Homework # 7
Distributed Computing
due Saturday, December 13th, 2:00 PM

In this homework you will add code to permit a calendar to be served to clients, and to open a calendar on a remote server. You will also code to avoid potential failures characteristic to concurrent programming.

1 Changing Command Pattern

The Command pattern separates the generation of desired actions from the carrying out of those actions. We already use the Command pattern for “undo.” As it happens, it also is very useful for distributed computing—command objects can be sent between sites to synchronize calendar objects (rather than copying entire calendars). However, the current way we have implement Command includes the model (calendar, here) in each command object. That approach must be changed.

1.1 Refactoring Command

Thus the first task of this homework is a refactoring to pull out the calendar from all the individual commands, and instead provide it in the “apply” method. The command log will need to be created for a particular model (calendar). This will render obsolete the use of the Singleton pattern and the convenience add method. The controller will create a command log for a given model (calendar) and pass this to the other parts of the program that need to execute commands. Make sure that the calendar program still works correctly after this refactoring.

1.2 Refactoring CommandLog

It will also be necessary to change the log itself for two reasons: we want to be able to broadcast all the commands in the log to client machines, and we need to be able to add commands to the log arriving non-deterministically from external clients/servers even while the user may be in an undo session. The new commands should be placed conceptually at the bottom of the “redo” stack so that if the user does ‘redo all’ this new command will get executed at the last. In our solution, we completely abandoned the two stack approach, and instead used the Broadcast utility class along with an “undo index” indicating the current state of the system within the command log. Again, this restructuring shouldn’t affect the correct working of the single-user program. After making the changes, make sure the program still works.

1.3 Serialization And Equality

We use serialization to send commands between the client and server, and thus commands will need to be “Serializable.” Furthermore, in order to determine whether a command was successful, it is easiest if commands can be compared using equals (and thus we need to define hashCode as well). Thus next you should update the command class implementation to handle serialization (with serial version id of 1) and equality.

To help you, there is a new test suite TestNewCommand. You will have to remove the indicated comment line so as to make the tests visible. The tests are commented out until this point because otherwise you will have compiler errors.
2 Calendar Clients

The “File” menu gets a new item under “New”: “Connect...” which after checking whether the user wishes to lose a dirty calendar, should put up a small dialog (A JOptionPane input dialog suffices) that asks to which host it should connect. Then it connects to the server on that machine. It should change the title of the frame to “Calendar@hostname”. Then it waits for the initial state of the calendar (see “Protocol” below) and puts up a modal dialog while waiting for the data to arrive.

A connected calendar is never considered dirty (since one can always connect to the server again) whether or not one saves it. If one becomes disconnected with the server, the calendar events stay but the title of the window indicates that it is disconnected. If it is then changed but not saved, it is dirty, of course.

While connected, the “undo” operation is affected: one can use undo-redo to explore previous states, but cannot perform any new actions until all actions are redone. Hence the “undo” capability is for view only; one cannot actually undo anything. The reason is because of possible race conditions: other clients performing actions while we are in the process of an undo. For convenience, you should add a “Redo All” menu item to redo all undone operations, enabling the user to perform new actions.

Assuming all actions are redone, the user can request command actions but they are never applied to the local copy of the calendar. Instead they are sent to the server (see below on the “Protocol”) which may send them back. While waiting for a response from the server, the client should put up a modal dialog. If the command was rejected, a (JOptionPane error message) dialog should be put up.

The dialog (re-)used for the initial calendar and for each request should have a ”disconnect” button that enables the user to give up on the server and disconnect from it. NB: It seems that in some versions of Java, dialogs sometimes remain up even after code has set them to no longer be visible (see Java Bug 5109571). In particular I notice this on my Ubuntu Linux system; the symptom is an unselected window without any contents. So if you notice your dialog which normally has text and a button linger without any contents, don’t try to debug your program. It will go away the next time it is made visible.

Implementation suggestion: The solution defines a subclass of the original command log to handle possibly connected calendars. Methods are overridden so that if a connection has been established, the commands can be diverted to the server

3 Calendar Servers

You should add a new menu “Server” with items to start and stop service. Operations to “Open,” “New” or “Connect” calendars stop service if it is currently active. It must be explicit restarted. It is possible for the program to be connected to a remote server and also serve the calendar to remote clients.

Service is started by opening a server socket and accepting client requests. Each client is handled with two threads: one to send the initial calendar and broadcast all commands from the initial calendar, and the other to receive requests from the client, each preceded by the expected command log size. If the size matches, and service isn’t suspended, then the command is applied to the calendar (or sent to the server, if any, of the calendar!). The client will be notified indirectly through the broadcast.
If the user is in the context of performing an undo operation, then the service is suspended (but not terminated). In other words, requests from the client are delayed until the common log has no undone commands. At that point, client requests will be considered, and either applied or discarded (because a command has been added to the log since the client sent the command). This behavior means that an “undo” operation on the server will not accidentally undo a client operation.

3.1 Implementation suggestion:

Every time we have conceptually a new calendar (after “New” “Connect” or “Open”), we create a new broadcast channel and close the old one, if one exists. When a client connects, we get the initial events (saved!) and the current broadcast and pass them to the new thread in charge of writing things to the client.

