

# Homework # 7

## Concurrency & Distributed Computing

### due Tuesday, December 18th, 12:30 PM

(As with previous homeworks, undergraduate students may do the work jointly in groups of 2, if desired. Discussions between any students is always permitted when credit for ideas is given.)

In this homework you will add features using threads and interprocess communication. You will also code to avoid potential failures characteristic to concurrent programming.

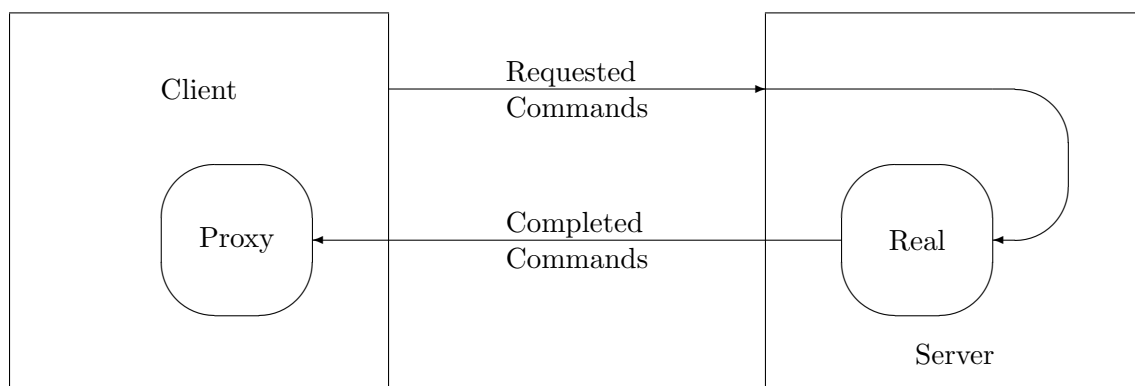
## 1 Serialization

In order to make this Homework shorter, we will be using `ObjectOutputStream` and “serialization” to write reservation structures instead of extending our XML capabilities. This requires, however, that the serialization of objects be compatible, and that the various classes making up reservations be `Serializable`. Furthermore, since rooms have identity, they need to be serialized using replacement objects. This is done in the solution to Homework #6. Thus, for these and other reasons, you should start with the solution to Homework #6 to do this Homework.

## 2 Distributed Computing

To give you a taste for the difficulties of distributed computing, in this Homework, you will program the reservation system to fit in with client-server computing. Instead of a simple driver, there will be two separate main programs: a `ReservationServer` that will behave similarly to the existing program except that it will also accept connections from external clients. The other main program is `ReservationClient` which connects to the server and displays the current state of the reservation system on the server. Clients can add, remove and change reservations, but these operations are sent to the server where they are acted upon. For simplicity, `New`, `Clear`, `Open` and `Undo` do not need to be supported on the client. It should still be possible to `Save`. Communication between the client and server uses sockets and object serialization.

### 2.1 Distributed Structure



The client program will locally maintain a shadow copy (a “proxy”) of the state of the system (scheduled reservations). When it wishes to make a change (add, change or remove), it will send a command to the server. The server will then apply the command locally and the client will receive

confirmation through commands to be used to update the proxy. The client should never directly apply commands to its proxy.

The situation is more complex because:

1. The requested commands are accompanied by a request number and a version number; the version indicates the state of the system, as seen by the client when the command was issued.
2. Interspersed with the completed commands are acknowledgments of requests. Each acknowledgment includes a request number and a status string, which is empty if the request was performed.
3. Each completed command is accompanied by a new version number.

Version numbers are used because a request may be made using old information. You should arrange that the version number sent before a request is the version in effect when the dialog was put up. In this way, the client user doesn't try to remove a reservation that has been changed or removed since the dialog went up. This condition is more strict than necessary, but is simple to implement.

## 2.2 Establishing a connection

The server, when it is started, opens up a “server socket” on the port specified on the command line, or 50505 by default. The client will attempt to connect to the server using the port given on the command-line, or 50505 by default. Once a connection is made, a bidirectional socket is created and object streams are opened in both directions. NB: to avoid deadlock, it is important that you create the `ObjectOutputStream` before you create the `ObjectInputStream`.

