Project 2 Description

CS 2510, Spring 2023: Project 1

Project Overview

In this project, you will implement a replicated version of the group chat system you developed in Project 1.

For this project, you should assume:

- Servers may crash and subsequently recover
- Clients may crash
- The network may arbitrarily drop, delay, or re-order messages, and may partition (and subsequently merge)
- You may have many clients; the number of clients is not known in advance and you should not assume any specific bound on the number of clients
- You have exactly 5 replicas of your chat server with IDs from 1-5. The chat server program should take its server ID as a commandline argument

You should construct a reliable and efficient chat service that is *available* as long as at least one server is up and reachable, and is *as consistent as possible* given network conditions. If a client can communicate with a server, it should never be blocked from interacting with the chat service and performing operations. The client's view of the chat group they have joined must always be as consistent as possible, given network connectivity, regardless of which chat server is consulted. The views should be consistent in the lines that are displayed, as well as likes and group membership. If lines are appended by different clients, possibly in different network partitions, they should all appear once the partition is healed. Causality should always be preserved (e.g. questions should appear before their answers).

Each chat server should store information on disk to allow it to recover after a crash. Note that servers can be eventually consistent with one another even if they are never directly connected (eventual path propagation).

General Requirements

- Your submission must include a design document with a detailed description of your algorithm and system design. You will submit an initial design document, but the final report must include an updated design document that matches your final implementation.
- Your submission must include a Dockerfile that creates an image we can use to run your code (installs dependencies and compiles code if needed) and a README with the exact commands

to run your code.

- You must use a Linux-based Docker image (we will be providing infrastructure to emulate message loss and network partitions in Linux)
- In designing and implementing your project, you should focus on **correctness** (the system produces the correct results, with as consistent a view as possible for all inputs and failure conditions), **efficiency** (does not send messages or do processing unnecessarily), and **simplicity** (only as complex as necessary)

Chat Service Specification

You should implement a chat service that allows clients to perform the actions below, along with a simple text-based client program that allows users to access those actions interactively as follows. New features compared to Project 1 are shown in blue:

1. Connect to a chat server. `c localhost:12000` will connect to a server running locally on port `12000`. Note that either a hostname or an IP address may be provided. A client must be connected to a server to perform any of the operations below. At any time, the client may only be connected to one chat server. The client may change its connected server at any time.

2. Login with a user name. `u Amy`. The user can change their identity at any time using the same option (the client will be removed from the chat group the old user was in, if any). Note that a **user** may be simultaneously logged in through multiple **clients** and may be connected to multiple different chat servers or to the same chat server (and should see a consistent view as much as possible). Different instances of the same user (i.e. different client programs) may join the same group or different groups. If one user instance (client program) logs out or changes groups, that should NOT cause other instances of the same user to change groups or be removed from their current group.

3. Join a chat. `j group1`

   If the specified group does not exist, it is created.

   Upon joining a chat, the current state of the group should be presented, such that all the current (logged in) group participants are listed, and the most recent 10 messages in this group (lines) are presented, oldest to newest. The number of unique ‘likes’ for each line (see below) should be presented to the right of the line.

   Example:

```
Group: CS2510 Staff
Participants: Amy, Maddie, Shriharsha
1. Amy: Hello!
2. Amy: Has anyone started the project?                    Likes: 2
3. Maddie: Not sure yet...
```

After joining a chat, upon an update to the chat, the screen should automatically update to include the new message, or to include or remove attendees (the screen should only refresh upon a relevant update).

A connected client can switch to a different chat room at any time by joining the new chat room. Joining a new chat removes the client from its previous chat room. The client needs to be in a chat to perform any of the operations below.

4. Append to a chat. `a` and then type a line.

   The relevant chat should be updated so that the new line is added at the end of the chat. You can assume that the maximum number of characters in a line is 80 and that each line is, in fact, a single line (i.e. no newline characters other than the final one at the end).

5. Mark line as 'liked'. `l 5`. (Note, this is a lowercase L, not the number one)

   The relevant line on the screen will be marked as liked by this user, but only if the user is not the creator of the line. Each user can only be counted once for each line. Note that as a consequence of this update, the screen may need to be refreshed. Line numbers are only relevant to the current display.

6. Remove your 'like' from a line (if you 'liked' it). `r 5`.

   The user's 'like' will be removed from that line if it is currently liked by that user. Note that any inconsistencies due to partitions must be resolved consistently and logically across all servers.

