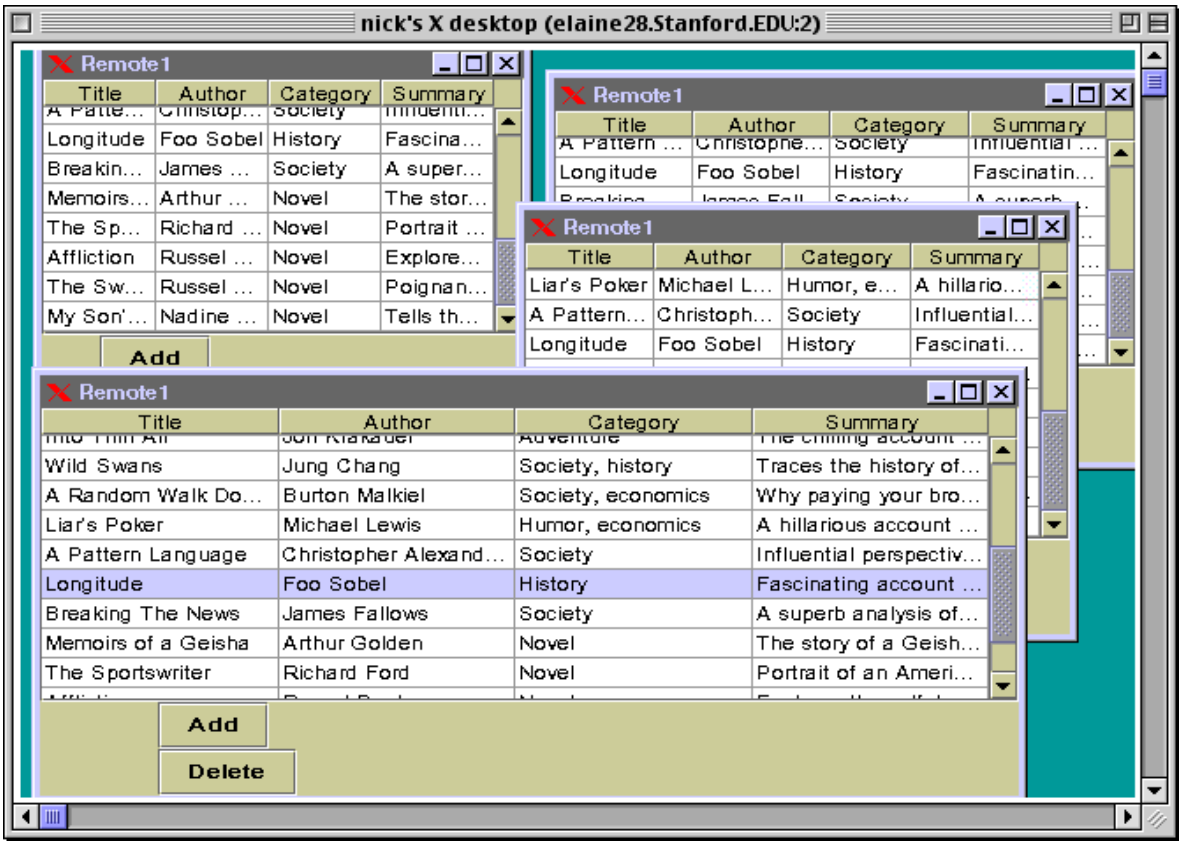
HW3 RTable

The goal is to create a typical Table / TableModel setup with the tiny variation that...

- There are many tables, but just one table model.
- The tables are all located on separate machines and use RMI to synch everything.