When a new connection is established, the first thing that happens is that the server will write through the socket to the client all reservations in the system (for all rooms) in no particular order and then write a version number (an `Integer`). This version number must be used in (future) requests.

The server should be able to handle at least 10 clients at a time. It will need to have a thread for reading requests and one for writing back acknowledgments and completed commands.

After every change in the state, *all* clients are informed of the change. You will find this easiest to implement by adding methods to add observers to the commands log (which is different than the existing observer—don't modify that, it's used to handle the undo dialog). You should add a single observer that communicates with each client thread. You will find `java.util.concurrent.ArrayBlockingQueue` useful to communicate the commands between the observer and each client handler's output thread. If this queue gets backed up too much, the server may decide to give up by closing the socket, but this functionality is optional.

## 2.3 Waiting for the Server

When first establishing a connection and when waiting for the server to acknowledge a request, the client program must not give a false impression to the user. In particular, it must put up a dialog while it is waiting for a response from the server. This dialog should include text that indicates what the waiting is. If waiting for an acknowledgment, then when the acknowledgment arrives, if there is an error message, the dialog should be changed to give the error message. The waiting dialog should include a button to “stop waiting.” If a “negative” acknowledgment (one with an error message) arrives after the dialog is down, a new dialog (which can be a `JOptionPane`) should be displayed.

The AWT/Swing thread cannot be used to do I/O. All I/O, especially connected to the server should be done in separate threads. You will find it easiest to read in one thread and write in a different one.

## 2.4 Formats

The server will expect the input from the client to consist of intermixed version numbers (**Integer**) and requests, where a request is simply a **Command** followed by a request number (an **Integer**). There is no “request” class.

The output from the server to the client consists first of the extant **Reservation** objects followed by the current version **Integer**. Then follows intermixed acknowledgments and completed commands. An acknowledgment simply consists of a **String** followed by a request **Integer**. If the string is not empty "", it represents an error message; the request was *not* carried out. A completed command is a **Command** followed by the new version **Integer**.

## 2.5 Undo

For this homework, **Undo** can only be carried out by the server. While the **UndoDialog** is active, the server will refuse all requests with error message “Server is busy.” When the undo dialog is done, any new commands (being undo’s of previous commands) are sent to the clients in the normal manner.

## 2.6 Closing down

If the server executes **New**, **Clear** or **Exit**, all existing clients should be terminated. If a client notices that the socket is closed, it should permit the user to continue to view reservations, but not permit changes. For simplicity, to establish a new connection, a new client must be started.

## 3 Race Conditions

The code we are currently working with has no protection against two threads accessing the same state. Add synchronization to the code so that two threads never access the same mutable state at the same time. In special cases, you may also use “volatile” fields. Make sure that you do not introduce deadlock—in particular you need to ensure that repaints are not blocked for a noticeable amount of time. Make sure to test it thoroughly.

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 if a reservation is added, changed or removed while the server is writing the initial state to a new client? The initial state received **must** be a consistent one, showing either the entire state before the change or the entire state after the change. And the version number sent must reflect the version that was sent.
2. How do you handle two requests (reservation to add, change or remove) coming in from two clients at the “same time” ? How do you ensure that all threads receive the notifications and version changes consistently (in the same order)?

3. What if a request to add, change or remove a reservation comes in while a file is being read (for `Open`) on the server? How do you ensure that the client command doesn't appear inside the compound command created by `doOpen`?
4. What if someone presses "Stop waiting" on a client just as an acknowledgment (error or not) comes in? How will you avoid confusion?
5. How do you make sure that no command from a client slips in just as the undo dialog is coming up or going down?

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.

## 4 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 reservation 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.

## 5 Files

The solution files for Homework #6 are in

```
/afs/cs.uwm.edu/users/classes/cs552/solutions/homework6/
```

You should start with our solution.

The programs must be runnable from the command line. In other words, on a grid, it must be possible to

```
grid.cs: cd homework7
grid.cs: java ReservationServer 50599
grid.cs: java ReservationClient grid.cs.uwm.edu 50599
grid.cs: java ReservationClient pabst.cs.uwm.edu
```

Your `README` should include:

- Explanation on how each of the 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.