Distributed Notification Framework using Java RMI

Argyris Zardilis
`az2g10@ecs.soton.ac.uk`

May 2, 2012

1 Distributed Notification Framework

1.1 Implementation and Design of the Classes

In this section an explanation of the static structure of the framework is given showing the classes along with the attributes capturing their states, the operations which they perform and how they are related. The NotificationSource is the object that has references to a number of interested NotificationSink objects. The NotificationSource object is the one reacting to an event by sending Notifications objects to all registered NotifcationSink objects containing information about the particular event. The event-driven programming paradigm is used with callback functions, a widely used approach for event-driven systems like GUI libraries(java swing,GTK+). The callback function is on the NotificationSink object with a Notification object as a formal parameter(Figure 1) . The Notification object acts as a wrapper for the event information so it contains the information in a class Object object so that it can be used with any kind of information(any kind of object) as all Java classes inhrit from Object class. It was also deeemd necessary for the Notification object to carry information about the sender of the Notification namely the NotificationSource that triggered the notify() callback function on the NotificationSink because one NotificationSink may subscribe to multiple NotificationSources therefore a way to distinguish notifications was needed . So the name of the NotificationSource that triggered the callback function is included in the Notifcation object. The NotificationSink is an abstract class with no implementation code inside acting as an interface that users of the framework may fill to capture their needs. Different applications would want to do something different upon receiving a notification, which might act as a wrapper for any type of object, and process that event coming in different ways. On the other hand the behaviour of the NotificationSource is more application independent because the operations are always the same. For example the fireEvent() method takes an Object containing the event details that need to be communicated to the clients and after wrapping them into a Notification object along with its name, which will act as an identifier for the source on the sink side, calls the notify() callback function on all the registered sinks which as previously mentioned are kept in a list. A pseudo-code for the above operation is :

```
Notification note = new Notifcation(Object info, name)
List<NotificationSink> registeredSinks
foreach sink in registerSinks:
        sink.notify(note)
```

The other operations supported by the NotifactionSource as can be seen in Figure1 are for removing and adding NotificationSinks to the list of interested sinks for receiving or stop receiving notifications. It is clear the those methods should be accessible from remote objects namely sinks that want to subscribe/unsubscribe to the particular source's notifications. Therefore the NotificationSource object is a remote object by extending the UnicastRemoteObject Java class. It is also obvious that NotificationSink objects are also remote by extending the UnicastRemoteObject Java class.

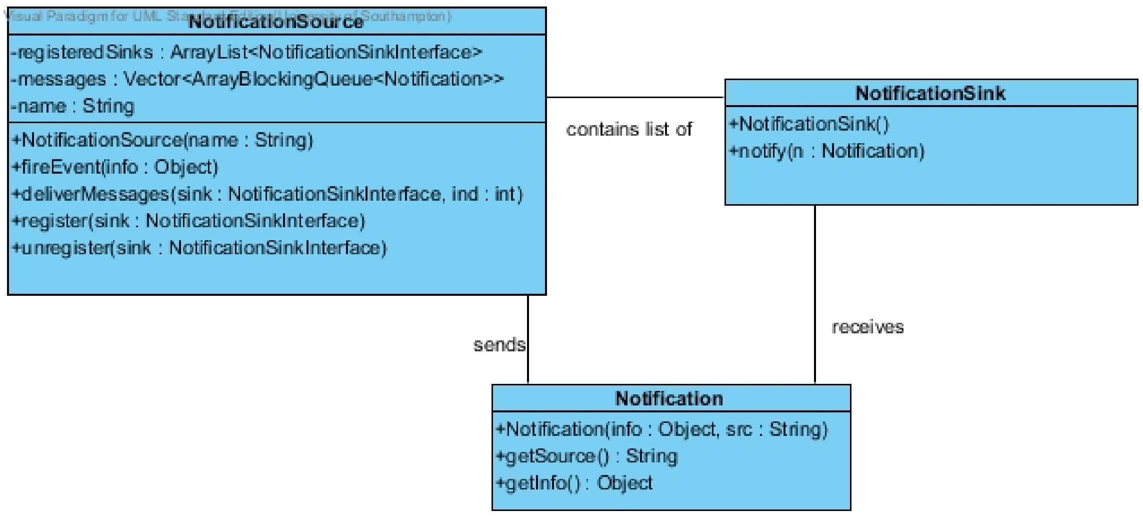
Visual Paradigm for UML Standard Edition(University of Southampton)
NotificationSource
-registeredSinks : ArrayList<NotificationSinkInterface>
-messages : Vector<ArrayBlockingQueue<Notification>>
-name : String
+NotificationSource(name : String)
+fireEvent(info : Object)
+deliverMessages(sink : NotificationSinkInterface, ind : int)
+register(sink : NotificationSinkInterface)
+unregister(sink : NotificationSinkInterface)
contains list of
NotificationSink
+NotificationSink()
+notify(n : Notification)
sends
receives
Notification
+Notification(info : Object, src : String)
+getSource() : String
+getInfo() : Object