7. Print the full message history of the chat (not only the latest 10 lines). `p`. Note that it is ok to go back to showing only the latest 10 lines the next time the screen refreshes.

8. Print the server's current view of which servers it can currently communicate with (i.e. are in the same network component/partition. `v`.

9. Quit the program. `q`.

Testing

We have provided a program to help test your project and emulate server and network failures, `test_p2.py`.

You can use this program to initialize and launch your five server instances with command:

```
python test_p2.py init
```

This assumes you have already built your Docker image and that its name is `cs2510_p2`. It also assumes each server can be started from that image with the command `./chat_server -id [id]`, where `id` is an integer from 1 to 5.

Note 1: this program starts up your servers as background processes which **WILL NOT WORK** if your Dockerfile includes the line "**ENTRYPOINT /bin/bash**" (bash requires a terminal to be allocated for the process). If you have this line in your Dockerfile, you should simply **remove it** (we will give the command to run in the container explicitly, so it is not needed).

Note 2: this program uses tc netem to manage emulated loss and partitions. This **WILL NOT WORK** if you are using the **WSL backend for Docker on Windows**. If you use Windows Pro or Education editions, you can change the backend in Docker Desktop General Settings (uncheck the "Use the WSL 2 based engine" option). If you use Windows Home, this is not an option, so you will need to do your testing in a Linux environment. If you do not have one available, we can set you up on GENI (please ask Maddie if you need this).

It will connect all servers to a bridge network called `cs2510`. Each server will be assigned an IP address of the form `172.30.100.10[id]`, where `id` is the server ID from 1-5 (e.g. server 1 has IP address 172.30.100.101, server 2 has IP address 172.30.100.102, …).

You can confirm that the command worked by running `docker ps`. You should see output similar to the following:

```
CONTAINER ID   IMAGE       COMMAND                  CREATED         STATUS         PORTS     NAMES
85362037e8d4   cs2510_p2   "./chat_server -id 5"    4 minutes ago   Up 4 minutes             cs2510_server
5
2f64102caf7f   cs2510_p2   "./chat_server -id 4"    4 minutes ago   Up 4 minutes             cs2510_server
4
3c51e009944d   cs2510_p2   "./chat_server -id 2"    4 minutes ago   Up 4 minutes             cs2510_server
2
d78002ba9917   cs2510_p2   "./chat_server -id 3"    4 minutes ago   Up 4 minutes             cs2510_server
3
3b33e302c8a9   cs2510_p2   "./chat_server -id 1"    4 minutes ago   Up 4 minutes             cs2510_server
1
```

Note that your servers will be running in the background - if you want to see the output from the chat_server process on server 1, you can check the logs with the command `docker logs cs2510_server1`.

You may want to test that your servers can communicate with one another by running a ping command. You can do this by opening a new terminal on one or more servers (e.g. `docker exec -it cs2510_server1 bash` to open a terminal on server 1), and running a command like `ping cs2510_server2` to check that servers 1 and 2 can communicate.

You can then add loss, create partitions, and kill and relaunch servers. See the program help for more details (e.g. `python test_p2.py --help`, or specific help for each command, like `python test_p2.py loss --help`).

Setting loss rates:

```
python test_p2.py loss container1 container2 loss-rate
```

This will add loss-rate percent loss between servers with IDs container1 and container2. For example:

```
python test_p2.py loss 1 2 20
```

adds 20% loss between servers 1 and 2.

Creating partitions:

Example:

```
python test_p2.py partition 1,2 3,4,5
```

Creates 2 partitions, one containing servers 1 and 2, and another containing servers 3, 4, and 5. NOTE: Creating/modifying partitions removes any loss that was previously added between containers.

Crashing and restarting servers:

Example:

```
python test_p2.py kill 1
python test_p2.py relaunch 1
```

The first command will crash server 1, and the second will restart it.

Cleaning up:

To remove all server containers (e.g. so that you can make changes to your image and re-create them):

```
python test_p2.py rm
```

Running Client Programs:

The test script only manages your servers, not clients. To run a client that connects to the bridge network, you can use:

```
docker run -it --cap-add=NET_ADMIN --network cs2510 --rm --name cs2510_client1 cs2510_p2 /bin/bash
```

To run multiple clients, you just need to give each one a different name using the --name parameter.