Distributed Collaboration - Assignment 4: Towards and Beyond Google Waves

Date Assigned: Wed Oct 21, 2009

Completion Date: Thu Nov 5, 2009

Objectives:

- Understand automatic replication of logical structures
- Understand and Implement distinguishing aspects of Google Waves
- Understand and implement flexible coupling
- Learn how to keep collaboration awareness and model functionality separate
- Implement a factory pattern

In this assignment you will use Sync to re-implement and extend the N-user instant messaging functionality of the previous assignment. The assignment has been broken into parts to allow you to do it incrementally. The sync library is available from the class page, and it includes ObjectEditor. So replace your current ObjectEditor library with this library.

Work alone on this project.

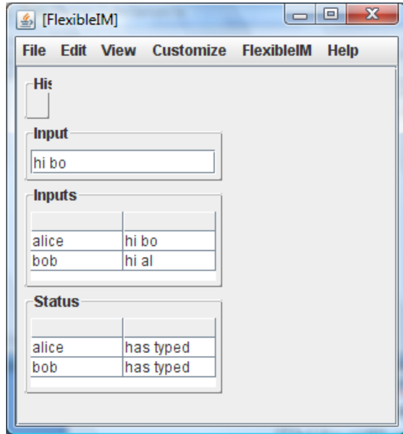
This time I am showing you the user-interface of my implementation of the assignment. It is simply an illustrative example. You are free to choose different implementation as long as you follow the requirements.

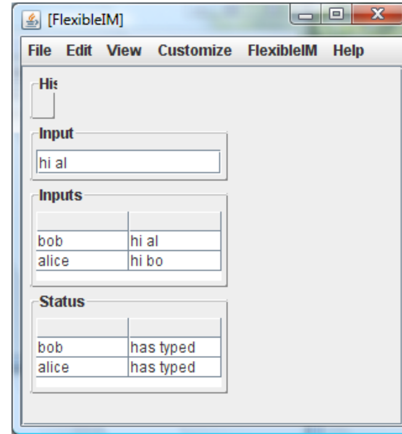
Re-implement IM functionality using Sync

Re-implement your IM tool from the previous assignment using Sync. Do not use RMI directly. The re-implementation should require no changes to the distribution-unaware model. (In fact, if your model implements Remote, Sync will not be able replicate it as it used RMI serialization to create replicas.)However, you will have to write a new main program that uses the Sync API. As Sync has a name server, your IM tool no longer has to assume that there is only one IM session in the world. The name of each registered model is essentially a session name. If Sync works correctly, the order of the messages in the IM history should be identical, which was not a requirement in the previous assignment. Use the replicateOrLookup() call to interact with the name server - do not use the replicate() and lookup() calls directly.

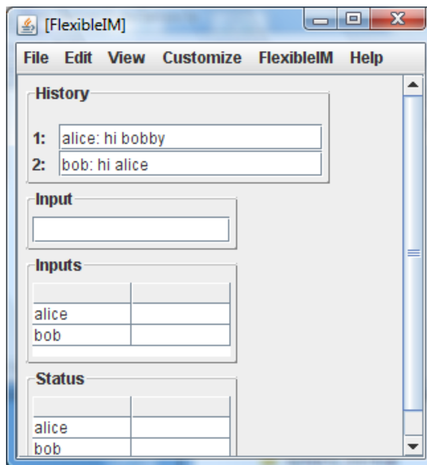
Towards Google Waves

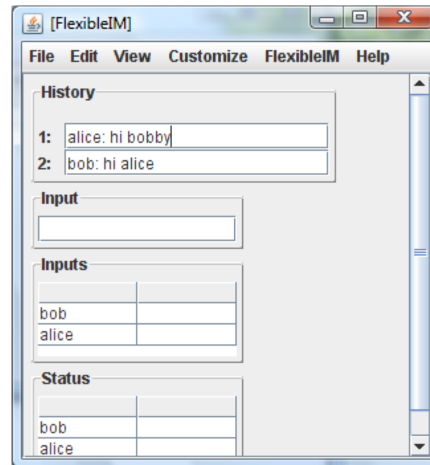
Google waves allows users to see incremental edits to the input message. Implement this functionality. That is, allow each user to see each edit of all other users in the session, as shown below. Do not get rid of the status information, as it will be useful later.