Figure 1: Class diagram of the Distributed Notification Framework

1.2 Error handling

One of the main differences between distributed applications and normal applications is their need for specific error mechanisms because the distribution of resources among different machines can cause all sorts of unpredicted errors like network connection problems etc. So it was deemed necessary to provide a way for at least recovering from network errors because otherwise the framework would not be reliable enough. The part of the error mechanism provided by the framework lies on the NotificationSource side since it is the only class with implementation code in. So a solution was needed that short term problems in the connection to any of the sinks by the source would not cause any long term problems to the running of the application and even some recovery of lost Notifications for a short period of time. The scheme that was devised uses queues of messages. As it can be seen from Figure1 the NotifationSource objects contain a list of queues, a queue for each one of the registered notification sinks. As soon as the notifications is ready for sending to a sink A it is queued in the appropriate queue for sink A. Then the deliverMessages function is then called that attempts to deliver all the messages in As queue to A. The notification is removed from the head of the queue and tried to be sent to the sink. If the operations succeeds then the operation is repeated for the next notification, if there is one, or goes back to the fireEvent() function which continues to the next sink. If the notification fails (RemoteException is thrown) the Exception is caught and the last notification is put back in the queue. That way if the sink comes back to live after a few events it will receive all the notifications that it missed at once. However a limit has been put at the amoun of Notifications that can be queued by the Source. If the size of specific queue increases past the limit which is ten that means the for the last 10 notification events that sink could not be reached. In this case the Source decides that the particular sink has a long term problem and unregisters it so as to not cause long term problems or event hinder the performance and delivery of notifications to the other sinks.

2 A working application using the Distributed Notification Framework

The application designed using the Framework is an online board which can be used as teaching tool. The basic idea is that there is a Teacher application sitting somewhere and the user draws/writes to the board(a window). All the writings are received in real-time by the student applications on different machine. The idea was inspired by the online education website Khan Academy

2.1 Implementation

In order to implement the applications signal for specific operations like drawing to the screen, clearing the screen needed to pass from the source to the sink along with the location information. All the information is packed into a WriteSignal object which contains the type of the signal and the location.This WriteSignal object is the one that will move wrapped in a Notification object from source to the sink. The Teacher Application contains a NotificationSource object that handles the passing of the signal to the interested/registered sinks. The events that trigger the NotificationSource to notify the sinks are: write to the screen (with mouse dragging), relocating the mouse (with clicking to a different area of the screen), clearing the screen(clear Button at the bottom of the application) and change of the colour of drawing (colour button at the bottom of the panel. When either of those events happen the application packs into a WriteSignal object and calls the fireEvent() method on the NotificationSource which then wraps the WriteSignal object into a Notification object and calls the notify() method on all the registered sinks passing them the Notification object. This implementation clearly supports multiple sinks since all of them are kept in a registry(List) and the callback function is called on all of them when the specific event occurs. At the other side the Student Application has a Student object which is an implementation of the NotificationSink(sinks need to be implemented since they contain no implementation code). Two drawing panels are used on the student application separating the main panel into 2 tabs(see Appendix) in order to demonstrate the use of multiple sources for the same sink object. The Student objects implements the notify() method of the parent NotificationSink class. When it received a Notifcation it checks the source (name of the source of the message) and then makes a dispatch based on that name sending the WriteSignal object contained in the Notification to the appropriate panel that has subscribed to that source. Then the panel interprets the signal and acts accordingly. The student application can also register to receive notification from a particular server by clicking the register button at the bottom of the panel. Clicking the

2.2 Error handling

For reasons previously mentioned the NotificationSink class of the Framework does not contain any code and therefore it was extended by the Student class in the application to handle the notifications. An error mechanims was needed for the sink side of view as well in case the register operations could not find the specified source. The way that this possible problem was dealt with was more primitive that the error handling in the NotficationSource. The Student application tries for a maximum number of 10 times to connect and get reference to the source object from the remote locations. If the number of maximum tries is reached the applications stops trying so as to not cause a long term problem in the running or responsiveness of the application.

2.3 Running on different machines

The application was created and tested on a single machine. But in order to leave this testing stage and move into a more real-world environment where it belongs since it is Distributed applications(by definition runs on more than one machine) some changes to the code need to be done. In order to run and

RMI application the supporting class files must be placed in locations where they can be found by both the server and the client. Such files include the remote service interface definitions the stubs and skeletons for the implementation classes and of course the class definitions for the applications itself. This distribution of classes is handled automatically be the RMIClassLoader. The ideas is to use FTP or HTTP servers to fetch the classes that are needed from either side. The way that this is done is thorugh setting some properties in the JVM when it is started. The most important one being the name of the server and location to look for the implemening classes.Another thing needed by Java when the remote object's stub resides does not reside on the client side is to install a security manager with System.setSecurityManager(new RMISecurityManager). WIth this manager you can specify permissions on a variety of aspects(file/socket permissions) and the main idea is that you don't download code you are not sure of.

3 Conclusions and Critical Evaluation