Assignment 5: Chat Server

Note: Assignment 5 is a double assignment. Each milestone (MS1 and MS2) is worth 1/6 of the assignments grade for the course, the same as (individually) Assignments 1–4.

Due:

- Milestone 1 due **Mon Nov 17th** by 11pm
- Milestone 2 due **Fri Dec 5th** by 11pm (no late hours allowed)

Note that you may **not** use late hours on Milestone 2. Please plan accordingly.

Grading Criteria

Milestone 1:

- Implementation of sender client: 22.5%
- Implementation of receiver client 22.5%
- Design and coding style: 5%

Milestone 2:

- Implementation of server: 30%
- Report explaining thread synchronization in server: 15%
- Design and coding style: 5%

Table of contents

Overview

Download csf_assign05.zip and unzip it.

In this assignment, you will develop a chat client program that communicates synchronously with a server in real-time. You may think of this as an implementation inspired by classical chat systems such as IRC.

Note: We highly recommend that you use C++ for this assignment. The provided skeleton code includes partially implemented classes which we encourage you to use as the basis for your client and server implementations.

Goals of the assignment

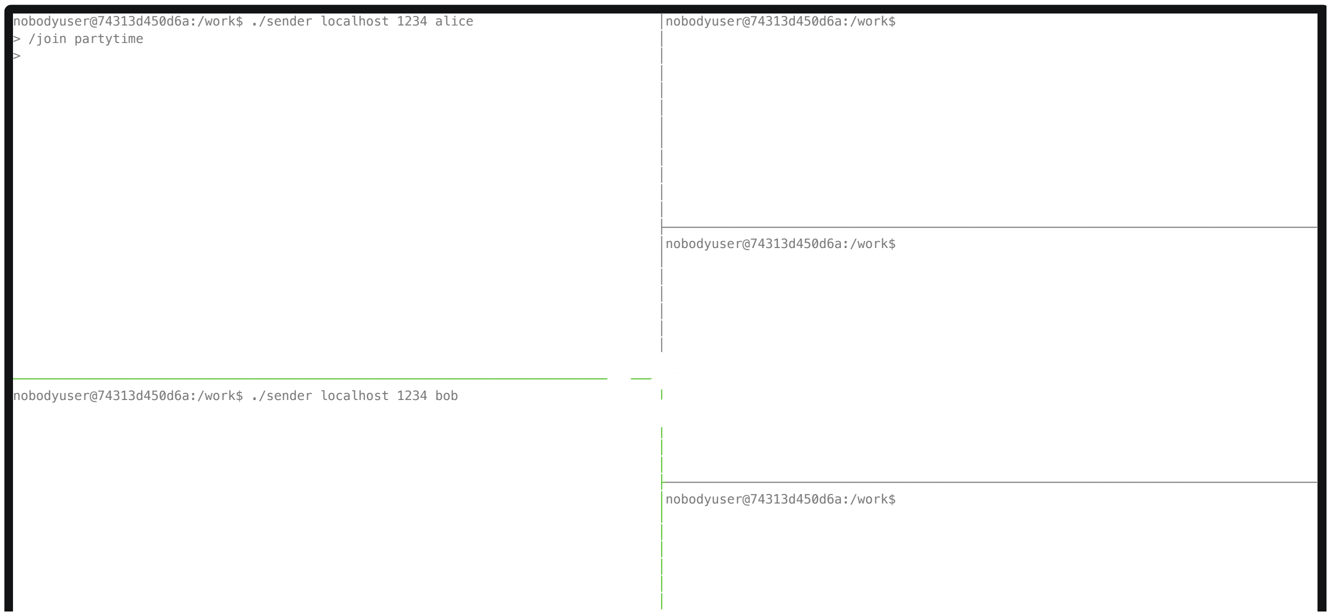
The main goal of the assignment is to provide an opportunity to create a network application.

Although this will be a relatively simple program, it is representative of a larger class of network-enabled systems:

- It will have a protocol for communication between clients and server
- it will allow communication over a network (specifically by accepting TCP connections from clients)
- It will use concurrency and synchronization primitives to coordinate access to shared data on a remote server

Demo

Here is an example chat session with two different senders and three receivers, all connected to the same server:



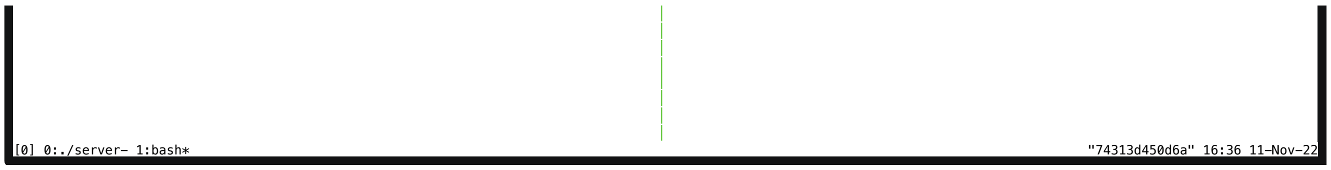
```
nobodyuser@74313d450d6a:/work$ ./sender localhost 1234 alice
> /join partytime
>
```

```
nobodyuser@74313d450d6a:/work$
```

```
nobodyuser@74313d450d6a:/work$
```

```
nobodyuser@74313d450d6a:/work$ ./sender localhost 1234 bob
```

```
nobodyuser@74313d450d6a:/work$
```



```
[0] 0:./server- 1:bash*                                    "74313d450d6a" 16:36 11-Nov-22
```

(thanks asciinema for the wonderful terminal recording widget!)

The Protocol

The client and server communicate by exchanging a series of *messages* over a TCP connection. There are two kinds of clients: a *receiver* which is used to only read messages from the server, and a *sender* that is used to send messages to the server. To allow multiple groups of people to talk independently, the server partitions clients into "rooms". All receivers in the same room will receive that same set of message, and all senders in the same room will broadcast to the same receivers.

A *message* is an ASCII-encoded transmission with the following format:

```
tag:payload
```

A *message* is subject to the following restrictions:

- A message must be a single line of text with no newline characters contained within it.
- A message ends at a newline terminator which can be either `"\n"` or `"\r\n"`. Your programs must be able to accept both of these as newline delimiters.
- The `tag` must be one of the operations specified in the "tag table".
- The payload is an arbitrary sequence of characters. If a tag has a structured payload, the payload must be formatted exactly as specified.
- If a tag has a payload that is ignored (e.g., the "quit" and "leave" tags), the tag/payload separator character `:` must still be present (e.g. `quit:` not `quit`), even if the payload is empty
- An encoded message must not be more than `MAX_LEN` bytes.

The first message sent to the server by a client is considered a login message, and must have one of the following tags:

- `slogin`
- `rlogin`

These commands allow the client to log in to the server with the specified usernames. `slogin` is for a sender, and `rlogin` is for a receiver. A receiver terminates its connection by simply closing its socket. The server will automatically detect when this happens by looking a send failure on the next message sent to the client.

