COMP2207 - Distributed Systems and Networks

A DISTRIBUTED NOTIFICATION FRAMEWORK USING JAVA RMI

December 15, 2016

Vedant Chokshi
Student ID: 27748456
University of Southampton

1 THE NOTIFICATION FRAMEWORK

This section explains how the classes for the Notification framework have been designed and implemented. This Notification framework requires for the the sink and source to both communicate with each other; sink when subscribing/unsubscribing to a source and the source when sending notifications to the subscribed sinks. Hence, this would require the source and sink remotely invoke methods in one another. As a result, I decided to implement two interfaces; NotificationSourceInterface and NotificationSinkInterface. Both classes extend java.rmi.Remote and declare a set of remote methods. All methods throw a Remote Exception as they are thrown by the underlying RMI implementation.

The Notification framework requires a Notification to be sent from the source to the sink. Therefore, I created a Notification class which stores the source , date and the message of the notification. The Notification class extends Serializable as instances of Notification are sent over the network when using RMI. It involves the conversion of a Notification instance to a series of bytes, so that it can be streamed across the communication link. The byte stream can then be deserialized and converted into a replica of the original Notification instance sent by the source.

Next I defined the objects distributed by NotificationSourceInterface and NotificationSinkInterface. The class NotificationSource implements NotificationSourceInterface and the class NotificationSink implements NotificationSinkInterface. Both classes also extend UnicastRemoteObject. This means that every instance of this NotificationSource and NotificationSink will be unique and so messages to it can be directed to this single copy.

The framework works so when a source is created (instance of NotificationSource), it is binded to the registry so it can be remotely accessed (by calling the bind method in NotificationSource). Then when a sink (instance of NotificationSink) wants to subscribe to a source to receive notifications, the sink looks up the source it wishes to subscribe to in the registry to try to obtain a reference to the source (by using LocateRegistry.getRegistry("localhost", 9999).lookUp(sourceName)). If found, the sink calls the method registerSink which remotely invokes the registerSink method (in the source) whilst passing itself in as the argument. This results in the sink being added to a HashMap with the User ID as the **key** and the sink as the **value** (decision made to handle reconnections). As the sink is now stored in the source, the source can also now remotely invoke methods in the sink. This is done when the source sends instances of Notification to the subscribed sinks. The source sends a Notification by calling the broadcastNotification method in the source which takes in a Notification instance as an argument. The method then iterates over all the sinks stored in the HashMap which contains all the sinks subscribed to the source and remotely invokes the retrieveNotification in each sink and sends the Notification to the sinks.

2 The Application

The application I built on top of my Notification framework is a Traffic Alerts application. The user subscribes to receive Traffic Alerts from the available areas and receives alerts from the Areas the user has subscribed to. To design my application on top of the Notification framework, I first constructed a System Diagram which outlines the basic structure of the application and framework. I decided to have a SinkGUI and a SourceGUI. The SinkGUI requests to subscribe to a source. The NotificationSink takes in the request from the SinkGUI and connects to the corresponding NotificationSource. The input from the SourceGUI is passed onto the NotificationSource which is sent to the NotificationSink as a Notification. The NotificationSink then calls methods to update the SinkGUI to display the Notification.