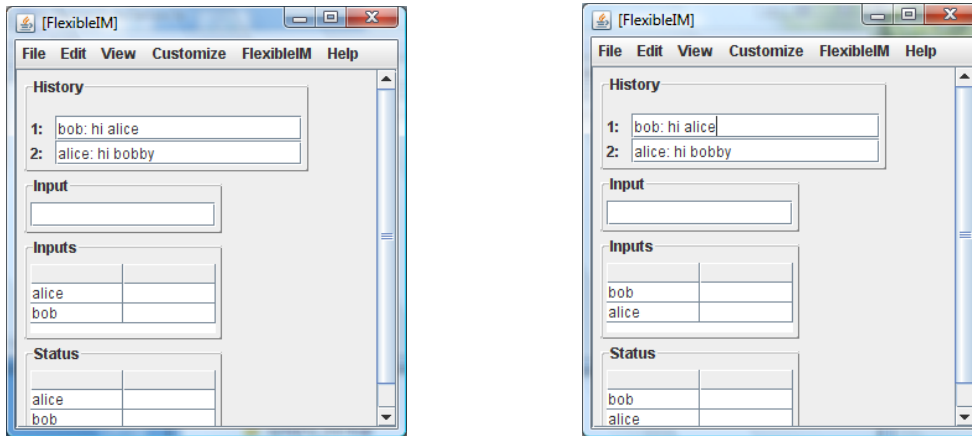
Also, as before committed messages should go into the history. However, a committed message, like a Google Wave email can be edited. In the following screen, the message “hi bob” is being edited to “hi bobby”. Each user should be able to see the incremental edits to a committed message. Allow users to edit all committed messages, even those they have not committed.





To allow history messages to be edited, you will have to use the default display ObjectEditor creates for the history. If you have bound the history to a JTextArea, ObjectEditor will not process your edited changes (correctly).

When a committed message is re-committed by pressing Enter, as in Google Waves, the message should become the latest message in the history, as shown below.



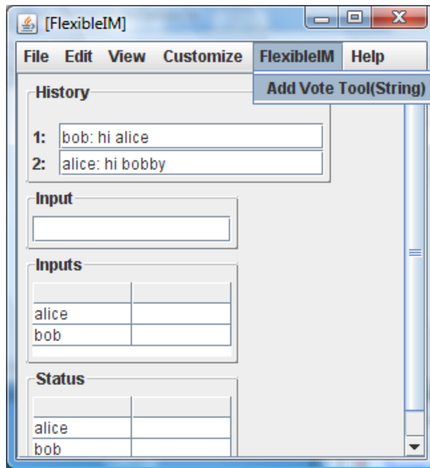
When an element of an instance of AListenableVector (or any class implementing the programming pattern used in AListenableVector) is committed, ObjectEditor calles the setElementAt method of the object.

Another interesting feature of Goole Waves is that a live synchronous application such as vote gatterer or a game can be sent in an email. Each user who receives the applciation can interact with it. Implement this functionality in your IM tool by allowing users to add post instances of the predefined Sync Votetool . This tool defines a constructor that takes a Yes-No question , allows users to vote on the issue, and and keeps track the total number of votes.

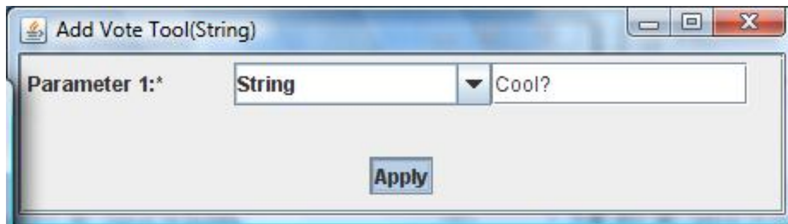
The following screenshots shows the use of the tool in our IM. To understand how you can use ObjectEditor to implement this user-interface, you need to know that if class C implements a non-pattern public method M, that is a method that is not a read or write method defined by the Bean, Vector and Hashtable patterm, ObjecteEditor creates a menu names C with a corresponding menu item whose name indicates the signature of the method M. Thus, if class FlexibleIM defines a method with the signature:

public void addVoteTool(String issue)

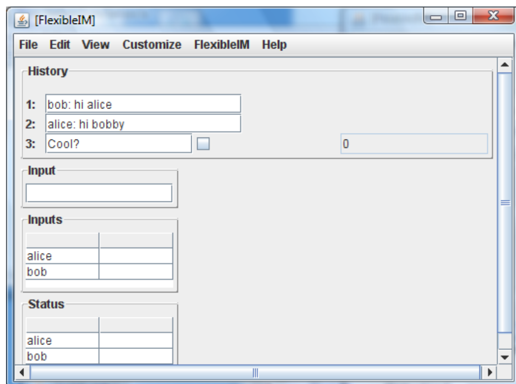
ObjectEditor creates the menu item shown below.