If a client logs in with `slogin`, from that point forwards, it is a synchronous protocol. The client sends a message, and the server sends a response, indicating the status of the request.

The following message types are defined:

Tag	Sent by	Payload content/format	Description
err	server	message_text	client's request was not carried out.
ok	server	message_text	client's request ran to completion.

Tag	Sent by	Payload content/format	Description
delivery	server	room:sender:message_text	a delivery of a received message to a receiver.
slogin	sender	username	log in as sender.
rlogin	receiver	username	log in as receiver.
join	sender/receiver	room_name	client wants to join specified room (which will be created as necessary). Client leaves the current room if applicable.
leave	sender	[ignored]	the sender sends this command to leave the chat room they are currently in
sendall	sender	message_text	send a message to all users in room
quit	sender	[ignored]	client is done, server will close the connection.

You may have the following assumptions about the usernames and room names we test your programs on:

- They will be at least one character in length
- They will contain only letters (`a`-`z` or `A`-`Z`) or digits (`0`-`9`)

The reference server implementation will reject operations in which the username and/or room name do not meet these criteria.

Assignment skeleton

We have included a reasonably comprehensive assignment skeleton in the starter code to help you factor your design into manageable parts. You are free to change any part of the design, up to and including writing your assignment from scratch, so long as your program follows all semantics of the reference executables.

If you elect to change the skeleton code or the Makefile, ensure that you build executables with the sames names. Exercise extreme caution if you change our synchronization architecture to avoid introducing issues.

Here is a description of the files included in the starter code:

- `client_util.{h,cpp}` - contain utility functions that are shared between the send client and receive client.
- `connection.{h,cpp}` - class describing a connection between a client and server. Used by both the receiver, the sender, and the server.
- `csapp.{h,c}` - functions from the CS:APP3e book. You are free to modify functions here as needed, e.g. adding const qualifiers for const correctness, but be careful if you don't completely understand the function you're changing!
- `guard.h` - RAII style block-scoped lock. Creating the object acquires the lock, destroying the object (i.e. when it goes out of scope) releases the lock.
- `message.h` - class representing the protocol message format.
- `receiver.cpp` - contains the main function for the receiver.
- `room.{cpp,h}` - room class used by the server.
- `sender.cpp` - contains the main function used by the sender.
- `server.{cpp,h}` - server class that tracks and aggregates the entire chat server's state. Highly recommended that you follow the sketch presented here.

- `server_main.cpp` - Contains the main function for the server. If you implement `server.cpp` correctly above, you should not need to make changes to this file.

Milestone 1: The clients

For the first part of this assignment, you will be responsible for implementing the *receiver* and the *sender* to communicate with a server binary included in the starter code. Note that the following messages are considered unused and do not need to be handled by any client:

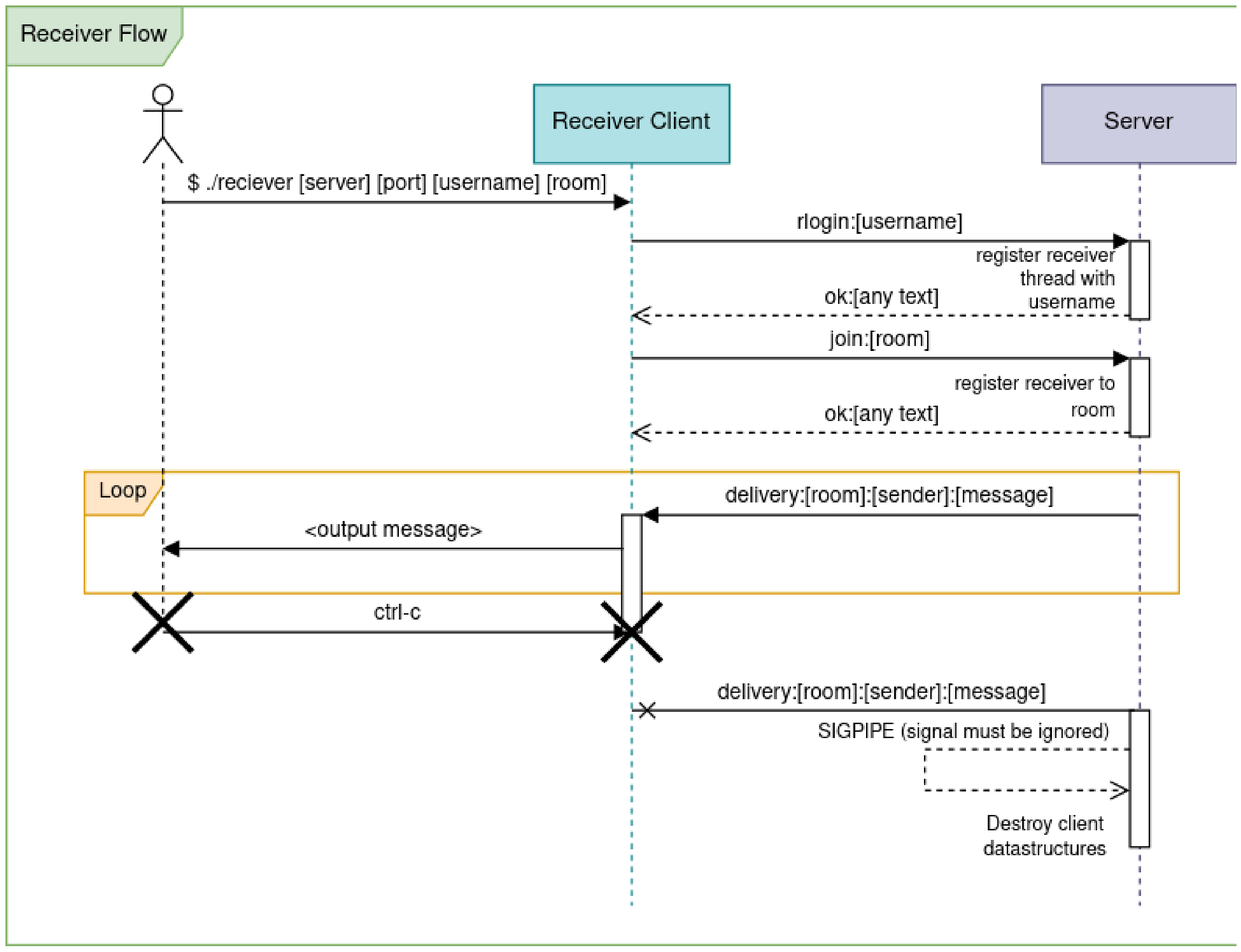
- `senduser`

Receiver

The receiver will be run in the following manner from the terminal:

```
./receiver [server_address] [port] [username] [room]
```