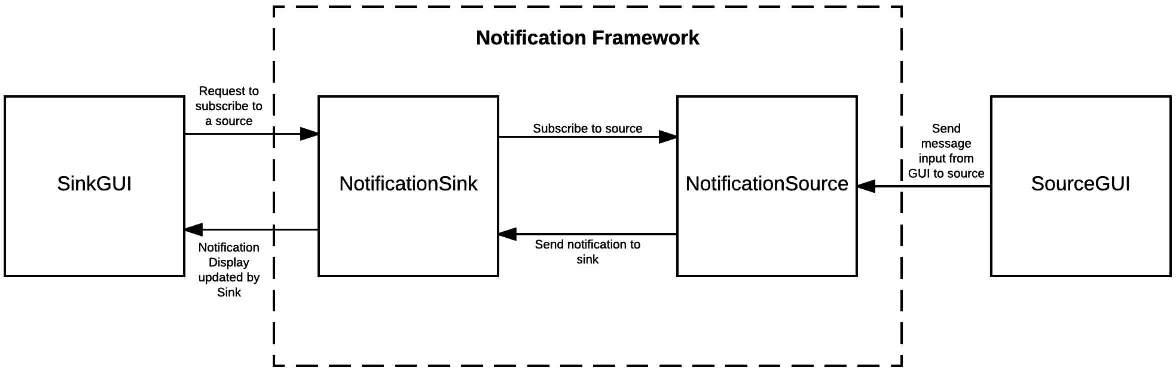

Notification Framework
SinkGUI
Request to subscribe to a source
Notification Display updated by Sink
NotificationSink
Subscribe to source
Send notification to sink
NotificationSource
Send message input from GUI to source
SourceGUI

