Homework 5

Version 1

This assignment aims to give you practice writing a (concurrent) chat server. The chat server maintains channels, each containing a list of messages. A client can post or retrieve messages from a given channel.

I strongly recommend that you complete your assignment following the sequence of steps below.

Step 1:
We will start by focusing on creating the data structures used to manage the channels and their associated messages. The type definitions are included in storage.h, and you'll implement several methods in storage.c. The channels are stored as a singly-linked list. Each element of the list conforms to the *channel_t* type, which includes:

- name – A unique string identifying the channel. You must ensure that creating multiple channels with the same name is impossible.
- next – A pointer to the next channel.
- head/tail – The head and tail of a linked list that includes the messages associated with this channel
- last_msg – The id of the last message plus 1. Initially, last_msg is 0.

As indicated above, for each channel, there is a list of messages that are stored. Each message is defined in the *message_t* type and includes the following members:

- id – An unique identifier for the message. Note that for a given channel, each message has a unique identifier. Furthermore, your implementation must ensure that the message's id is created incrementally (i.e., 0, 1, 2, and so on) and the biggest message id in the channel is one less than the value of last_msg of the channel.
- text – A string including the text of the message.
- next – A pointer to the next message of the channel.

To manipulate these two data structures, you will implement the following API:

- *channel_list_t *get_channels()* – returns a pointer of type *channel_list_t* that specifies the head and tail of the channels list. I have already implemented this method for you.
- *channel_t *create_channel(channel_list_t *channels, const char *name)* – this method creates a new channel with the supplied name and adds it to the list. If a channel with channel_name already exists, a pointer to that channel is returned.
- channel_t *get_channel(channel_list_t *channels, const char *name) – the method returns a channel based on its name. If the channel does not exist, NULL is returned.
- *void add_message(channel_t *channel, const char *text)* – the method will create a message_t that contains the provided text as the text field. Then, the created message is added to the list of

messages for the specified channel. Note that you must ensure that the message's id is created according to the constraints specified above.

- *message_t *get_message(channel_t *channel, message_id_t id)* – the method searches the channel's messages, and if it finds one with the supplied id, that message is returned. If no matching message is found, then NULL is returned.
- *void dump(channel_list_t *channels)* – Prints the channels and the associated messages. This method is provided for you.
- *void free_channels(channel_list_t *channels)* – frees all the channels and their associated messages.

To get you started with testing your code, I'm providing test_step1.c. The function includes an illustrative usage of the above API.

Step 2:
Next, we implement the client/server for the case when at most one client is connected to the server at a time. The basic functionality for the client and the server is provided in client.c and server.c, respectively. We have walked to similar examples in the class, so it should be familiar to you. The code is set up to connect to the localhost (which has an IP address of 127.0.0.1) and on port 8080.

In the following description, I assume that the server already contains a channel called "weather." The channel has the following messages with message ids specified in the square brackets:

- [0]: the weather is going to be great!
- [1]: sunny with a high of 70F

The client can perform three operations --- retrieve one message from a channel, retrieve all messages from a channel, send a message to a channel – depending on the arguments supplied from the command line.

Retrieve a message: When the client is invoked using "*client -channel weather -msg 1*", the client is instructed to retrieve the message with id of 1 from the weather channel. As expected, the output of the client should be:
"!! weather>> sunny with a high of 70F"

I have already parsed the arguments for you (in the *parse_args* method) and calls *retrieve_message(int socket, const char *channel_name, message_id_t msg_id)* with the parameters extracted from the command line. The method should behave as follows:

- If the server has a channel with a name equal to channel_name and a message with id msg_id, the client should retrieve the message text and display its contents. Please use the following printf statement:
- printf("!! %s>> %s\n", <<channel name>>,<<text of message>>);

- if the server has the channel with a name equal to channel_name but no message with id equal to message id, then it should print:
    - printf("!! Message not found\n");
- if the server does not have the specified channel, then the client should print
    - printf("!! Channel not found\n");

Behind the scenes, the client and the server must exchange messages. I leave it up to you to decide what the server and client must exchange information.

In my implementation, the client sends a request, and the server responds with a reply. The request includes an identifier to discriminate between the request and reply, the message id, and the channel name. Note that since the channel is a string, I'm using the same trick I showed you in class to avoid the complexity of handling null-terminated strings. Accordingly, I first send the length of the string and then the content of the string. The reply includes a status indicating whether:

1. The specified channel does not exist.
2. The specified channel exists, but no message with the specified id exists.
3. Both the channel and the message exist. In this case, I also send the size and contents of the message.

Retrieve all messages: Next, we will implement the retrieval of all the messages of a channel. You obtain this behavior by invoking the client with only the channel as an argument: "./client -channel weather." Executing this command should return all the messages in the weather channel. I parse the arguments and call *retrieve_messages(int socket, const char *channel_name)* with the appropriate arguments. You'll have to implement this method.

If you have correctly implemented the previous step, printing all the messages is quite simple. Given a channel, you iteratively request messages with ids 0, 1, 2, and so on until the server responds that the request message (or channel) does not exist. In this manner, you can retrieve all the messages of a channel.

Hint: You can write the retrieve_messages using the retrieve_message you previously implemented. In this way, there is no need to deal with sockets again.

Send message: Finally, you want to implement the sending a message to the server. You need to pass the channel and the message's text to be posted to the channel from the command line. For example, ./client -channel weather -text "sunny today" should create a message with the text "sunny today" and post it to the weather channel. If the specified channel does not exist, a new channel with the specified name is created before adding the message.

You'll put your implementation in the send_message function. The send_message takes the name of the channel and the text of the message to be posted. I suggest that you structure the exchange between

the client and servers as a request and a reply (i.e., do not omit the reply). The request can include information about the channel and text you want to add. The reply is a simple confirmation that the operation has been completed on the server.

Step 3:
The final step of the assignment is to support multiple concurrent clients. Your implementation must meet the following requirements:

- A separate thread must handle each connection between the client and server. It is safe to assume that a maximum of 100 clients will be connecting to your server. Note that only the server needs to use threads, it is okay for the client to use a single thread.
- You must appropriately protect access to the shared channel and message lists. Your implementation must ensure that:
    - You must lock access to a channel's messages to ensure that clients can safely add messages concurrently. Messages cannot be lost.
    - You must lock access to the channels list to ensure that clients can safely access and create channels.
    - You must ensure the previously stated invariants about channel names and ids hold.

To turn in:
Create a zip file including all the modified files. I expect that running make will create the client and server executables used for testing. You cannot use other libraries than pthreads (and the standard c library). Include a readme.md file specifying how much time you spent on the assignment and what you learned.

Extra Credit:
I am providing the following options for extra credit:

- Modify the server to persist the received messages to disk. When the server is stopped, dump all the data to disk. When the server is restarted, load the saved messages from the disk.
- Write a (minimal) Java Script interface to connect to the server (written in C).