The receiver must send the `rlogin` message as its first message to the server. The following sequence diagram has been provided for your reference (note that this only covers the "happy case"):


Receiver Flow
Receiver Client
Server
$ ./reciever [server] [port] [username] [room]
rlogin:[username]
register receiver thread with username
ok:[any text]
join:[room]
register receiver to room
ok:[any text]
Loop
delivery:[room]:[sender]:[message]
<output message>
ctrl-c
delivery:[room]:[sender]:[message]
SIGPIPE (signal must be ignored)
Destroy client datastructures

The receiver should print received messages to `stdout` in the following format:

```
[username of sender]: [message text]
```

The following messages must be handled:

- `rlogin`
- `join`
- `delivery`
- `ok`
- `err`

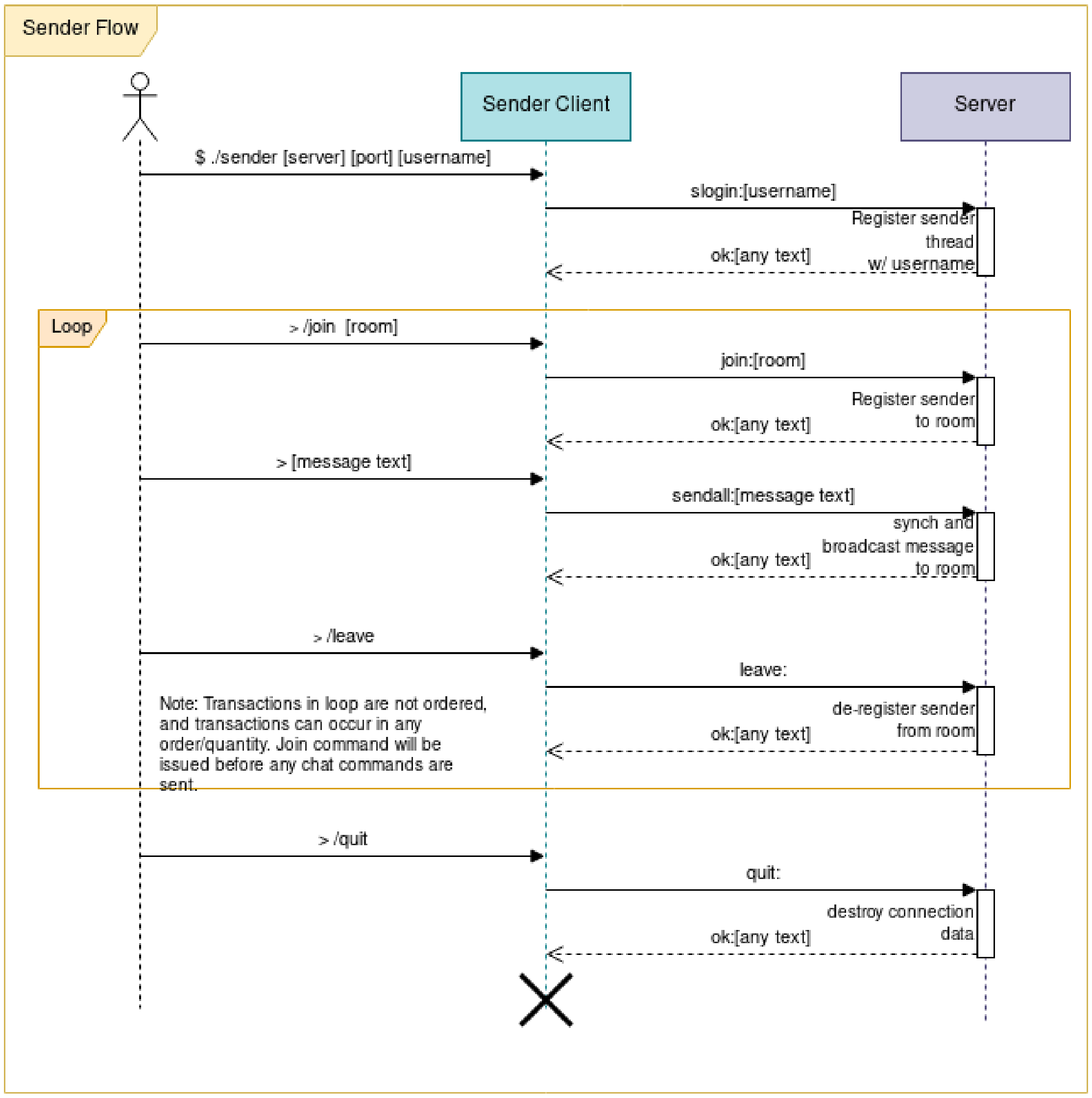
If the server returns `err` for either the `rlogin` or `join` message, the receiver must print the error payload to `stderr`/`cerr` and exit with a non-zero exit code. The receiver does not need to exit cleanly, we expect it to terminate it by sending it a `SIGINT` (a.k.a. `<ctrl>+c`).

Sender

Run the sender using the following command:

```
./sender [server_address] [port] [username]
```

The sender must send the `slogin` message as its first message to the server. The following communication flow has been provided for your reference (note that this only covers the "happy case"):


Sender Flow
Sender Client
Server
$ ./sender [server] [port] [username]
slogin:[username]
Register sender thread w/ username
ok:[any text]
Loop
> /join [room]
join:[room]
Register sender to room
ok:[any text]
> [message text]
sendall:[message text]
synch and broadcast message to room
ok:[any text]
> /leave
leave:
de-register sender from room
ok:[any text]
Note: Transactions in loop are not ordered, and transactions can occur in any order/quantity. Join command will be issued before any chat commands are sent.
> /quit
quit:
destroy connection data
ok:[any text]

The following messages must be handled:

- `slogin`
- `join`
- `sendall`
- `leave`
- `ok`
- `err`

After the sender logs into the server, it should read `stdin` for messages and commands. Commands start with the `/` character and may be one of the following:

- `/join [room name]` - joins the specified room on the server using a `join` message
- `/leave` - leaves the current room, stopping all message delivery using a `leave` message.
- `/quit` - Instructs the server to disconnect the current send client using a `quit` message.
- All other commands should be rejected with an error message printed to `stderr`/`cerr`

