MULTI-THREADED CHAT SERVER

Tiffany Christensen and Matthew Niemiec

Introduction

I. Why is the problem interesting?

One of the most valuable functionalities of the Internet is its ability to offer real-time transmission of messages between two end-points. The capability of being able to communicate with anybody in the world with just a server, two computers with Internet access, and a user-friendly chat application helps accomplish this. As such, building a simple chat server provides a window into these technologies, which include threading and client-server socket protocols.

II. Why is it Hard?

In creating a chat server, there are many difficulties that can arise. This includes the unpredictability that characterizes users and networking. To elaborate, the chat server needs to be able to handle a variety of user inputs and behaviors. Ensuring that the chat server implementation handles all potential cases is challenging. In order to ensure consistent and reliable handling of a multitude of situations, it requires a lot of user testing. Additionally, there were many failures that can happen on the client side as well as the network side, and each of these failures needs to be caught and properly handled. As such, it requires a lot of forward thinking and planning.

III. Example

The application of a chat server can be modeled by the following scenario. There are three people—Alice, Bob, and Sam– who are close friends, but recently

have been geographically split up. Although each friend lives in a different part of the country, they would like to still have conversations with each other. Fortunately, they can use a chat server. All of them can enter a chat room at the same time and send messages to each other. Thus, they can continue talking to each other by means of the chat server.

IV. Solution Exploration

There are many potential solutions to implementing a chat server. While the example above is between just three users, a chat server needs to be able to support a large number of users. In order to support a large number of users effectively, messages need to be delivered between users in a timely manner. In doing so, the chat server can serve as a medium for conversation between multiple endpoints.

To minimize the amount of wait time between sending messages and receiving messages, using a multi-threaded implementation seemed the most promising. In order to explore the efficiency of a multi-threaded chat server, we implemented two versions of a chat server, a server which assigns a thread to every user (refer to part V) and a server in which each user must share the same thread (refer to part VI). Then, using quantitative analysis, we determine which implementation is best suited for a chat server.

V. Multi-Threaded Chat Server

In the multi-threaded implementation of the chat server, there are three main classes - Server, Client, and Mediator. The program begins in the Server class. Immediately, main() spawns a Server thread which begins listening on a given port (in our case, port 9090). It is listening for any new connection by a Client.

Before the server accepts a connection from the Client, the user must choose a unique username. Once this is accomplished, each Client spawns their own thread and connects to the server. The server will continue to accept connections as long as there is room in the chat room. This is determined by the variable 'maxUsers,' which is the size of Server's allUsers[].

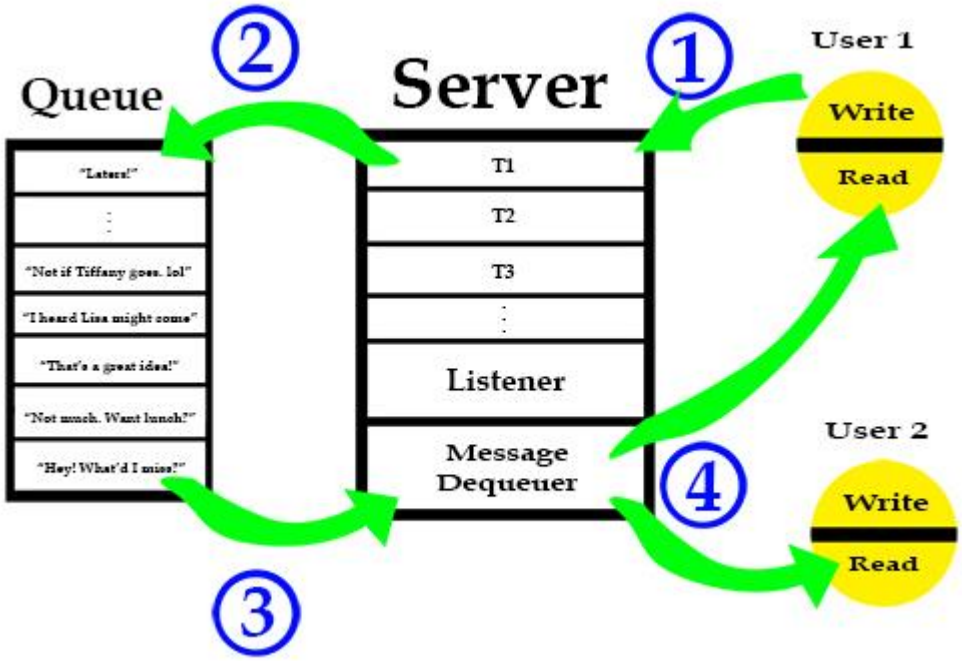
The allUsers[] is an array of type Mediator. Each client connection is assigned to a Mediator in said array. The Mediator object is on the server, and is the means of communication between the Server and the Client. To elaborate, while the server is still running and there is a valid connection between the client and the server, the Client objects take in client messages via the command line and sends it to the server. The corresponding Mediator thread listens for that message, then enqueues it onto a queue on the server.