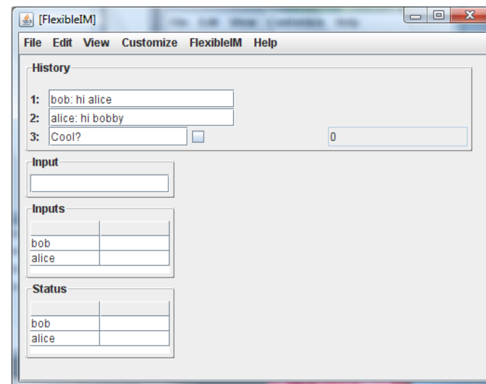


Selecting the menu item creates a dialogue box that allows the user to enter the values of the parameters of the method.

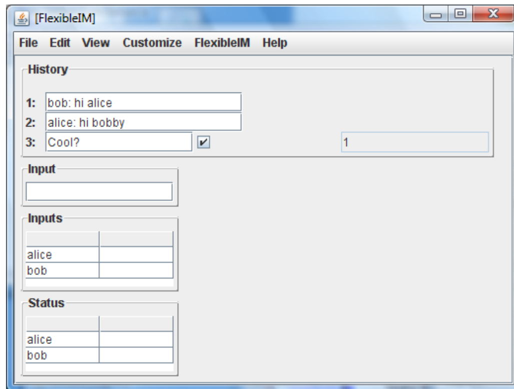


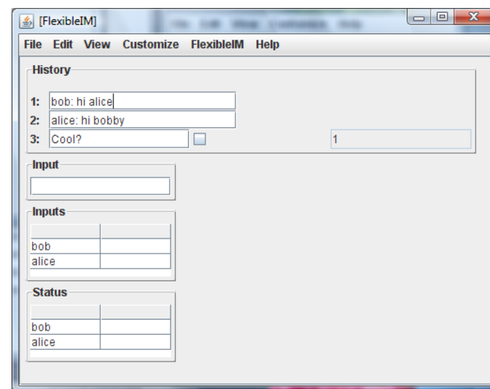
Pressing the Apply button executes the method. In this example, the method adds an instance of the Sync VoteTool to the history.





If Alice votes yes, the new cote count is shown to all users.



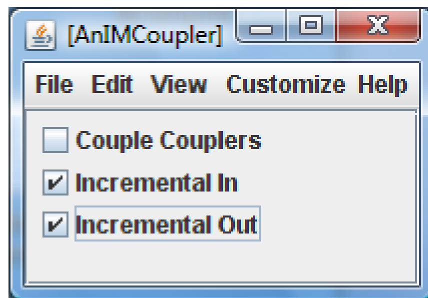


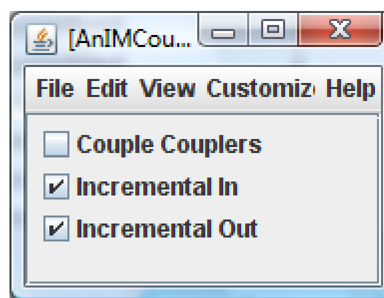
Similarly, Bob can vote yes to increment the count.

Beyond Google Waves: Sophisticated Flexible Coupling

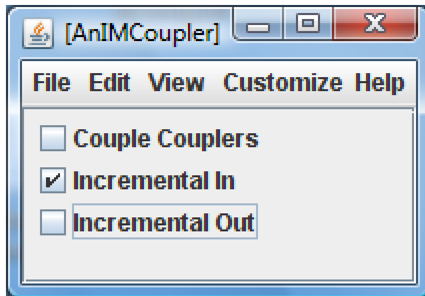
According to Lana, at least one review of Google Waves complained that it is uncomfortable to have others see one's incremental input and if there is a way to disable it, it is easy to find it. Therefore, let us add functionality that allows users to determine the coupling between their IMs that probably goes far beyond what is implemented in Waves.

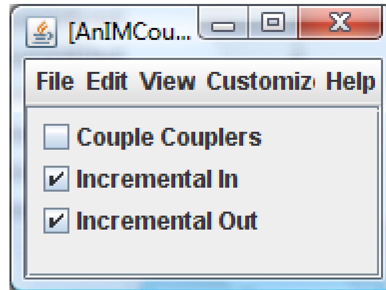
This part of your assignment should involve no changes to the IM model. In fact, your IM model should be completely unaware of the Sync API. As the screen shots show, the coupling functionality should be managed by a separate object. Each user sees a "replica" of this object, as shown below.