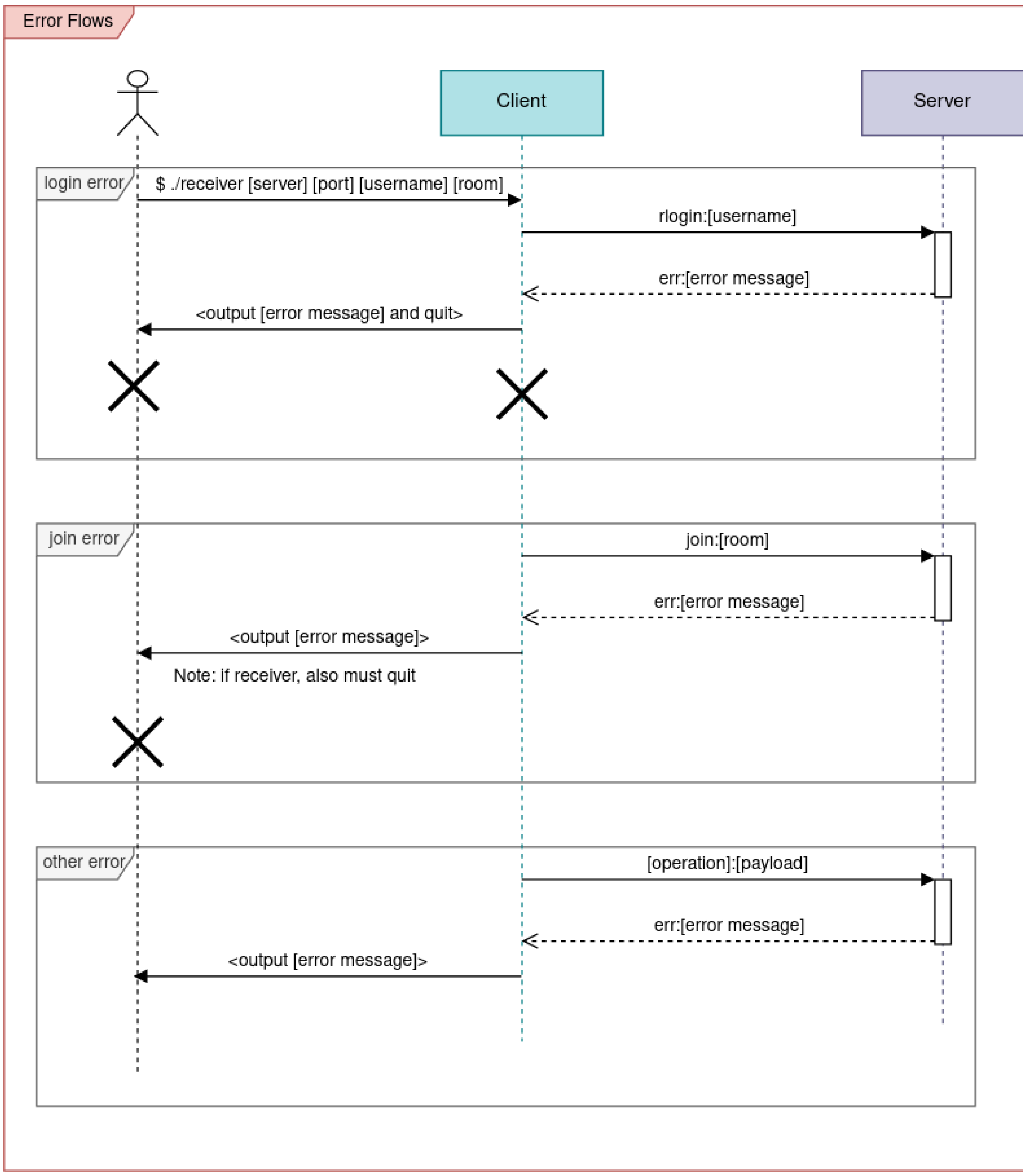
You may assume that all command arguments are valid if the command matches a recognized command.

The client must listen for a response from the server after sending each message (synchronous protocol). It is okay to stop reading user input during this time. If the server returns `err` in response to the `slogin` request, the sender should print the error payload to `stderr`/`cerr` and exit with a non-zero exit code. If the server returns `err` for any other request, the sender should print the error payload to `stderr`/`cerr` and continue processing user input.

If the `quit` commend is issued, the sender must wait for a reply from the server before exiting with exit code 0.

Error Handling

The following diagram summarizes how errors should be handled:



Each error message must be exactly one line of text printed to `stderr` / `cerr`. The error text printed must be exactly the payload returned from the server in the `err` message. You may assume that this payload will always be correctly formatted. For client-side errors, you may choose any error string.

You must handle failures to open the TCP communication socket by printing an informative error message and exiting with a non-zero exit code. You may assume that the server will stay online for the entire duration of the chat session.

If a client is run with an invalid number of arguments, a descriptive usage message should be printed to `stderr` indicating how the program should be invoked.

Implementation Tips

You are free to use any functions in the provided `csapp.h` header. In particular, we recommend that you use the `rio_*` family of functions for writing to the TCP socket file descriptors instead of using the raw syscalls. TCP connections have significant latency that requires reads and writes to be buffered correctly for expected behaviour. Remember that `rio_readlineb` does not strip the newline characters.

To open the client connection to the server, we recommend using the `int open_clientfd(char* hostname, char* port)` function. This function accepts a hostname (server address) as a string and the desired port as a string, and returns a file descriptor that is ready for use with the `rio_*` family of functions.

Testing

To aid your testing your program, we have provided a sample server implementation as a Linux binary in the starter code (see the Reference implementation section below.) We have intentionally compiled it without debugging information and stripped it of symbols. If your clients are implemented correctly, you should be able to type in a message and see the message appear on all read clients in the same chat-room. You may run our server binary using the following command:

```
./reference/ref-server [port number]
```

where `[port number]` is any integer greater than 1024. If the server fails to open on the given port, try another one. You must specify the same port between all clients and the server.

Note that you might need to set execute permission on the executable before running it:

```
chmod a+x reference/ref-server
```

We have only tested the binary on the Ugrad systems, and do not guarantee that it will work anywhere else. It definitely will not work on Mac computers, but may work on certain versions of WSL2.

You can also test one client at a time by using netcat as follows:

```
nc localhost [portnumber]
```

You can also spawn a netcat “server” using the following commands:

```
nc -l <port>
```

where port is a number greater or equal to 1024. You would then type in the server responses yourself in the netcat terminal window after you get a client connected to the “server” following the sequence diagrams above.

You can then pretend to be a receiver by sending a `rlogin` request:

```
rlogin:alice
join:cafe
sendall:Message for everyone!
```

Or you can pretend to be a sender by sending a `slogin` request:

```
slogin:bob
join:cafe
<messages will appear here as they are sent to the room "cafe">
```

Do not Valgrind `netcat` as that will not be testing your program, and may generate false positives. You should ensure that you run Valgrind directly on the client executables (e.g. `valgrind ./receiver ...`).

We have recorded a screencast which demonstrates several testing scenarios using combinations of the reference server, your clients, your server, and netcat:

> https://jh.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=c1bb9b7b-197d-490f-a811-b38e000c451b

We also have a recording of a terminal session where we demonstrate some of these manual testing workflows:

```
root@7f66594db398:/work#          We will now demo how to test your clients against a netcat server
                                  This will allow you to see exactly what your client
                                  ~
                                  -- INSERT --                                2,53          All
[0] 0:vim*                                          "7f66594db398" 23:42 21-Apr-22
```

Automated testing

You can obtain the automated test scripts here:

- test_receiver.sh
- test_sender.sh

Download them on in the terminal using `wget [link]` while you are in the same directory your project is in. Don't forget to make them executable after downloading them using `chmod u+x [file]`.