Figure 1: System Diagram

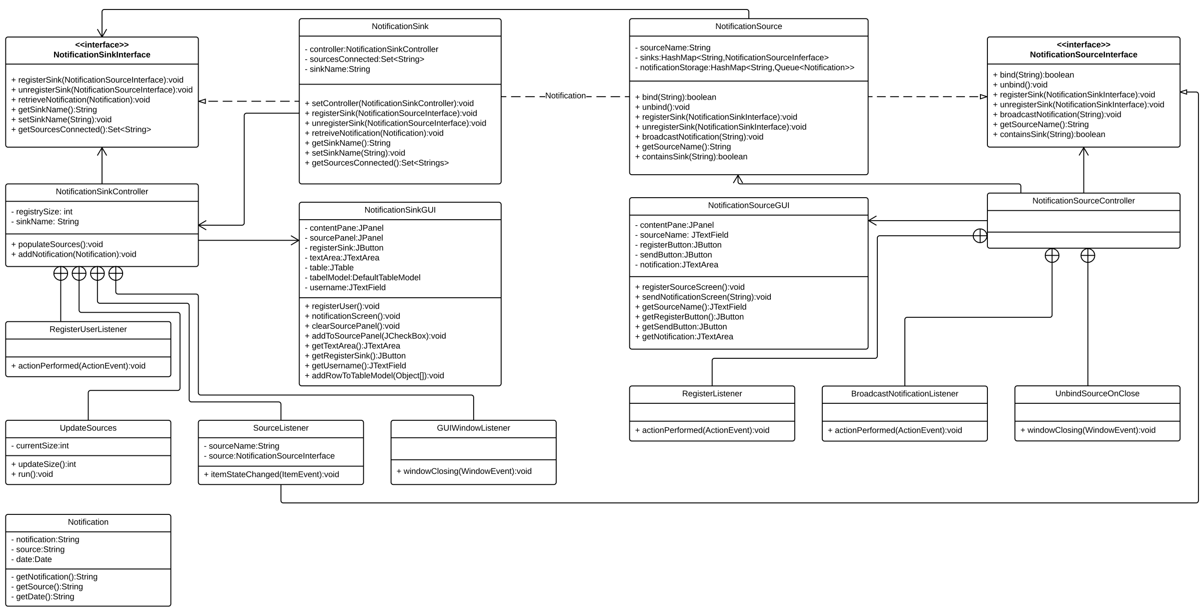

<<interface>>
NotificationSinkInterface
+ registerSink(NotificationSourceInterface):void
+ unregisterSink(NotificationSourceInterface):void
+ retrieveNotification(Notification):void
+ getSinkName():String
+ setSinkName(String):void
+ getSourcesConnected():Set<String>
NotificationSink
- controller:NotificationSinkController
- sourcesConnected:Set<String>
- sinkName:String
+ setController(NotificationSinkController):void
+ registerSink(NotificationSourceInterface):void
+ unregisterSink(NotificationSourceInterface):void
+ retreiveNotification(Notification):void
+ getSinkName():String
+ setSinkName(String):void
+ getSourcesConnected():Set<Strings>
Notification
NotificationSource
- sourceName:String
- sinks:HashMap<String,NotificationSourceInferface>
- notificationStorage:HashMap<String,Queue<Notification>>
+ bind(String):boolean
+ unbind():void
+ registerSink(NotificationSinkInterface):void
+ unregisterSink(NotificationSinkInterface):void
+ broadcastNotification(String):void
+ getSourceName():String
+ containsSink(String):boolean
<<interface>>
NotificationSourceInterface
+ bind(String):boolean
+ unbind():void
+ registerSink(NotificationSinkInterface):void
+ unregisterSink(NotificationSinkInterface):void
+ broadcastNotification(String):void
+ getSourceName():String
+ containsSink(String):boolean
NotificationSinkController
- registrySize: int
- sinkName: String
+ populateSources():void
+ addNotification(Notification):void
NotificationSinkGUI
- contentPane:JPanel
- sourcePanel:JPanel
- registerSink:JButton
- textArea:JTextArea
- table:JTable
- tabelModel:DefaultTableModel
- username:JTextField
+ registerUser():void
+ notificationScreen():void
+ clearSourcePanel():void
+ addToSourcePanel(JCheckBox):void
+ getTextArea():JTextArea
+ getRegisterSink():JButton
+ getUsername():JTextField
+ addRowToTableModel(Object[]):void
NotificationSourceGUI
- contentPane:JPanel
- sourceName: JTextField
- registerButton:JButton
- sendButton:JButton
- notification:JTextArea
+ registerSourceScreen():void
+ sendNotificationScreen(String):void
+ getSourceName():JTextField
+ getRegisterButton():JButton
+ getSendButton:JButton
+ getNotification:JTextArea
NotificationSourceController
RegisterUserListener
+ actionPerformed(ActionEvent):void
RegisterListener
+ actionPerformed(ActionEvent):void
BroadcastNotificationListener
+ actionPerformed(ActionEvent):void
UnbindSourceOnClose
+ windowClosing(WindowEvent):void
UpdateSources
- currentSize:int
+ updateSize():int
+ run():void
SourceListener
- sourceName:String
- source:NotificationSourceInterface
+ itemStateChanged(ItemEvent):void
GUIWindowListener
+ windowClosing(WindowEvent):void
Notification
- notification:String
- source:String
- date:Date
- getNotification():String
- getSource():String
- getDate():String

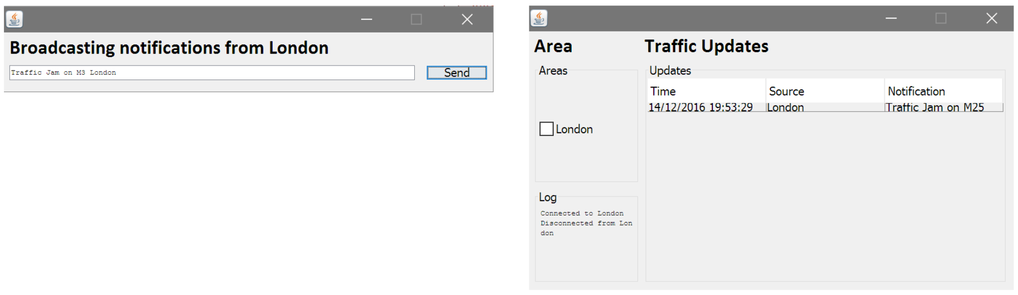
Figure 2: Class Diagram

Then I constructed a Class Diagram (Figure 2) which implements the System Diagram in more detail to show how

the application is implemented on top of the framework and summarises the whole implementation. In short, there are 2 controllers, NotificationSinkController and NotificationSourceController. NotificationSinkController acts like the medium between the NotificationSink and NotificationSinkGUI and NotificationSourceController does the same for between NotificationSource and NotificationSourceGUI (replicating the functionality of the arrows in the System Diagram).
To start the application:

1. Compile all java files using javac *.java in the directory which contains all the files
2. After compilation, first run RegistryRun.java by using java RegistryRun
3. Next, run java NotificationSourceMain (as many times) to create an area to broadcast from. Broadcast alerts by writing an alert and sending them.
4. Lastly, run java NotificationSinkMain (as many times) to create sinks which receive traffic alerts according to which source(s) the sink subscribes to.