The Queue object is an array that has two methods enq() and deq(). As a shared resource, a ReentrantLock is used in order to ensure mutual exclusion when accessing the Queue, as demonstrated in the enq() method. The functionality and utilization of the Queue by the Server contributes to the guarantee that every user sees the messages in the same order.

The first instance of the Server object repeatedly calls the deq() method. The server iterates through the array of Mediators, and if the enqueue is successful, it then sends that message to all clients. This is accomplished through the Mediator's writeMessage() method. It is thereby important to note that while multiple reads can occur at one time, only one thread will be writing at a given moment.

When the user is done using the chat server, they can then type "exit" or disconnect the current process. The Server then calls its killMediator() method. Accordingly, if the room was full, there is now room for a new Client to enter the chat room.

To further elaborate on the implementation, please reference the depiction below which models the general architecture of the chat server.



To follow the flow of the diagram, assume there are two users - User 1 and User 2. Each user has two threads running on their machine. One thread will wait for user input and write messages to the server. The other thread will listen for incoming messages to the user. User 1 wants to write a message.

1) User 1 sends his data (or message) through the buffer, which is then read by the Mediator class on the server.

2) The Mediator enqueues this message. The server has one Mediator Thread for each of its client connections, so the server can listen for messages to enqueue from each client in parallel.

3) The first Server thread monitors the queue, and if it isn't empty, dequeues the next message. The Server class has two threads, one of which listens to new connections. The other thread, the message dequeuer, is currently in use.

4) The Server then writes the dequeued message out to every user's buffer stream, which can then read by User 1 and User 2.

VI. Naïve Chat Server

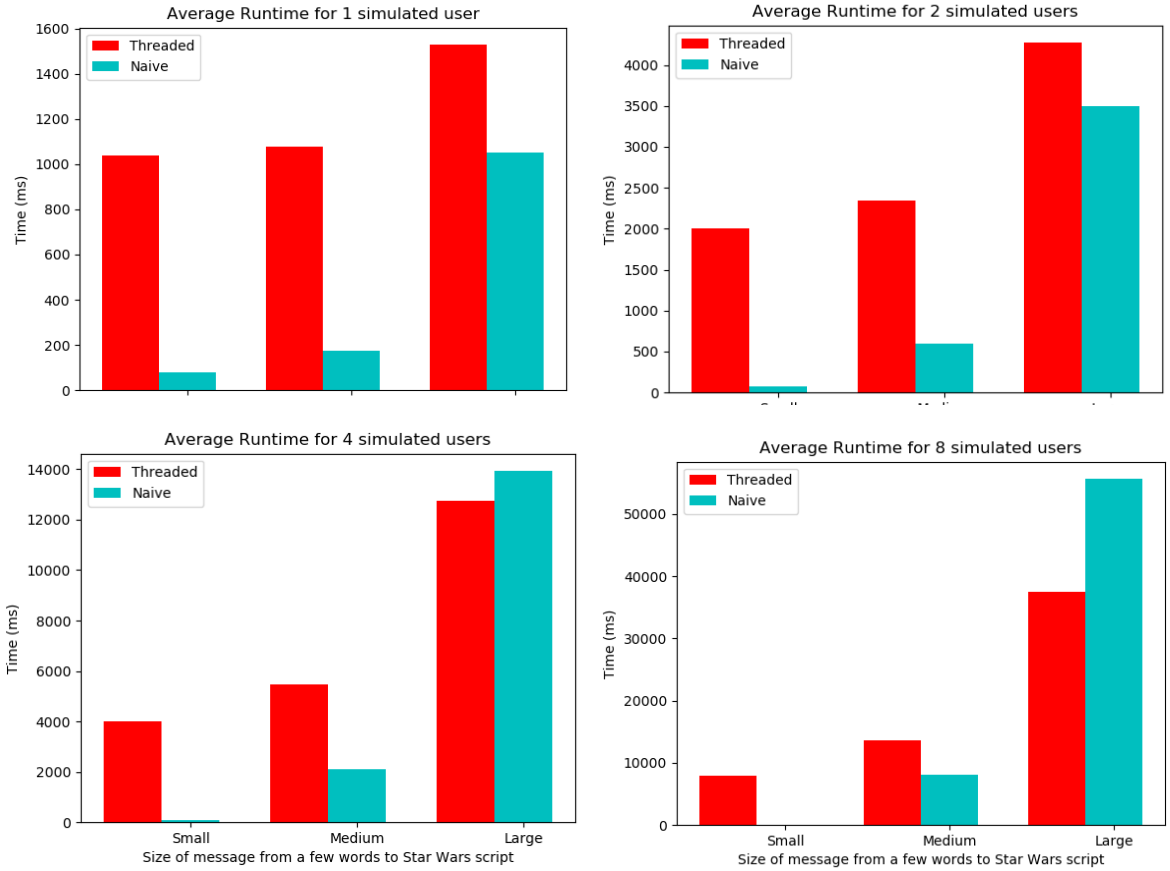
The Naïve Chat Server has many similar qualities to the multi-threaded chat server. Namely, it is still follows a Server-Client architecture. However, one of the key differences is that it does not need to manage a thread for each client, which can be numerous; instead, all client connections are stored in the same Socket array. As a result, the naïve server only has two threads to manage – a thread to listen for incoming connections and a thread to monitor current buffer streams.