`test_receiver.sh` is invoked as follows:

```
./test_receiver.sh [port] [sender_client] [room] [server_in_file] [outpu
```

and `test_sender.sh` is invoked as follows:

```
./test_sender.sh [port] [sender_client] [client_in_file] [server_in_file
```

Note that `test_sender.sh` exits with the exit code the client exited with, so you can verify that your client exited with the correct exit code by running `echo $?` **immediately** after running the test script.

The arguments are:

- `port` - port to run server on. Pick anything above 1024.
- `*_client` - name of the client binary to run.
- `room` - room to connect the sender to.
- `server_in_file` - file containing list of messages server should send, one message per line
- `client_in_file` - file containing list of user inputs to the client, one per line.
- `output_stem` - base filename for the output, the files `[output_stem]-received.out`, `[output_stem]-client.out`, `[output_stem]-client.err` will be created which correspond to the messages sent by the client to the server, the output the client printed to `stdout`, and the output the client printed to `stderr` respectively.

While we highly encourage you to come up with your own test inputs, we have provided the following test inputs for reference:

- test_receiver_server.in
- test_sender_server.in
- test_sender_client.in

You can run the example receiver test using:

```
./test_receiver.sh 12345 receiver partytime test_receiver_server.in rece
```

and you should verify that `receiver_test-client.err` is empty, that `receiver_test-client.out` contains exactly:

```
bob: hi alice
robert_de_bobert: I have the cookies.
bob: cookies?
```

and that `receiver_test-received` contains exactly:

```
rlogin:alice
join:partytime
```

You can run the example sender test using:

```
./test_sender.sh 12346 sender test_sender_client.in test_sender_server.:
```

and you should verify that `sender_test-client.err` is empty, and that `sender_test-received` contains exactly:

```
slogin:alice
join:partytime
sendall:Hello World!
join:cafe
sendall:get me 1 coffee
quit:bye
```

With the exception of the payload to `quit` (it can be any text).

Milestone 2: The server

For this part of the assignment, you will be responsible for implementing the *server*. The server is responsible for accepting messages from *senders* and broadcasting them to all *receivers* in the same room.

The server can be run using the following command:

```
./server [port]
```

where `[port]` specifies the port that the server should listen on.

⚠ Important

Keep in mind that you may **not** use late hours for Milestone 2. Please plan accordingly.

Tasks

Here is a suggested order of implementation:

- Create a thread for each client connection. You will need a datastructure to represent the data associated with each client. We recommend that you use the same `Connection` class you used in part 1 to represent these connections. You may use the login message to determine what kind of client is trying to connect.
- Process *control messages* from clients.
- Broadcast messages to all receivers in a room when a sender sends a message. At this point your server should work with well-behaving clients that don't send messages at the

same time.
- Add synchronization for access to share data structures (`Room`s, `Server`) so no messages are lost, even if a receiver leaves or tries to join in the middle of a broadcast. You also must not lose messages that are sent at the same time from two different senders.

Using threads for client connections

Your server applications will to handle connections from multiple clients simultaneously in order to be useful (after all, it is quite sad to only be able to chat with oneself). Threads are a useful mechanism for handling multiple client connections because they allow the code that communicates with each client to run concurrently.

In your main server loop (`Server::handle_client_requests()` if you are following our scaffolding), you should create a thread for each accepted client connection using `pthread_create()`. A struct should be created to pass the `Connection` object and other required data to the client thread using the `aux` parameter of `pthread_create()`, and `worker()` should be used as the entrypoint for the thread. It may also be a good idea to create a `User` object in each client thread to track the pending messages, and register it to a `Room` when the client sends a join request.

You can test that your server handles more than one connection correctly by spawning multiple receivers and senders on the same server, and checking that the messages sent from all senders get correctly delivered to all receivers.

Receiver and sender loops

We recommend that you separate the communication loops for the senders and receivers into the `chat_with_sender()` and `chat_with_receiver()` functions respectively. Please refer to the sequence diagrams in Part 1 to determine how the loops should be implemented.

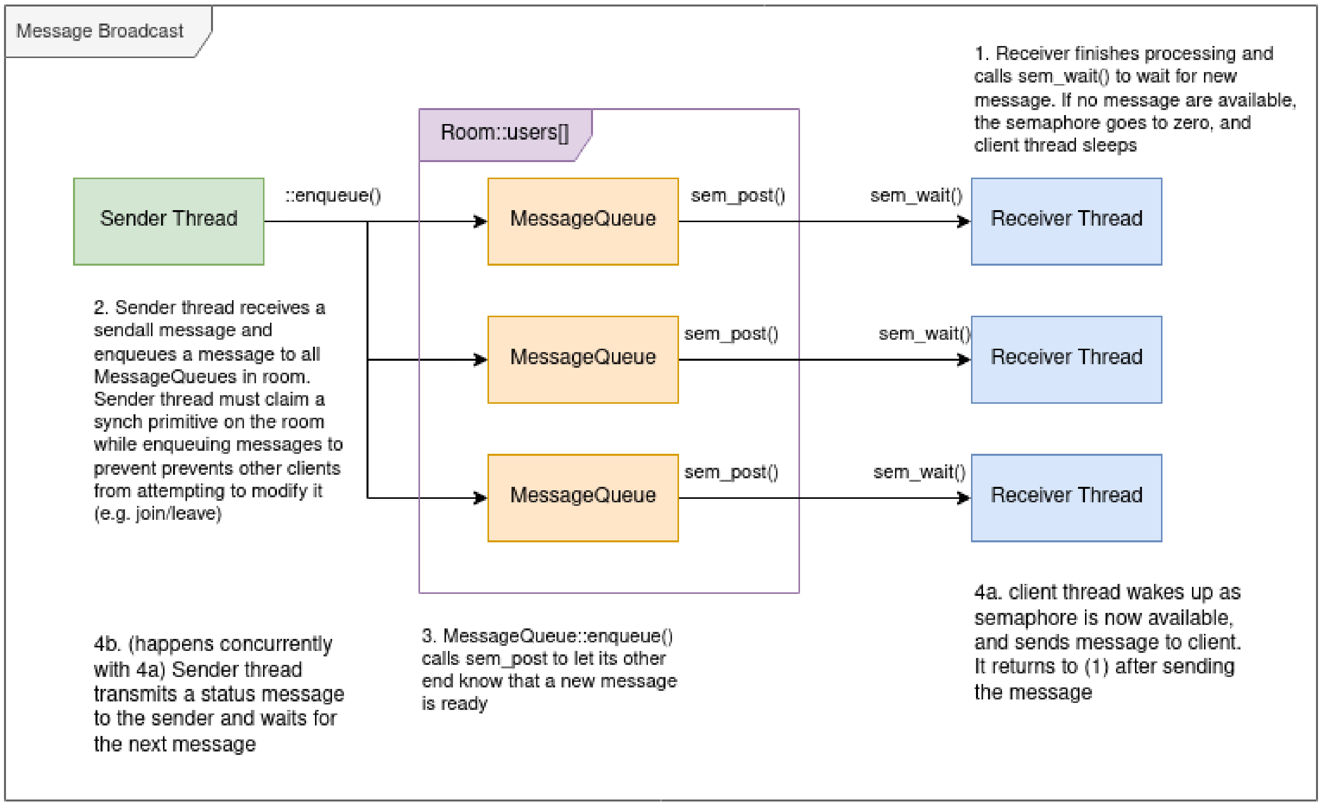
We have already handled the `SIGPIPE` signal for you in our provided server main function, so you should be able to detect partial reads by matching the return value of `rio_*` against the size of the message transmitted. If they do not match, you may assume that a transmission error has occurred and should handle it as an error.