Next I carried out some tests to show evidence that my application works. Figure 3 shows the test where I firstly subscribed (seen in log) to the area London and sent the alert "Traffic Jam on M25" which displays the alert on the GUI. Next, I unsubscribed from the area London (seen in long) and sent the alert "Traffic Jam on M3" which does not show up on the GUI as I have unsubscribed from receiving traffic alerts from London. Hence, this test clearly shows that my subscribe and unsubscribe functionality works.



Broadcasting notifications from London

Traffic Jam on M3 London

Send

Area

Areas

☐ London

Log

```
Connected to London
Disconnected from Lon
don
```

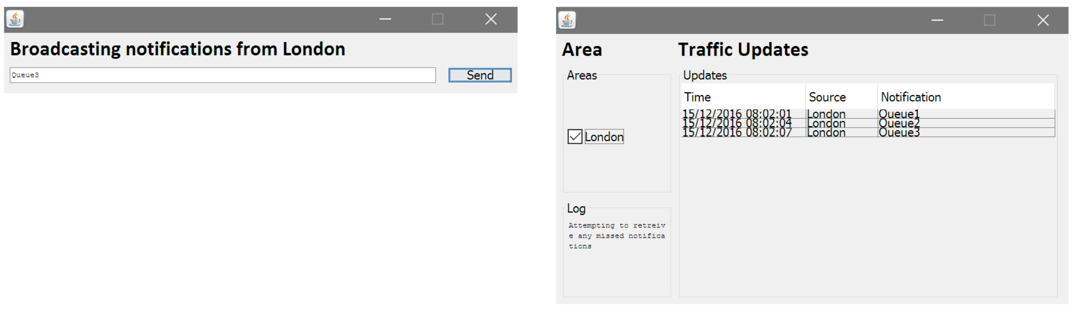
Traffic Updates

Updates

Time	Source	Notification
14/12/2016 19:53:29	London	Traffic Jam on M25

Figure 3: Test to check subscribe/unsubscribe

Figure 4 shows the test where I firstly subscribed to the area London and then closed the Sink GUI. I then sent 3 messages "Queue1", "Queue2" and "Queue3" to test if these missed notifications are retrieved when logging in again with the same sink name/user ID. The Sink GUI log shows that the missed notifications are retrieved and the messages are displayed on the GUI. Hence, this test clearly shows that my reconnection functionality works.



Broadcasting notifications from London

Queue3

Send

Area

Areas

☑ London

Log

```
Attempting to retreiv
e any missed notifica
tions
```

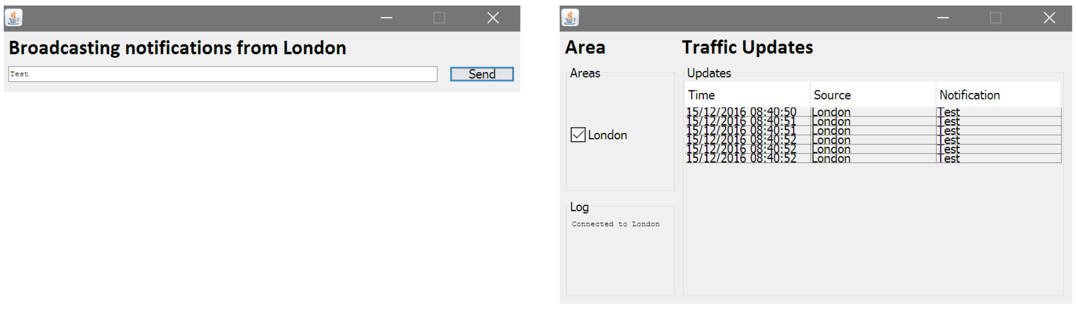
Traffic Updates

Updates

Time	Source	Notification
15/12/2016 08:02:01	London	Queue1
15/12/2016 08:02:04	London	Queue2
15/12/2016 08:02:07	London	Queue3