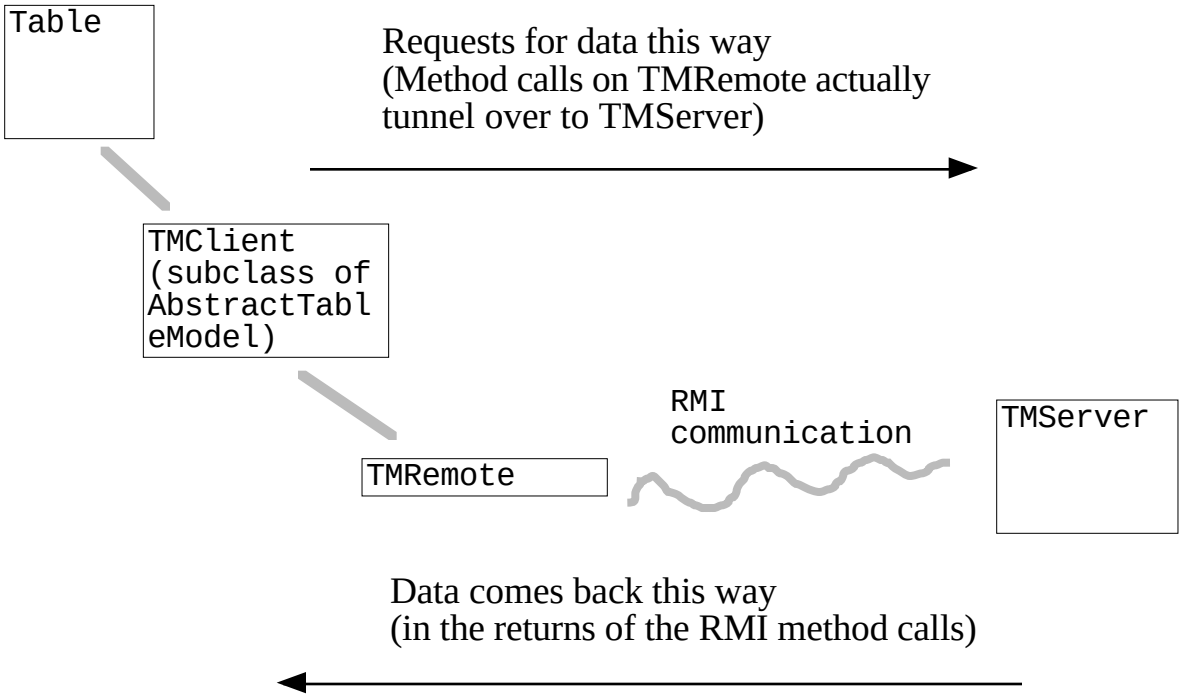
This is the last assignment, and it is due Thu May 25th.



The Plan

- Have a regular table on the client side
- The table uses a TMClient table model, subclassed off AbstractTableModel in the usual way. TMClient does not store any actual data; it has a pointer to a TMRemote that appears to store the data. All the table model messages, such as getDataAt(), get forwarded on to the TMRemote.

- The TMRemote is actually the stub (automatically generated) of the TMServer.
- The TMServer is the one object in the whole system that actually stores the data, everybody else just passes the buck.



(There are many classes mentioned here — in order to keep the staff relatively sane, please use the same class names in your solution: TMRemote, TMServer, TMClient, TMFrame, XObjectRemote, XObjectServer, XObjetListener.)

Phase 1

The first goal is to get the simplest sort of table working through this RMI channel. We'll just send getData() and setDataAt() requests down the pipe, and not worry about firing any update events. The tables will tend to get ok looking data just through the natural occurrence of their repaints as the windows are moved and resized.

TMRemote

An interface that extends java.rmi.Remote to define the messages for the remote channel.

TMServer

Implements TMRemote. Stores the data to respond to all the messages. There will be one, centralized instance of TMServer in the working solution. All the remote methods coming into TMServer should be synchronized in case there are multiple tables out there. The main() in TMServer should create and register a single instance of the class — use your leland id as the first part of your service

name so that it does not conflict with anyone else's. The single command-line argument to TMServer should be the name of a tab-delim data file that the TMServer reads in as its initial contents (we'll use books.txt again). All the cells should be editable. The number of columns and their labels will not change once the file is read in. TMServer should print a "TMServer: server bound" message once it is successfully read in and bound.

TMClient

This is a "glue" class between the Table on the client side and the TMRemote which is the channel to the server side. TMClient should be a subclass of AbstractTableModel and should own a single TMRemote object to which it relays messages. Use Naming.lookup() to get the TMRemote. TMClient should not send any of the fireXXX notifications – it should pass the messages through to the TMRemote.

It would be nice if the TMRemote could just *be* the TableModel for the table directly, but this is impossible for the following reason: all of the messages in TMRemote throw RemoteException, but the standard Table classes don't know how to handle that from their table model. So TMClient must sit between the two, taking AbstractTableModel messages on the one hand, passing them off to TMRemote, and dealing with the exceptions as they occur.

TMClient should catch all RemoteExceptions, print out a notification error message, but then blunder forward with the most harmless data like "" for Strings, 0 for numbers, and so on.

TMFrame

Just a JFrame that contains the JTable and everything. The main() in TMFrame should create a single JFrame (Actually for testing, you may want to have it create two TMFrames to observer the data flow through the server. For the final version, please have it create a single TMFrame.) Make it so closing the window exits the program (use the window listener or its setDefaultCloseOperation) and / or install a "Quit" button.