For all synchronous messages, you must ensure that the server always transmits some kind of response (`err` for error, `ok` for success) to receive full credit. Failure to transmit a response to a synchronous message will cause the client to hang.

In the receiver loop, you must terminate the loop and tear down the client thread if any message transmission fails. For the sender loop, you must terminate the loop and tear down the client thread if any message fails to send, as there is no other way to detect a client disconnect. Be sure that you clean up any datastructures and entries specific to the client before terminating the thread to prevent resource leaks.

Broadcasting messages to receivers

We recommend that you implement the pub/sub pattern for synchronization, using the `MessageQueue` class we outlined for you. In this pattern, a sending thread iterates through all the `User`s in a room and pushes a message into each `MessageQueue`. This event wakes up the receiver thread, allowing it to dequeue the messages at its leisure. Here is a diagram of how this could work:


Message Broadcast
1. Receiver finishes processing and calls sem_wait() to wait for new message. If no message are available, the semaphore goes to zero, and client thread sleeps
Room::users[]
Sender Thread
::enqueue()
MessageQueue
sem_post()
sem_wait()
Receiver Thread
MessageQueue
sem_post()
sem_wait()
Receiver Thread
MessageQueue
sem_post()
sem_wait()
Receiver Thread
2. Sender thread receives a sendall message and enqueues a message to all MessageQueues in room. Sender thread must claim a synch primitive on the room while enqueuing messages to prevent prevents other clients from attempting to modify it (e.g. join/leave)
4a. client thread wakes up as semaphore is now available, and sends message to client. It returns to (1) after sending the message
4b. (happens concurrently with 4a) Sender thread transmits a status message to the sender and waits for the next message
3. MessageQueue::enqueue() calls sem_post to let its other end know that a new message is ready

Queues are a useful paradigm because they allow messages to be processed at a different rate than we they are transmitted . If we had to wait for all messages to finish sending before releasing the lock on the room, we could end up spending all of our time servicing send requests, which would deadlock the server.

To implement this “notification” behaviour, we recommend that you use a combination of a semaphore and a lock. The lock ensures that the message queue can only be modified by one thread at a time, and the semaphore is used to “notify” the other end that a new message is available. Recall that a semaphore blocks a thread when it goes below zero, and can be incremented (`sem_post`) and decremented (`sem_wait`, `sem_timedwait`) from different threads. Thus, we can essentially use the semaphore in each `MessageQueue` as a sort of “smart” counter of the available messages in the queue. This implements the correct behaviour: if there are no messages available, we want the receiver to sleep until there are available messages, and each time a message is sent, it reduces the available messages by one.

The sender client thread may return a response to the sender as soon as the message is done being added to all MessageQueues. It does not need to wait until the message has actually been delivered to all of the receivers in the room, but it should not return a status until the message is done being enqueued.

Important: Remember that stack allocated data is *thread local* and must not be shared between threads. Since the `MessageQueue` is being used as an inter-thread communication primitive, you must ensure that messages pushed to the queue are always heap-allocated (e.g. allocated using `new`). Likewise, ensure that the dequeuing thread takes responsibility for freeing the memory. Finally, don’t forget to empty the queue (i.e. free all queued messages) when the queue is destroyed to prevent memory leaks when a *receiver* disconnects before all messages could be delivered to it (consider where this happens carefully to implement the correct synchronization).

Note: While we recommend `sem_timedwait()` in the starter code, `sem_wait()` is also acceptable for simplicity (i.e. you may safely ignore `ts`). (Using `sem_timedwait()` has the advantage that the thread handling a connection with a receiver will not be blocked indefinitely if there are no messages waiting to be delivered to that receiver.)

Synchronizing shared data

Synchronization is typically necessary when multiple threads can attempt to access the same data in a hazardous manner at the same time. Synchronization may also be necessary if certain semantics are desired of accessed to shared data (e.g. guaranteed ordering).

Strictly speaking, if the data type is *atomic*, read accesses need not be synchronized so long as they can never occur at the same time as a write. However, for this assignment, we are not using *atomic types*, so you will need to synchronize *all concurrent access*.

The section of code where synchronized access to data is imposed is called a "critical section", and should be limited in length as concurrency is greatly restricted in these sections. Making critical sections too long can potentially cripple performance in real-world applications.

Add synchronization to the `Server`, `MessageQueue`, and `Room` objects to ensure that updates to these objects will never be lost, not matter how the objects are accessed. For example, if multiple clients try connect to the server at the same time, both clients must be registered correctly, without losing either one. Likewise, if two clients try join a room or send a chat at the same time, both requests must be successfully carried out, with neither operation "lost" or partially completed.

In a more practical sense, you may want to introduce a mutex to the `Server`, `Room` and `MessageQueue` objects, and then add critical section(s) where needed to ensure that the synchronization requirements are met. **Very important**: You should not allow critical sections to be accessed across object boundaries to prevent synchronization bugs. For example, if you implement a mutex in the `Room` class, you should make it private and only synchronize to it from `Room` methods.

Consider your synchronization hazards carefully! There are a few cases that may cause data races that are not immediately obvious (e.g. you must ensure that clients never broadcast and join/leave the room at the same time to prevent races).

Guard locks

To help ensure that locks are always released, we have provided a "block scoped lock" implementing the "Resource Acquisition Is Initialization" (RAII) pattern in `guard.h`. This means that constructing the `Guard` object blocks until the lock is acquired, and allowing it to go out of scope releases the lock. If you need the lock to be held for a shorter scope than the entire enclosing block, you can introduce additional scoped blocks:

```
void foo(pthread_mutex_t *lock) {
    ...
    // introduce new block scope
    {
        // Aquire lock, blocks thread until lock becomes available
        Guard(*lock);
        // do something with the lock held
        // invariant upheld: only one thread may enter this section at a
        ...
    }
    // lock is RELEASED here, and threads will be concurrent
```

```
        ...
    }
```

We **highly recommend** that you use `Guard` objects instead of raw calls to `pthread_mutex_lock()` and `pthread_mutex_unlock()`, as the block scoping ensures that you will never forget to release the lock. This prevents a vast class of possible deadlocks. Remember that `pthread_mutex_init` must be called exactly once on each mutex before it can be used.

