Programming Assignment #1
ChatServer

Due Thu, Jan 23

In your first programming assignment, you will write the concurrent server component of a simple web-based AJAX chat system. You will have to spawn threads to handle incoming requests in parallel, use locks to protect access to the shared data structures, and use conditional wait and notify to block asynchronous queries when a new message has arrived.

Overview

The chat system you are to implement has a very simple user interface. To visit the chat room *R*, users navigate to the web page http://***host***:***port***/***R***/chat.html. There they will be presented with a read-only box containing the most recent messages, a text entry field in which they can enter a message, and a Send button. Each time Send is activated, all of the users of the room will immediately see the new message.

AJAX is a popular mechanism for improving the interactivity of web applications. In its most basic form it consists of using JavaScript to perform asynchronous requests to a server, and then using the results of those requests to directly update a portion of the web page, without reloading it. This assignment focuses on the server side of a chat application, so we have already written the HTML and the JavaScript that implements the client.

The Protocol

Your web server must handle three types of requests:

1. `GET /R/chat.html` – The server returns the contents of `chat.html`.
2. `POST /R/push?msg=M` – The server immediately posts the message *M* to the room *R*.
3. `POST /R/pull?last=N` – The server returns all of the messages from room *R* that have an ID larger than *N*. A response consists of one line per message, with each line of the form `(id + ": " + text + "\n")`. If no messages are immediately available, the response may be delayed up to 15 seconds while the server waits for additional messages to be posted to the chat room.

Rooms are created on demand.

What We Give You

`chat.html` – An HTML page with embedded JavaScript that implements the client of the chat protocol. We do not expect you to modify this file at all. We've tested this page on current versions of Firefox, Internet Explorer, and Chrome.

`ChatServer.java` – A very simple HTTP server. This class opens a server socket, accepts incoming connections, extracts the request, and sends a response. This class handles request (1) directly, and implements requests (2) and (3) by calling methods in the `ChatState` class. The starter code is single threaded, and you will be expected to make it multithreaded to complete the assignment.

`ChatState.java` – Holds the shared mutable state of a chat room. The state consists of the 32 most recent messages, and a 64-bit ID that is used by the protocol to identify which messages have already been seen. The starter code is single threaded, and you will be expected to make it multithreaded to complete the assignment.

The source code can be found in `/usr/class/cs149/assignments/pa1` on Leland machines such as `corn.stanford.edu`.

Your Tasks

Modify `ChatServer.java` so that it contains a thread pool of 8 threads. Incoming requests should be handed off to one of the threads in the thread pool to be handled. Add locks or implement concurrent data structures as required to make the chat server thread-safe; there should be no spin-waiting anywhere in your implementation. You are permitted to use ONLY the locking primitives we have discussed in class, specifically:

- the `synchronized` keyword
- relevant methods of `java.lang.Object` (i.e. `wait()`, `notify()`, and `notifyAll()`)
- `java.util.concurrent.Semaphore`
- `java.util.concurrent.locks.Lock`
  as implemented by `java.util.concurrent.locks.ReentrantLock`
- `java.util.concurrent.locks.Condition`
  resulting from calls to `Lock.newCondition()`

Do NOT use any high-level concurrent data structures such as Java's Concurrent Collections classes, or anything else in `java.util.concurrent` not listed above.

Modify `ChatState.java` to be thread-safe. The `ChatState.recentMessages()` method should NOT call `Thread.sleep()` and should instead use a proper synchronization method such as `Object.wait()` to wait up to 15 seconds for new messages to arrive before returning. Note that `recentMessages()` should return as soon as new messages arrive; it should not always wait the full 15 seconds as in the starter code.

Also modify `ChatState.addMessage()` to be thread-safe. Note that `addMessage()` is responsible for waking any blocked calls to `recentMessages()` so that they can return the newly posted messages.

When blocking a thread for any reason, do not use `Thread.sleep()`, as this degrades responsiveness and is considered a poor concurrency practice. Instead use `Object.wait()` or another similar method to make the thread block properly.

Hints

By convention, web browsers only permit 6 simultaneous connections to a single server. Therefore the chat server will not behave properly if you attempt to open more than 6 tabs in the same browser. To test with more than six connections, either open multiple browsers, or use multiple browser sessions (e.g. by using the `firefox -P <profile>` command-line parameter). That said, historically few students have encountered bugs with 8 connections that were not already apparent with 6.

Functional Correctness (50%)

Since the performance of the chat server is limited by network I/O, this assignment will be graded only on correctness. We will test your application with a number of concurrent sessions by opening several browser tabs, typing into each one, and making sure the chat room behaves correctly.

Please note that we will never test your chat server with more than 8 simultaneous connections, but that even with more than 8 simultaneous connections, the server must never drop any connections and should eventually respond to all requests (perhaps with poor performance).