Exceptions

To aid in tracking down problems, all RemotExceptions should print something out prefaced with the name of the class. Do not silently drop any RemoteExceptions.

```
...
catch (RemoteException e) {
   System.err.println("TMClient exception: " + e.getMessage());
}
```

The TMClient should try to proceed on its exceptions. TMServer can exit on its exceptions.

Printing and Testing

Here's your new best friend: System.out.println(). The number of classes, machines, and VMs involved in our modest little exercise is too much to fit in a

debugger. It's best to a) run each of the major players in the foreground of their own terminal job, and b) have them print out things when they do anything. You can comment it out later. Also, use a 2-by-2 test file for starters.

Logistics

Everything will take place in one directory, but on several machines...

1. Compile all java files

2. Run "rmic TMServer" to generate the _Stub and _Skel files for the TMRemote / TMServer system. You probably only need to re-rmic when you change TMRemote (or PipeRemote) (Macintosh: I've only run rmic on unix, but if you copy the _Stub.class file back to the Mac and incorporate it into your project, you can run the client. This may not work in Phase 2 however.)

3) Have the file "rtable.policy" in the directory

4) Pick a random magic port-number in the range 2000-65000 and use it for the duration of the assignment.

5) Run rmiregistry with your magic port number as an argument in the background on the server machine. If some bad person is using your magic number, there will be an error. Pick a new magic number (first making sure that the "bad person" isn't yourself!). You can leave the rmiregistry running for several runs.

```
server> rmiregistry 35039 &
```

6) Set up the "rjava" alias as in the lecture example. It will be used by both client and server sides. I got so sick of typing it, I put it in my .cshrc. (Also, there should be no setting for CLASSPATH — we're just relying on Java 2 class finding in the current directory.)

```
%> alias rjava java -Djava.security.policy=rtable.policy
```

7) Run the TMServer, it should block (we'll kill it with ctrl-c when the time comes). The first argument to TMServer should be the port number. In your code, append the port number to the string "localhost" to make "localhost:35039" as the rebind call. The second argument should be the data file to use.

```
server> rjava TMServer 35039 books.txt
TMServer: server bound
```

8) On the client machine (but in the same directory), launch your TMFrame passing it the name of the server machine and the port number. You can run it in the background or not. Try running a couple TMFrames when you're feeling brave. They'll work pretty well, except for the lack of updates.

```
client> rjava TMFrame elaine33:35039
```

9) Use the unix "ps" command to list your running processes, and use "kill -9" to kill them off.

```
elaine21:~/java/rmi2> ps
   PID TTY      TIME CMD
 16566 pts/34   0:01 tcsh
 17610 pts/34   0:00 zwgc
 20546 pts/34   0:01 rmiregis
 16586 pts/34   0:00 zwgc
elaine21:~/java/rmi2> kill -9 20546
elaine21:~/java/rmi2>
[1]    Killed                        rmiregistry 32456
```

Phase 1 Milestone

You should be able to fire up a TMServer with some data in. You should be able to bring up several TMFrames pointing to that server. The TMFrames should be able to scroll around in the data. They should be able to edit cells, and hitting return after editing should send the change over to the TMServer. The changed data won't show up in the other frames right away.

Phase 2

Phase 1 has the core functionality, but it has many small deficiencies, all of which will be fixed in Phase 2. You can fix these in whatever order appeals to you...The updates are the hardest part.

1. Add Row / Delete Row

Add an Add Row button which adds a new, empty row at the bottom of the table. Similarly, add a Delete Row button which deletes the selected row. The whole TMClient-TMRemote-TMServer chain will need some new messages to support this feature. We're not going to ever change the number of columns. You can put in fireXXX messages in TMClient temporarily, but they will need to come out when you fix updates the right way below.

2. Caching

The TMClient is pretty prolific in the way it generates getData requests. Rather than going all the way back to the server for every request, the TMClient should keep its own memory cache. We'll just cache the getData() calls, and let the rest of the messages go through. Check each getData() to see if its in the cache first. Update the cache according to setData() calls as they go out.

Implementation idea: You could build a 2-d structure that eventually shadows that of the TMServer. But I couldn't resist this cheesy approach because it's so

easy and it supports the sparseness of the data nicely: just use a Hashtable of arrays. Use the row as the key. It's a little slower than it could be, but it's very convenient. The in-memory hash table is so much faster than network traffic. The Hashtable is just one way — feel free to do it however you like. The cache will need to update itself to deal with row deletion (below).

3. Server File Saving

The server should keep a separate thread that checks approximately every 20 seconds to see if the in-memory database has been changed, and if so writes it back out over the original file. The file-write operation can acquire and hold the TMServer lock while it goes so the data is not changing out from under it. An industrial strength implementation would need to do something clever to avoid locking out all the table threads during the potentially slow file write (make an in-memory copy, or make add / delete / edit operations atomic in such a way that the file-write pass can co-exist with them).