4 Protocol

The server and the client must agree on a protocol so that communication can be successful. For this Homework, you are given the protocol that you must follow; thus your client should work successfully with our server and our client must work successfully with your server.

The server establishes a server socket on port 53112 (5+3(‘C’)+1(‘A’)+12(‘L’)). The client opens a connection to this socket, and object streams are established in either direction.

4.1 Client to Server

After establishing the connection, the client writes a 32-bit integer (using writeInt) indicating what extra features it wants. Zero is always legal. Currently, other numbers simply cause the server to slow down unpredictably. If the client sends -1, it will get a server that will try to test the client by pausing at arbitrary points.

Then the client repeatedly writes a positive 32-bit integer and then a Command object (never null). The first number represents the length of the client’s mirror command log. The command is a command that the client wishes to execute next. At the end, the client writes the 32-bit integer negative 1.

\[\text{toServer} - \begin{array}{c}
\text{int} \\
\text{Command} \\
\text{int} \\
\end{array} \rightarrow -1\]

4.2 Server to Client

After a client connects to the server, the server sends a 32-bit integer indicating the number of events in the initial calendar. The initial calendar is the the empty one if the calendar was the result of “New,” the contents of the file in the case of “Open” or the initial calendar sent by the server if the calendar is the result of “Connect.” Then all the events are sent (in the same way that events are written to a file). Once these are sent, the server then sends the client every command executed since the initial creation of the calendar (either by “New,” “Open” or “Connect”). Additionally the server may occasionally write a null object value to determine if the client is still alive. The null values should be ignored.
4.3 Using ObjectStreams

In both directions, the programs should use “object streams.” These are streams that can send entire Java objects, as long as they consist entirely of primitive values and serializable objects.

Make sure to create the `ObjectOutputStream` before the `ObjectInputStream` or else you could get deadlock. (Why?) You may wish to buffer the output stream (I do), but then make sure to `flush()` after you’re done writing something. Otherwise, the other program will never get the data, and you could have deadlock (why?) Don’t buffer the input streams (why not?).

4.4 Using event thread correctly

The AWT/Swing event thread (which handles the UI) should never be used when writing to or reading from sockets. That is why we need two threads on each end of a connection.

In order to send information away (for an output stream), you will find `Broadcast` useful, as explained above. The event thread (and only this thread) can add commands to the broadcast from which objects are taken by the thread in charge of output. The `add` method requires a lock, but this lock is never held for a lengthy period (how can one tell?).

In the other direction, the thread handling input will often need to enter some data into the UI or models. But only the event thread can touch the UI data. So you should use `invokeLater` to ensure the correct thread handles the UI or model data.

5 Race Conditions

Your code will be inspected by hand to see if it avoids race conditions or deadlock. Please add comments to explain anything unusual or why certain unprotected accesses cannot lead to race conditions.

Please also write a file `README` which explains how your code avoid the following concurrency problems:

1. What happens if a new request shows up while an existing command is being executed?
2. What happens if a command is executed while a client is being sent the initial state?

3. What happens if a client never reads its incoming messages after the initial state is sent?

4. What happens if the client sends a request before the initial state is finished being sent?

5. What if the user on the server executes an open (shutting down the server) and then starts the server up again while a client is just getting the initial state from the previous calendar?

6. What if two command requests arrive at the server at the “same” time?

7. What if a client kills the connection while the server is setting up a connection for a different client?

8. What if a client kills the connection while the server is setting it up?

9. What if a client kills the connection just as the server is executing a request from this client?

10. What if a request is accepted by the server and sent back before the client has put up the wait dialog?

Finally, you must document the locks. For every object that your code uses as a lock (for synchronized statements), you must document in README the state that is being protected. You must also give the order of acquiring locks, if it is ever possible that a thread could have two locks.

Our solution never uses synchronization or volatile except in the utility class Broadcast. If you follow this example, you will have less to document!

6 Hacking and Denial of Service

Distributed programs, in which pieces communicate over sockets often suffer from vulnerability to various kinds of attacks. First there are those that use vulnerabilities in the input and output to cause the internal structures of the attacked system (the calendar server) become invalid. Your code for must protect itself against such attacks. (It’s not necessary to protect the client from the server: the situation is asymmetric: a server accepts connections from anywhere on the Internet, but the client is attaching to a known server.)

Another kind of attack is the “denial-of-service” attack. A denial of service attack is one in which an attacker uses the service in such a way as to prevent others from using it. Your program does not need to protect against these attacks.

7 What You Need to Do

1. Update the Command pattern.

2. Make classes serializable.

3. Update CalendarFrame to handle new UI requirements.

4. Answer questions and document locks in README.

Leave the “wait dialog” (in the client) until last. It is tricky and messy to get right.
8 Files

We provide you with a “bare” repository in your homework7 directory including the solution to Homework #6 and the following files:

src/TestNewCommand.java New JUnit tests to be used after refactoring.

src/edu/uwm/cs552/util/Broadcast.java The broadcast utility class.

README Skeleton.

Your completed README should include:

• Explanation on how each of the ten race conditions is prevented, or why it was infeasible to avoid.

• List each object used as a lock and what state it protects.

• Give the partial order of locks: what order they may be acquired if more than one is ever acquired by the same thread at a time.