Matthew Niemiec
Tiffany Chistensen
Project Proposal
CSCI 4830
Prof. Cerny
February 23, 2017

Multi-Threaded Chat Server

I. Introduction

A. Why is the problem interesting

One of the most useful things about the Internet as we know it is the ability to communicate. We can talk to anybody in the world with just a server and two computers with Internet access, and a simple user-friendly chat application helps accomplish this. In order to implement this project, we will additionally have the opportunity to explore client-server socket protocols.

B. Why is it hard

In creating a chat server, there are many difficulties that can arise. One of the hardest elements in the implementation will be the concurrency in itself, as the threading will need to be efficient applied to support multiple clients over the Internet. There will be a lot of programming to do, as creating a server is no trivial task. However, the coding will not be the only challenge. There will also need to be many tests between different versions of the server to figure out exactly how much the concurrency helps. This includes not only writing the different versions, but making sure that the tests used to compare them are fair and equivalent, so that way we can get a meaningful result.

C. Why Concurrency Helps

It is inconvenient if only one person can send information at a time to a chat server. This issue is especially evident when a client sends a message the size of a novel, for this could result in copious amounts of busy wait for other clients. A simple solution would be to place a limit on the size of the message to be sent; however, if there is a high number of users, then there will still be apparent busy waiting and slow communication. In a fast-paced society, people are very impatient and want everything instantly. In order to ensure client satisfaction, threading will speed up the rate in which messages are sent and received on the chat server.

II. Example (What is the problem we are solving)

Our application is going to be a chat application. The challenge is that there may be many users in the chat room, and each user may need to send long messages.

Whenever the user sends a message, they will make a request to get access to the critical section, which will be accessing a shared database. The database will be a MySQL database (or similar), and the user will make queries to it. Users should not be able to make queries at the same time. If they do, the data will be inconsistent or overwritten. Accordingly, the application needs to be fast in supporting multiple requests from multiple users hence the threads. However, that then creates the problem of having our application overlap when accessing a shared resource in the critical sections.

III. Proposed Solution

In order to implement a chat server with fast performance, threading will be applied in the application's implementation. One way of achieving this is by creating a thread for each user when they sign into the chat application. The thread will be an object, and will have their name, credentials, ThreadID, and other information that will be necessary. We will need to implement locks and mutual exclusion. One way of doing this is by using an algorithm such as the Tree Lock or the Filter Lock to grant mutually exclusive, starvation-free access to the database. Then, when the user writes data, the application will broadcast the new message to all users currently connected to the chat room. To elaborate on "broadcast," we will use a TCP connection rather than UDP. In doing so, we can ensure that no data packets are lost. The server will iterate through all clients currently in the chat room and make sure that they each receive every message.

IV. Quantitative Evaluation

In order to quantitatively evaluate the multi-threaded chat server, we will use several methods. To evaluate the efficiency of the speedup between a concurrent and sequential execution, there will be two versions of the chat server. The first version will execute the program sequentially. The second version will execute the program using concurrency and multi-threading. This includes timing how long the code takes to execute. In order to accomplish this, Java's built in CurrentTimeMillis() method will be used to help us calculate the span of time ranging from the moment of threads' spawning to the threads' reaping. We will need to run each test multiple times so that we can get an accurate representation of the average runtime. The results will then be plugged into Amdahl's law