Figure 4: Test message queueing when sink is disconnected

Figure 5 shows the test where I attempted to send the same message ("Test") multiple times to test that there are no other errors. As the GUI shows, each message was delivered successfully; showing that the subscription and notification sending functionality works


Broadcasting notifications from London

Test — Send

Area — Traffic Updates

Areas: ☑ London

Updates

Time	Source	Notification
15/12/2016 08:40:50	London	Test
15/12/2016 08:40:51	London	Test
15/12/2016 08:40:51	London	Test
15/12/2016 08:40:52	London	Test
15/12/2016 08:40:52	London	Test
15/12/2016 08:40:52	London	Test

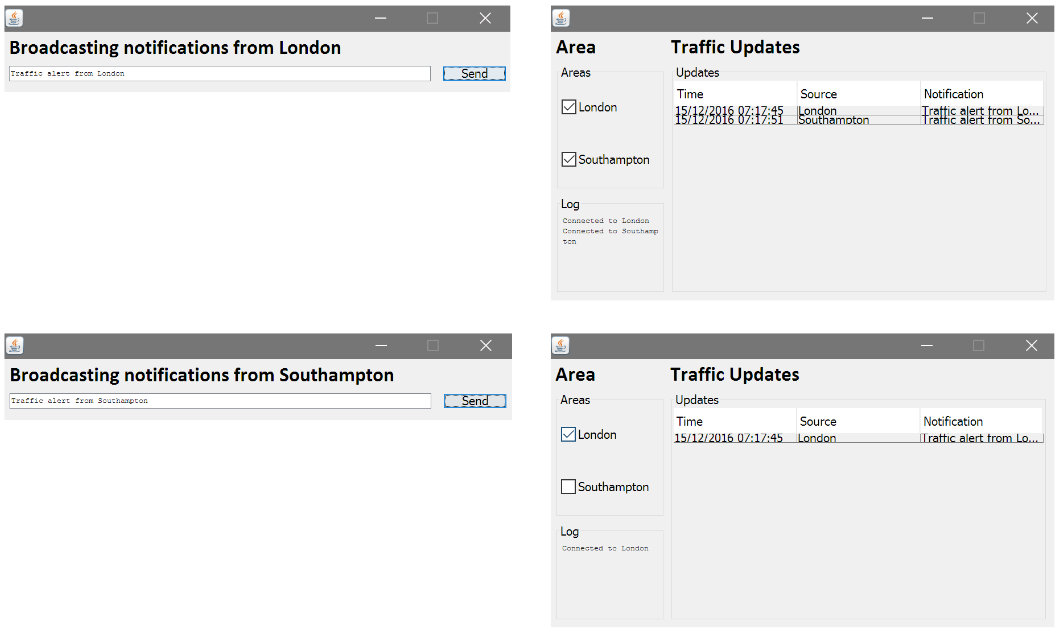
Log

```
Connected to London
```

Figure 5: Test to check multiple Notifications being sent

3 Multiple Sources and Sinks

The application allows for multiple NotificationSources and NotificationSinks to connect to each other. So for my application, it allows multiple users (sinks) to subscribe to multiple areas (sources) to receive traffic alerts. The sources and sinks are distributed as they are different processes on the same computer, but they work together by sharing data, message passing and and remotely invoking methods. Figure 6 shows evidence of multiple sources and sinks as it shows a test where each source only sends alerts to the sink(s) which has subscribed to it in an environment when multiple sources and sinks are present.


Broadcasting notifications from London

Traffic alert from London — Send

Area — Traffic Updates

Areas: ☑ London ☑ Southampton

Updates

Time	Source	Notification
15/12/2016 07:17:45	London	Traffic alert from Lo...
15/12/2016 07:17:51	Southampton	Traffic alert from So...

Log

```
Connected to London
Connected to Southamp
ton
```

Broadcasting notifications from Southampton

Traffic alert from Southampton — Send

Area — Traffic Updates

Areas: ☑ London ☐ Southampton

Updates

Time	Source	Notification
15/12/2016 07:17:45	London	Traffic alert from Lo...