Synchronization report

Since synchronization is an important part of this assignment, we'd like you to support a report on your synchronization in your README.txt. Please include where your critical sections are, how you determined them, and why you chose the synchronization primitives for each section. You should also explain how your critical sections ensure that the synchronization requirements are met without introducing synchronization hazards (e.g. race conditions and deadlocks).

Error Handling

If the server fails to bind the listen TCP socket for any reason on the host, you must print an error message to `stderr` and return a non-zero return code. Once the server binds the port and starts listening for clients, it does not need to handle shutting itself down.

We expect your server to be *robust*. This means no matter what any client sends, in any order, your server should not crash. To ensure that this is the case, you probably will want to use the `rio_*` functions, and the `Connection` class you implemented for the clients. Some (non-exhaustive) examples of bad things the clients may do that *should not* crash your server include:

- Sending messages longer than `Message::MAX_LEN`
- Sending invalid messages that cannot be parsed, including empty messages.
- Sending messages with invalid tags.
- Leaving and joining at any point in the communication sequence.
- Attempting to send a response to a client that has dropped off between the time a message was sent and before it could accept its response.

If a message cannot be parsed, could not be carried out, or is not a valid message, you must send an `err` message to the client with a descriptive payload. If a sender tries to send a message or leave a room while it is not in a room, you must also return `err` with a suitable payload. This *should not* stop the server, nor disconnect the client. Otherwise, you *must* send an `ok` message with suitable payload. Failure to send a response to a client operating in synchronous mode at any time is a *severe* bug that will cause most tests to fail.

All client data should be cleaned up as soon as the server detects that the client connection has died. You may assume that any transmission error indicates that a client has died. It is okay if receivers are not cleaned up until the next broadcast is sent to a room for ease of implementation.

Since your server has no way of shutting down, you may ignore the "in-use at exit" portion of valgrind. You should still fix any leaks (sections marked "definitely lost"), invalid reads, invalid writes, and invalid conditional jumps

Implementation tips

Start early! There are quite a few things you will need to consider in order to receive full credit. We recommend that you start by implementing the logic that waits for new client connections and spawns client threads to handle them. If you are struggling with synchronization, we recommend that you start with a basic implementation without any synchronization, which might help you identify critical sections.

We recommend that you use *detached threads*. This means that you will not have to join them back to the primary thread, and that you do not have to save the `pthread_t` returned from `pthread_create()`. There are two safe ways to do this. This first is to initialize a `pthread_attr_t` struct with the correct flags using `pthread_attr_setdetachedstate()`, and pass this into the relevant argument for `pthread_create()`. The second is to call `pthread_detach(pthread_self())` from the child thread. Under no circumstances should you attempt to detach the thread using a call in the creating thread, after the child is created, as that will cause a data race.

Don't forget to initialize your synchronization primitives before use. For `pthread_mutex_t`s this is `pthread_mutex_init()`. For semaphores, this is `sem_init()`. Use of synchronization primitives before they are initialized is undefined behaviour and *will break your code*. You should also destroy your synchronization primitives when you release their associated resources. Failure to call the destruction functions may result in leaked memory.

Do not attempt to share stack-allocated data between threads. This is undefined behaviour and generally causes *severe* bugs. Instead, ensure that any data that must be accessed between threads is part of a heap-allocation.

We also recommend using scope to your advantage. RAII resources such as the provided `Guard` type prevent mistakes like forgetting to release resources, and defining variables to the narrowest scopes they require dramatically reduces the clast radius of any bugs that do arise. Aim to fail fast and early if invalid states are encountered.

If the server appears to become unresponsive for a long period of time it has probably deadlocked, and you will need to examine your synchronization. If a server doesn't bind a socket on a given port, try another one, as the port you are trying to bind may be already taken (e.g. by another student on your ugrad machine).

Testing

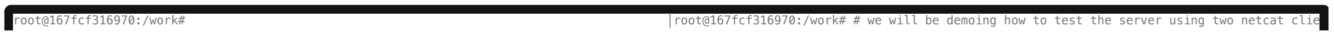
We have provided the testing methods below to help you ensure that your program is working correctly. *We highly recommend against using the autograder as your primary testing solution*. The autograder is designed to be robust and thorough, and intentionally does not provide test feedback to you. Instead, we recommend that you use local testing techniques so you can use tools like debuggers and print statements to help debug.

Manual Testing

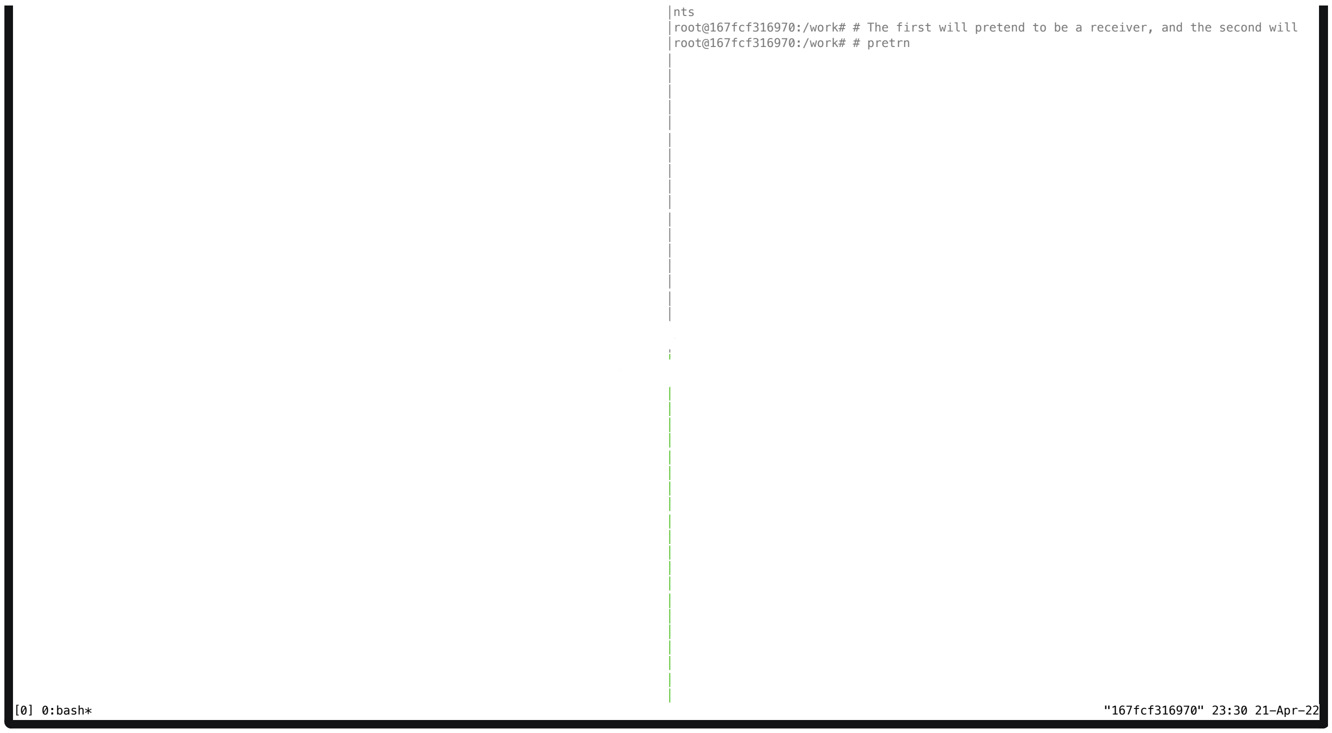
To test this program, you may follow the instructions in the MS1 Testing section, replacing the invocations of your client with `reference/ref-[client]`. Netcat testing will probably also be a good idea so you can figure out exactly what your server is sending.