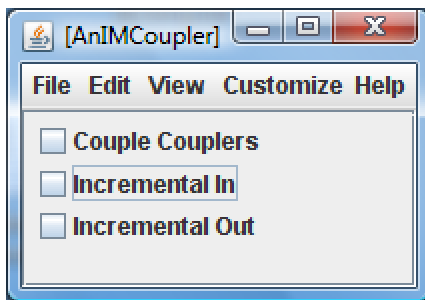


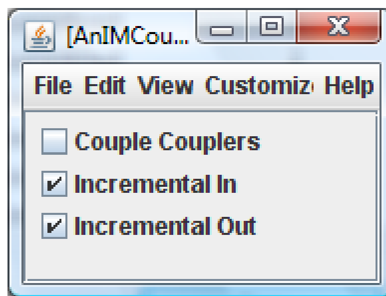
The IncrementalOut property of each user's coupler determines if others can sees his/her changes to the input message. Thus, if Alice sets it to false, and Bob sets it to true, her incremental input will not be seen by Bob, but Bob's incremental input will be seen by Alice.



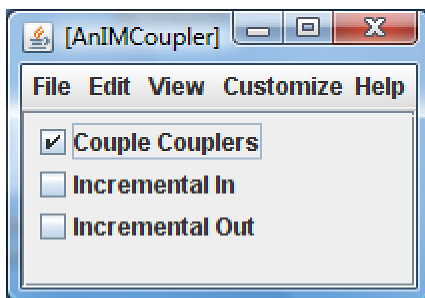


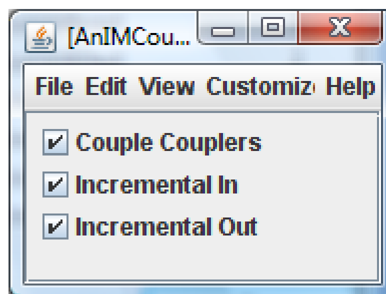
The IncrementalIn property determines if the incremental input of others is seen by a user. Thus, if Alice sets it to false, and but Bob sets it to true, and there is some other user, Cathy, in the session, whose IncrementalOut property is set to true, then Cathy's incremental input will be seen by Bob but not Cathy.



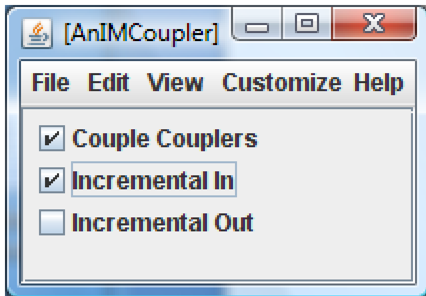


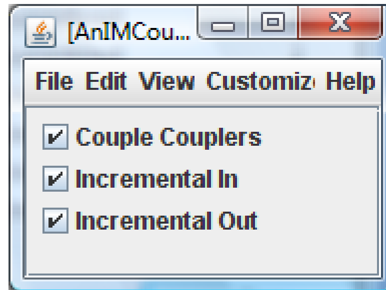
It should also be possible to couple the couplers. A side effect of changing this property to true (false) is that this and all subsequent changes to all properties of the coupler object are (not) replicated on all other coupler objects. Thus, now that this property is true, the change to it is sent to Bob's coupler object.



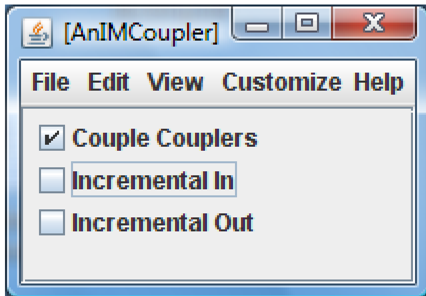


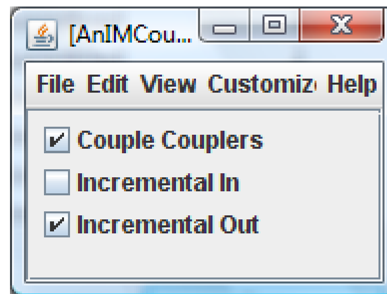
If Alice now changes IncrementalIn to true, this change is also sent to Bob's coupler object. However, it has no effect, as his IncrementalIn was already true.