Log

```
Connected to London
```

Figure 6: Multiple Sources and Sinks

4 LOST CONNECTIONS

To handle lost connections I implemented the functionality of user ID/sink name which gives each instance of NotificationSink a unique name so when registering with a user ID/sink name that has been registered before, reconnection takes place. I did this by adding a Set<String> which contains all the names of the sources the sink is connected to. The Set<String> is then outputted to file which is named after the user ID/sink name so the files are unique as well. When registering with the user ID/sink name, if file for the user ID/sink name exists, then the file is read which returns a Set<String> containing names of all the sources connected to the sink before the loss of connection. The Set is iterated over and the the sink is re-registered with all the sources connected (if they exist) and the instances of Notification in the queue for the sink that has reconnected are re-delivered.

In NotificationSource, I decided to use a HashMap<String, NotificationSinkInterface> for storing the all the sinks connected to the source and I decided to use a HashMap<String, Queue<Notification>> to queue up the Notifications for sinks that have been disconnected. The **key** in both HashMaps store the unique sink name/user ID which can be used as a reference reconnect a new instance of NotificationSinkInterface and queue instances of Notification for sinks that cannot receive the Notification. The queuing is done by a try-catch statement in the broadcastNotification method in the NotificationSource. This method iterates over the HashMap<String,NotificationSinkInterface> which contains all the sinks connected to the source and remotely invokes the retrieveNotification method in each sink. However, if a sink has lost connection, trying to remotely invoke the retrieveNotification method will throw a RemoteException. In this case, the RemoteException is caught and the Notification is instead added to the corresponding queue to the sink name in the HashMap<String,Queue<Notification>>.

5 FUTURE WORK

To run the application on multiple machines, all that needs to be done is replace the "localhost" in the NotificationSource to a bind location in the network like - "//" + hostName + ":" + port + "/sourceName". To remotely access a NotificationSource via the NotificationSink change the the lookUp("localhost", 9999) in the NotificationSinkController to the location of the NotificationSource in the network like - lookUp("//" + remoteHostName + ":" + remotePort + "/sourceName"), remotePort). The port and remotePort should have the same port as the port the RegistryRun creates the Registry on (currently 9999).

One limitation of my framework and application is the way reconnection is handled. The files for user IDs/sink names are stored locally so when logging in from another machine, the file will not be available and so there is no list of the sources that the user was previously connected to and hence the ability to automatically reconnect to the source is lost. One way to eliminate with this issue would be to save the files save the files in one central storage location in the network which stores all the files instead of saving the files locally.

Another limitation of my framework and application is the way the NotificationSource stores Notifications when a sink is disconnected. The source stores the Notifications that are not delivered in a HashMap<String, Queue<Notification>> which are re-delivered when the sink connects to the source again. However, if the source crashes and shuts down, the stored Notifications are lost and when the sink reconnects, the Notifications are not re-delivered as they have been lost. This could be eliminated by having a file store which stores all the data for the sources when closed and loads it back up when a source with same area name is created (similar to reconnection from sink to source) .

6 CONCLUSIONS

There are many advantages to using a distributed objects architecture for implementing my framework and application. Using a distributed objects for the framework and application means that information can be shared between many applications with ease. The separation of the Notification framework and application means that there is a very open system architecture that allows new functionality to be added easily. This type of architecture also makes the the system very flexible and scalable.

However, there are also some disadvantages and limitations to using distributed objects for the framework and application. One disadvantage of using distributed objects is that their performance is inferior compared to other message passing technologies. Another disadvantage of my implementation is that the distributed objects are centred around single machine (localhost). This would mean that the health of the single machine is the health of the entire system and so if the machine was to shut down, so would the entire system. To overcome this limitation, my implementation can be slightly changed as mentioned in Section 5 to run distributed objects over a network so damage to one machine would not affect the entire system.

To conclude, I believe that there are many advantages and disadvantages and overall distributed objects is a great way to implement an application over a network.