The server iterates through all the different connection streams, and if one is ready, it takes the data from the stream, and sends it to all users currently online. A message is ready when the Client writes a newline character to the buffer stream, presumably with other data before it. It then keeps iterating through the connections, watching for more ready messages. If the message is not ready, it will keep going through the data streams and come back to that message.

VII. Data

In order to compare the two implementations of the chat server, we created a test, ServerTest.java, which essentially simulates having multiple users sending large amounts of data simultaneously. We tested each server with respect to two variables – how many users were online at the same time and how large of a message they were sending. The small message consists of about 25 bytes of data, the medium message was the script of *Star Wars: The Empire Strikes Back*, and the large script consisted of all six *Star Wars* scripts concatenated. We also tested each of these for n=1, 2, 4, and 8 users.

ServerTest creates *n* different threads, each of which establishes a different connection to the server that is currently running. Then when they are all connected, a timer begins and they begin to each send the same message to the server in parallel. When they have finished sending their message, they listen for *n* incoming messages. When they receive all *n* messages and join, we know every message has made it to the server and back, and we stop our timer. Each thread was run five times and averaged in the graphs here. The results are shown below:



VIII. Quantitative Analysis

With any concurrent program, there is always going to be an initial overhead. The Mediator must receive the message, store it in a queue, and wait for the message-sending thread to dequeue it and send it to all users. To keep the message order linearizable, there is a spin lock on a queue, which has a lot of overhead. There is also a sequential bottleneck when sending the messages, which is not done in parallel.

However, the purpose of a threaded application is scalability. While it will never offer speedup for a single user, when many users are trying to send large amounts of data to each other, it is expected to be superior. This is demonstrated by our results.

For a single user, the threaded application is far slower. However, even there the naïve version does not send the large message proportionally as well as the threaded version, which has a sublinear runtime for having data several orders of magnitude larger. Then there is a continual spectrum until we have just four users online at the same time, each sending a large message. It is at this point that the threaded application clearly demonstrates superior performance compared to the naïve version. At still only eight users, the threaded application runs almost 50% faster than its naïve counterpart.

Another important thing to note is that, while the threaded application often had a higher finish time, the average turnaround time was much lower given similar runtimes. This means that, on the naïve version, the threads all finished at roughly the same time. On the threaded server, the threads finished at evenly-spaced times throughout the lifetime of the process. This confirms our theory that the bottleneck occurs when sending the message, and it also tells us that there is more to consider than the total time. In practice, many users will see the data right away, and some will wait a little longer. In terms of performance, this is very good.

Without drastic optimization, and with only four users, the threaded application offers a significant speedup. In a project like Facebook Messenger where millions of users are sending images and videos, threading is not only handy, but a must for any kind of meaningful performance.