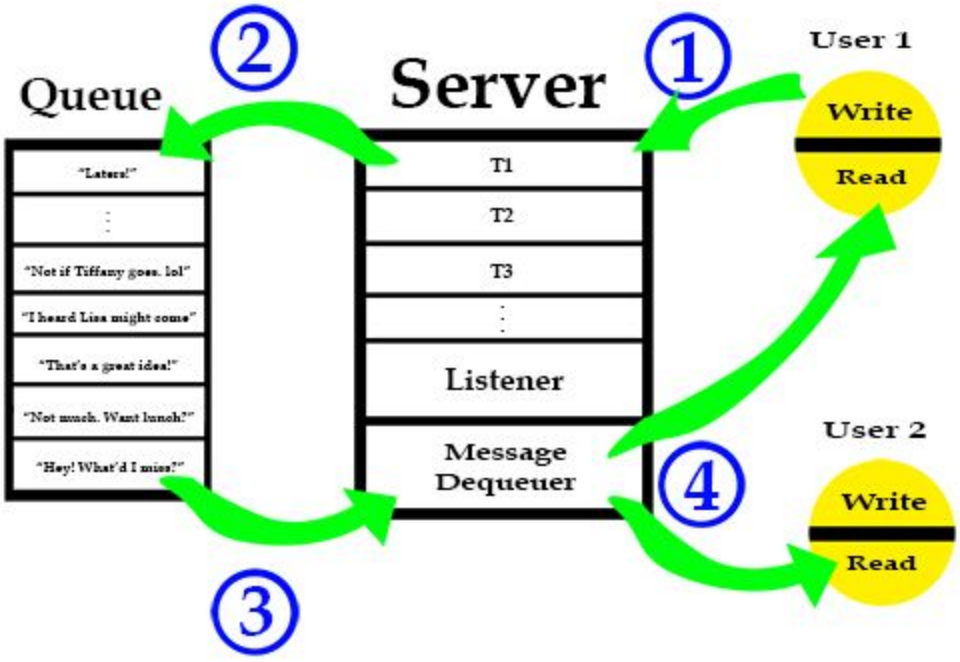
$$\text{Speedup } S = \frac{1}{(1-p) + p/n}$$

Where p is the parallelizable proportion of a program, (1-p) is the sequential portion of the program, and S is the maximum speedup that can be achieved using *n*

processors. The speedup will assumedly prove the benefits of the multi-threaded application.

V. Solution Details

In the implementation of the chat server, there are three classes - Server, Client, and Mediator. The problem at hand is approached as a simple reader-writer problem. Each client (i.e. the user) will be assigned an individual thread on the server. The client will receive messages from the server, who sends the messages to each of the clients. While multiple reads can occur at one time, only one thread will be able to write in one instance. This will be initially ensured by using a queue and re-entrant locks within the enque() method. The locks will provide mutual exclusion on the shared resource (i.e. the queue). However, later we will create something more efficient, such as atomic enqueues. At any rate, the ordering of writes (or messages) will be correct and consistent across multiple thread reads. 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, take into accounts 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. He sends his data through the buffer, which is then read by the Mediator class on the server. The Mediator class's job is to interact with the client from the server. The Mediator then enqueues this message

using some kind of thread-safety. The server has multiple threads running. It has one Mediator Thread for each of its client connections, it has a Listener thread to open new connections, and its final thread - the one now in use - is the Message Dequeuer. Its sole job in life is to monitor the queue, and if it isn't empty, dequeue the message and write it to every user's buffer stream, which is then read by the user

VI. First Results

We have already built a local server and client system using threading. First we diagrammed everything out so that we can get a plan. After discussing this with Professor Cerny and getting feedback, we began implementing the actual code in a Java project. Currently we have a system that starts the server on localhost, and then when a client connects, it infinitely takes input from the user, sends it to the server, and the server sends the information back to all connected parties

VII. Plan for the remaining weeks

The basic foundation of the chat server has been built. Nonetheless, there are still many steps that need to be taken before we reach the final product, most of which can be broken down into two main tasks: 1) We need to move our server to the server provided by the ECEE department, so that it can be accessed from anywhere. Unfortunately, both of us are having difficulty accessing the server via SSH, so we cannot send files to the server via FTP, nor can we connect with Java. And 2) Optimization. While our code works, it is very inefficient. Right now we are using locks to provide mutual exclusion, which, as we've shown in class, is very slow and inefficient. We will experiment with other means of providing thread-safety, including atomic operations provided by Java.

As far as our plan goes we will begin by tackling any remaining problems in the code. We mentioned what the code does, but we have not done extensive testing to make sure there are no corner-cases which could break the code. We also plan on moving our code to the server and rewrite our connection classes accordingly. Next we will work on optimization, which includes testing different types of locks to provide thread-safety. Finally, we will evaluate how well each of these methods works by using tests to evaluate the concurrency and decide which is the best method to use.