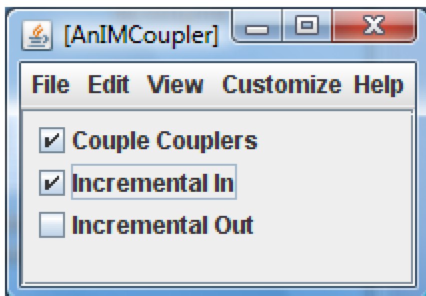


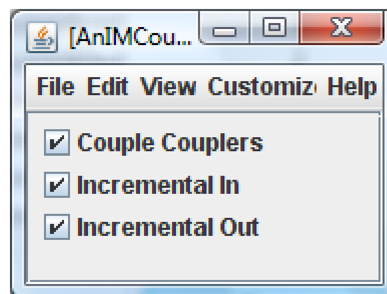
If Alice now sets IncrementalIn back to false, this change is again sent to Bob, and this time it does have an effect.



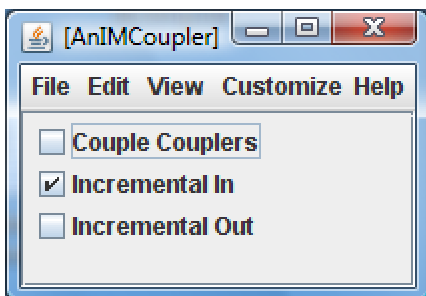


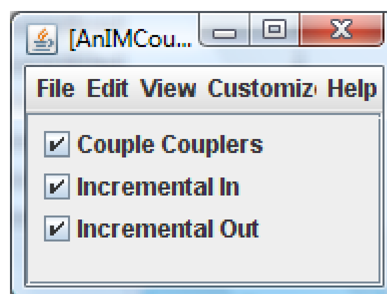
If Bob sets it back to true, Alice's property changes, so they can now have coupler wars.





Alice can withdraw from this war by setting CoupleCouplers to false. As the property is now false, this change, all subsequent changes to the properties of the coupler object, are not reflected in Bob's coupler.





Class-Independent Coupler

Make your coupler should be independent of the exact class of the IM object it is controlling. The class of the IM object should be passed to it as an argument to its (one and only) constructor. The coupler can, however, assume the name of the properties of the class it controls.

Extra Credit

To implement these features, you can make the model collaboration-aware. Also you may have to implement logical components that are processed by Sync and not ObjectEditor. See the UI section in the end on how to position and hide components. However, our code should not be distribution-aware – all network communication should be done by Sync.

1. Implement some game and share it in the IM.
2. Implement access control that prevents a user from editing a history message committed by some other user. Make this access control an option.
3. Support an invitation-based session management, wherein users are explicitly sent invitations, and acceptance of the invitation puts them in the session.
4. Allow users to undo their posting of messages to the history.
5. Allow users to share their status such as available and busy.
6. Allow each user to add a user-specific "robot" object that transforms incremental inputs of other users before showing them to him/her. Implement a robot object that makes all characters upper case.
7. Implement robot objects that also do the reverse translation. Use this feature to show and edit transformed messages in the displayed history.
8. Support (private and/or public) playback.

You can make the implementation of these extra credit features the basis of the final project.

Written Document

Write a document that:

- Has test cases, in the form of screen shots, illustrating the functionality you implemented. Be sure to explicitly indicate the extra credit features you implemented.
- Gives scenarios/applications in which it is useful to couple the couplers.
- Answers the following questions: (a) Explain, giving "realistic" scenarios, why it is preferable to have separate properties controlling whether incremental inputs are received and sent, respectively, rather than a single property to controlling both? (b) When the couplers are coupled, their properties do not immediately converge because of the current semantics and implementation of Sync.setReplicate(Class, property, val). Suppose we wanted to create a version of this call that ensures that the property has the same value in all associates. What value would you choose? How would you

implement this call? (c) Sketch how you could use Sync to build (the model of) a directory synchronization tool such as DropBox, Groove or LiveMesh. (d) Explain the differences between the Sync IM model and the N-user model of the previous assignment. (e) Your implementation should have used a factory. Explain which requirement forced you to use it.

Submission

Submit a printout of the document.

Put your code and documents in a subdirectory in the folder: \\dewan-cs.cs.unc.edu\deposit\790-063-Assignments\Assignment4

User Interface

As mentioned in the previous assignment, the general way to improve the UI is to set UI attributes of some property, P, of class C. These attributes must be set before you ask ObjectEditor to edit an object.

Here are three attributes that may be of interest in this assignment:

1. *READ_ONLY:* determines if the property is editable or readonly by the user. If this attribute is set to false, even if a setter exists for the property, it will not be editable by the user.
2. *VISIBLE:* determines if the property is visible to the user. If this attribute is set to false, even if a getter exists for the property, it will not be displayed to the user.
3. POSITION: determines the position of the property in the display.