Example functional correctness problems we're looking for include:

- Protocol errors
  - Early empty responses
  - Responses that don't include the latest data
  - Room contents in a different order for different clients

Note that when testing your chat server, we will never test with more than 8 simultaneous connections.

Concurrency and Thread-Safety (50%)

Additionally, we will manually audit your code to evaluate your use of synchronization primitives and verify the thread-safety of your algorithms and implementation.

Example concurrency problems we're looking for include:

- Generic concurrency problems
  - Unprotected access to mutable data
  - Incorrect protection
  - (Potential) deadlocks
  - Missed wakeups
- Poor concurrency practices
  - Busy-waiting (a.k.a. spin-waiting)
  - Use of `Thread.sleep()`
  - Locks held during I/O

Extra Credit (+15%)

Make a magic room named `all`. The system should create this room automatically when the server starts up. Messages posted to any other room should appear in `all`, and messages posted to `all` should appear in the other rooms. (The "hello" messages created by the system do not need to be posted to `all`.) Newly created rooms need not contain messages posted to `all` before the new room was created.

The rooms are required to obey the following ordering constraint: If message *A* appears before message *B* in room *R*, then *A* must also precede *B* in `all`. Arrange your locks and condition variables so that blocked threads are only woken up if a new message is available for a room.

A naive approach to the implementation would be to make all requests to a room *R* acquire two locks: one for *R* and one for `all`. However, this leads to global lock contention, because requests

to all other rooms must now contend on the lock for `all`. Multiple levels of extra credit are available, depending on the quality of the implementation.

1. Correctness (+5%): The implementation is correct and has none of the functional or concurrency problems listed for the normal portion of the assignment.
2. Reduced global lock contention (+10%): The implementation minimizes the time spent in synchronization which has a potential for global lock contention, but does not completely remove it. For example, at this level, an implementation is still allowed to use a single global lock to protect lookups on the map of chat room names to `ChatState` objects.
3. Minimal global lock contention (+15%): The implementation has no global lock contention whatsoever, except in making the initial socket connections, as accepting connections is an inherently serial process. Note that at this level, even lookups on the map of chat rooms names to objects must be completely concurrent.

Please include a note in your `README.txt` indicating which level of extra credit your code achieves (and explain any relevant algorithms).

Note: Achieving the final level on the extra credit may be difficult and time consuming. Make a backup of your code before attempting to work on the extra credit.

Submission Instructions

You should submit the complete source code for your working solution, as well as a brief text file named `README.txt` (maximum 1 page) with your name and SUNet ID and an explanation of how it works and why it is correct.

To submit the contents of the current directory and all subdirectories, log in to a Leland machine such as `corn.stanford.edu` with your SUNet ID and password and run

```
/usr/class/cs149/bin/submit pa1
```

This will copy a snapshot of the current directory into the submission area along with a timestamp. We will take your last submission before the midnight deadline. Please send email to one of the TAs if you encounter a problem.

Compiling

To compile your code, run `javac` with a list of all source files. One way is:

```
find . -name '*.java' -print0 | xargs -0 javac
```

To run the chat server, invoke `java` with the base directory in which the compiled code lives (the current directory `"."` if you just ran `javac`) and the `ChatServer` class:

```
java -cp . ChatServer
```

TCP Server Ports

By default, running

```
java -cp . ChatServer
```

will open a server socket on port 8080. The DNS name localhost is bound by convention to a loopback interface on the local machine, so you can get to the chat page `world` by navigating a browser to

http://localhost:8080/world/chat.html

Only one process at a time may bind a service to a port, so you can only run one chat server instance at a time. If the port is not available `ChatServer` will exit immediately with a `java.net.BindException: Address already in use`. This might happen because an older instance is still running. **This might also occur because another student on the same machine is working on the assignment!** On *nix you can check if the port is already bound with the `netstat` command. (The exact output format will vary.)

```
> netstat -na | grep -w 8080
tcp6       0      0 :::8080                 :::*                    LISTEN
```

You can pass an optional parameter to `ChatServer` to request that it bind to another port:

```
java -cp . ChatServer 43210
```

Working Remotely Through SSH

If you aren't compiling and running `ChatServer` locally, then you can use SSH port forwarding to allow your local web browser to connect to a remotely running chat server, despite firewalls. For command line SSH clients, add the parameter `-L 8080:localhost:8080` to the `ssh` invocation, as in:

```
ssh -L 8080:localhost:8080 corn.stanford.edu
```

This instructs SSH to forward all connections made to port 8080 on the local machine to the address 127.0.0.1 and port 8080 on the remote machine. You can then navigate to http://localhost:8080/world/chat.html on your local machine and be transparently connected to the remote `ChatServer` instance.