For example, to test a server implementation against netcat, you could open three terminal windows. In the first, you would run `./server [port]`, in the second you would run `nc localhost [port]` and send a `rlogin` message, in the third you would run `nc localhost [port]` and send a `slogin` message. Then you would follow the flow diagrams to send messages from your netcat "clients", verifying the server responses that appear. If everything works in manual netcat testing, you would move onto testing with our reference binaries using a similar approach, before trying the automated test scripts posted below.

Here is a capture of an example testing session:



```
root@167fcf316970:/work#                                      |root@167fcf316970:/work# # we will be demoing how to test the server using two netcat clie
```



```
nts
root@167fcf316970:/work# # The first will pretend to be a receiver, and the second will
root@167fcf316970:/work# # pretrn
```

```
[0] 0:bash*                                                   "167fcf316970" 23:30 21-Apr-22
```

Automated Testing

Here are some automated tests you can try:

- test_sequential.sh
- test_interleaved.sh
- test_concurrent.sh

Don't forget to make these scripts executable using `chmod u+x [script name]`!

`test_sequential.sh` runs two senders, one after the other, and is invoked using:

```
./test_sequential.sh [port] [first_sender_input_file] [second_sender_inp
[output_stem]
```

`test_interleaved.sh` runs two senders, alternating between them for each line of the input file. I can be invoked as follows:

```
./test_interleaved.sh [port] [unified input file] [output_stem]
```

`test_concurrent.sh` does its best to break your server's synchronization by spawning all sorts of clients that try send data as fast as possible while simultaneously switching rooms and disconnecting. It can be invoked as follows:

```
./test_concurrent.sh [port] [iterations] [settling time]
```

`[output_stem]` will set the file that contains the receiver output after each run. `[output_stem].out` contains the output of the receiver, and `[output_stem].err` contains the receiver errors. For the sequential and concurrent tests, we always expect the first user to be `bob` and the second user to be `alice`, and errors for their respective senders will be found in `[user].err`. Please keep this in mind as you write additional tests.

While we highly recommend you write your own test cases, we have provided the following tests inputs as examples:

- test_inter.in
- seq_send_1.in
- seq_send_2.in

You can run the reference sequential test using the following command:

```
./test_sequential.sh [port] seq_send_1.in seq_send_2.in seq_recv
```

and you should get nothing in `seq_recv.err`, `bob.err`, `alice.err`, and the following output in `seq_recv.out`:

```
alice: Hello everyone
alice: I am trying to purchase the cookies
alice: Please give me your headcount and the number of cookies you want
bob: Hi Alice
bob: This is Bob.
bob: I want a chocolate peanut cookie with walnuts.
bob: Thanks!
```

and you should ensure that your server does not print anything to `stdout`.

You can run the reference interleaved test using the following command:

```
./test_interleaved.sh [port] test_inter.in inter_recv
```

and you should get nothing in `inter_recv.err`, `bob.err`, `alice.err`, and the following output in `inter_recv.out`:

```
alice: This is a message from alice
bob: And this is a message from bob
alice: each alternating line of messages...
bob: will be send by a different client...
alice: this test ensures that clients are served in order
bob: and that one client may not monopolize the entire transmission
```

and you should ensure that the server does not print anything to stdout.

Some good parameters to start the concurrency test with are 10000 iterations and 30 seconds of settling time. If the test succeeds, you should see `Tests passed successfully!`. Increasing the number of iterations will increase the likelihood of detecting a race condition. If the test ends with `Failed to verify*` try increasing the settling time. If increasing the settling time to over one minute does not allow the test to pass, you probably have a race. However, do note that having this test pass does not guarantee that your code is sync-safe.

You should run the concurrency test *last*, after you get all other functionality working. The concurrency test will exercise all parts of your server while it tries to cause race conditions.

Note that the server can be run under valgrind by setting the `VALGRIND_ENABLE` environment variable to `1`. For example, if you want to run the sequential test with valgrind, the command would be run using `VALGRIND_ENABLE=1 ./test_sequential.sh ...`. Remember that you may ignore the reports for “indirectly lost”, “possibly lost”, and “in use at exit”, and any leaks caused by `pthread_*` functions, but must fix everything else.

Reference

In the `reference` directory of the project skeleton, you will find executables called `ref-server`, `ref-sender`, and `ref-receiver`. As the names suggest, these are the reference implementations of the server, sender, and receiver. Your `server`, `sender`, and `receiver` executables should be functionally equivalent.

Here is a suggested test scenario. You will need three terminal sessions.

In terminal number 1, run the server (user input in **bold**):

```
$ ./ref-server 47374
```

You can use any port number 1024 or above instead of 47374.

In terminal number 2, run the receiver (user input in **bold**):

```
$ ./ref-receiver localhost 47374 alice cafe
```

Make sure you use the same port that you used in the `server` command.

In terminal number 3, run the sender (user input in **bold**):

```
$ ./ref-sender localhost 47374 bob
/join cafe
hey everybody!
/quit
```

In terminal number 2 (where the receiver is running, you should see the following output):

```
bob: hey everybody!
```

Note that while the `ref-sender` program will terminate when the `/quit` command is executed, the `ref-server` and `ref-receiver` programs will need to be terminated using Control-C.

Submitting

You can use the `solution.zip` target in the provided `Makefile` to create a zipfile you can submit to Gradescope:

```
make solution.zip
```

Upload your `solution.zip` as **Assignment 5 MS1** or **Assignment 5 MS2**, depending on which milestone you are submitting.

Make sure your Milestone 2 submission includes your `README.txt` describing your approach to thread synchronization in the server.