4. Updates

The TMServer is the one, centralized entity that hears about all changes (edits, adds, deletes), so it needs to be the starting point for notifications going out to all the clients.

The short story for this is: use RMI for each client to register with the server. The server can then have a "notifyAll" behavior where it uses RMI to update back to each of the clients with messages like "row 3 deleted" or "row 6 edited". Specifically...

XObjectServer - XObjectRemote

We'll use a relatively simple XObjectRemote / XObjectServer system that supports a single send(Object) message. The holder of the XObjectRemote can do a send() on it, to convey any serializable Object over to the XObjectServer. Arrays, Strings, Integers (the class) are all serializable, so messages can be built out of those with no extra work.

```
import java.rmi.*;
import java.rmi.server.*;

public class XObjectServer extends UnicastRemoteObject
      implements XObjectRemote {

   XObjectListener listener;


   public XObjectServer(XObjectListener listener) throws RemoteException {
      super();
      this.listener = listener;
   }

   public void send(Object message) throws RemoteException {
      listener.send(message);
   }
}
```

When the holder of the XObjectRemote does a send(), the flow of control tunnels over to the send() in the XObjectServer. The send() method in the XObjectServer is what runs when a message is coming in. The nice way to keep this general, so that XObjectRemote / XObjectServer can be used in any context without change is: create an interface called XObjectListener that defines a single send(Object) method. The XObjectServer takes a pointer to a "delegate" XObjectListener in its constructor. When the send() happens on XObjectServer, it calls send() on its XObjectListener. Any object that implements XObjectListener get the notification. Its similar to Swing's built-in listener structures, except we allow just one listener instead of a collection of listeners. So to get messages, create an XObjectListener with your implementation of send(), and pass an instance of your listener to XObjectServer.

Register

Add a addXObject / removeXObject feature to TMRemote / TMServer so that clients can send an XObjectRemote over to the server to be stored as the communication point for that client. The XObjectRemote points back to the XObjectServer on the client side.

SendToAll

Add a sendToAll() method on TMServer that takes an Object representing some message, and sends it to all of the registered XObjects. Since sendToAll() is probably being called from a synchronized method which is locking up the whole TMServer, it should actually spawn off a bunch of threads to do the sending.

Communication

The TMClient can listen for messages coming in on its XObjectServer. This whole structure can be used to convey updates about the TMServer state...

- Because the number of columns does not change, the edits that we care about are: row added, row deleted, row edited.
- The TMClient should not do any fireXXX notifications when it sends messages back to the TMServer to change the data state.
- The TMServer should send out notifications to all registered clients when a state change happens. The encoding of this message is up to you — anything that can be conveyed in an XObject.
- The TMClient will receive messages from the TMServer on the XObject. It should a) throw out parts of the cache if necessary (especially for delete), and b) do the appropriate fireXXX event. The result is that if a Client requests, say, a row delete. The request goes out to the server, causing a message to go out to all the clients, and so eventually the client hears about their own delete, and then does the fireXXX notification. It may seem a little roundabout, but it has the advantage that the update scheme is not fixed to any client — so when you get it debugged for one client, it's most likely to scale

correctly to many clients. This is the "run everything through one code path" rule of thumb for reliable software.

- (HW2) Note that the XObject messages are coming in a thread which is not necessarily the Swing thread.

Testing

You can get it working with just one client first. Test: editing a cell, adding a row, and deleting a row.

Mega Testing

Bring up as many clients. Some can be running on different machines from the server (or different OS's / architectures. Do an edit. The change should propagate over to all the clients. Change the first name of an author to "Foo", and which your little jest propagate around. Enjoy.

Problems We Ignore

There are potential problems if two clients are editing the same cell at the same time. We won't worry about that. A simple solution would be some sort of visual cue that showed where other clients where editing. There's also bad cases that could come up where an update has propagated to some tables but not others, and so when one table says "delete row 6" it means something different from when another table says "delete row 6" because they are in different update states. We'll ignore that too. The simplest solution in that case would be to use a row identifier more robust than just row number in the ordering. The TMServer could generate the identifiers.

Deliverables

We should be able to run one server and many clients from your directory (but on many machines) by...

1. Compiling all java files and rmicing TMServer and XObjectServer

2. rmiregistry port-num &

3. rjava TMServer port-num books.txt / / start the server

4. rjava TMFrame server-name:port-num / / launch a client (repeat)