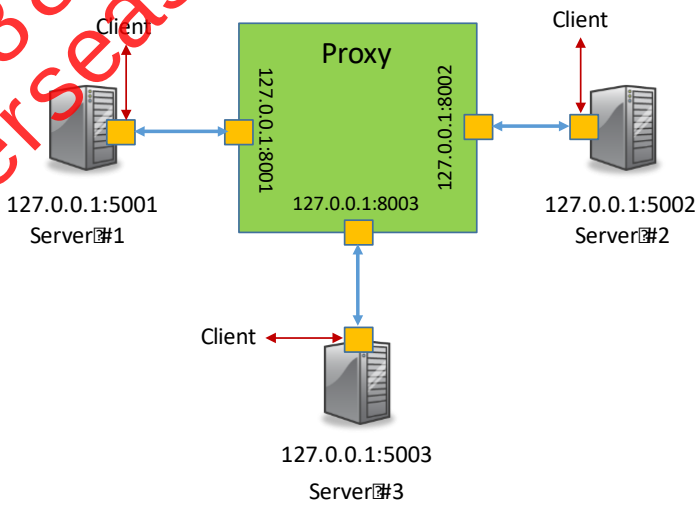


For the proxy to work, the forwarding address and the bind address must be different: the servers themselves will listen on the bind addresses, while the proxy listens on the forwarding addresses and then forwards any messages it receives to the bind address of the destination server. When you use the proxy, create an address file that contains two different addresses on each line (it can be `127.0.0.1` with different port numbers) and give the same file to both the servers and the proxy. Please note that, if you do not use the proxy, the forwarding address of each server must be the same as its bind address; hence, the address file should contain a single address for each server in each line.

An illustration of using the proxy:

The figure on the right shows a simple configuration file with three servers, as well as the communication pattern that should be used in this case. Notice how each server only has one socket, which is bound to the bind address, and how the proxy has a socket for each of the servers, which is bound to the forwarding address.





Suppose server #1 wants to send a message to server #2. Then the following should happen: Server #1 sends the message to server #2's forwarding address. The resulting UDP packet has source address 127.0.0.1:5001 and destination address 127.0.0.1:8002, since it is going to server #2. (Don't let the arrows in the picture confuse you - each server can send messages to each of the proxy's ports!). The message is received by the proxy on port 127.0.0.1:8002. Since this is the forwarding address of server #2, the proxy knows that this message was supposed to go to that server. Also, since the source address is server #1's bind address, the proxy knows that the message came from server #1. The proxy forwards the message to server #2, using the socket that is bound to server #1's forwarding address (127.0.0.1:8001). The resulting UDP packet has source address 127.0.0.1:8001 and destination address 127.0.0.1:5002. (Again, don't let the arrows in the picture confuse you!). The message is received by server #2. Since its source address is server #1's forwarding address, server #2 knows that this is where the message (originally) came from.

Please note that the complete functionality of the proxy is already implemented in the provided `proxy.cc`. When you want to test your implementation with the proxy, all you need to do is to invoke it together with your servers for testing. The proxy takes as input the same configuration file as the server does. The configuration file must contain two addresses—forwarding address and bind address—per line, and these addresses must be different. (An example of the configuration file when using the proxy is shown in the illustration above.)

In contrast, when you test your implementation without the proxy, the configuration file should contain only one address per line, which serves as both the forwarding address and the bind address for the corresponding server.

3.3 Testing your solution with the tester

Another challenge with testing comes from the fact that it is tedious to manually send lots of test messages, and that it can be difficult to see whether a long string of messages was ordered correctly. This is where the second tool, `stresstest.cc` (provided in the `test/` subdirectory) comes in.

At a high level, the stress tester is a single program that can emulate multiple clients. For instance, when you run it like this:

```
./stresstest -o fifo -m 20 -c 5 -g 2 servers.txt
```

it will emulate five clients (`-c`) that each join one of two (`-g`) chat rooms and then post a total of 20 messages (`-m`). It will read the list of servers from '`servers.txt`'; the format is the same as in the configuration file you give to the server processes. The stress tester will examine the messages it receives from your servers, and it can check whether they are ordered correctly; you can specify an order with the `-o` option, which works just like the `-o` option on the servers. (When you use `-o unordered`, the stress tester only checks whether each client receives all of the messages that are being posted.) If something goes wrong, you may want to re-run the stress tester with the `-v` option, which produces a much more verbose output. The tester should be used in conjunction with the proxy. (As explained above, without the proxy, the communication delay will be so low that you rarely get message reordering; as a result, the messages will naturally appear to be ordered, even if your implementation does not work correctly.) Note that the tester only waits for inflight messages for a certain amount of time (5 seconds, by default). You can use the `-f` option to specify the amount of time (in seconds) it should wait for messages. (In general, the timeout should be at least more than the total communication delay under the ordering algorithm you are implementing, but it should not be too large to still be useful in detecting subtle bugs.) Please make sure you have at least tested your solution with the stress tester before submitting.

3.4 Debugging hints

Debugging truly distributed systems, such as this one, can be tricky because many things are happening concurrently on different nodes. The `-v` option is intended to help you with this. If you encounter a problem (say, some messages are not being delivered, or the proposed sequence numbers look wrong), you can use `-v` on all the servers and redirect the output of each server to a file. Then you can `cat` the files together and `sort` them. Because each line starts with a timestamp and a node ID, this will result in a kind of chronicle that shows all the events on the different servers in the order in which they occurred.

For this to work well, it is important to produce sufficiently many (and sufficiently detailed) log messages. Also, it is usually a good idea to include periodic 'dumps' of key data structures, such as the holdback queue, so you can see the status of the various messages at different points in time.

4 Submitting your solution

Before you submit your solution, please make sure that:

- ☐ Your solution compiles properly.
- ☐ Your code contains a reasonable amount of useful documentation (required for style points).
- ☐ You have completed *all* the fields in the `README` file.
- ☐ You have checked your final code into the Git repository *before* you submit.
- ☐ You are submitting a `.zip` file (e.g., one created by running `make pack` from the `HW3` folder)
- ☐ Your `.zip` file contains all of the following:
  - ☐ all the files needed to compile your solution (especially all `.cc` files!);
  - ☐ a working Makefile; and
  - ☐ the `README` file, with all fields completed
- ☐ Please do not submit binaries or input files!

As with the previous assignments, you must submit your `.zip` file online via GradeScope – we will not accept submissions via email, or in any other way. Please note that this is the last assignment that you can use jokers (jokers do not apply to the final project).

5 Grading

The total score for this assignment is 100 points. The breakdown are as follows:

- Style points (README, documentation, etc.): 6 points.
- Chat client: 10 points.
- Basic server and multicast functionalities: 40 points.
- FIFO ordering: 20 points.
- Total ordering: 24 points.

5 Extra credit

5.1 Causal ordering (+15%)

For this extra-credit task, you should add a fourth ordering mode: causal ordering. As with the existing three modes, your implementation must be fully distributed – for instance, you may not use a central coordinator. Your server should support an `-o causal` option to enable this mode. Since this is the extra credit, you will need to write test cases on your own; however, at the very least, your solution should pass the FIFO test, which you can use our provided tester for checking.

5.2 Lossy links (+5%)

For this extra-credit task, you should add a way to handle message losses on the client-server links as well as on the server-server links. Lost messages should be retransmitted, but you may assume that the network remains connected, and that a message will eventually be delivered if it is retransmitted sufficiently often. The `proxy.cc` tool already supports a command-line option `-l` that causes the proxy to drop a certain fraction of the messages it forwards; this should be useful for testing.