COMP2207 Java RMI Coursework

Hannah Short
has1g15@soton.ac.uk

This assignment explores the use of Java Remote Method Invocation to design and implement a distributed notification framework. Java RMI is a Java application programming interface serving as a platform for distributed application creation. It simulates the behaviour of Remote Procedure Calls but instead makes use of object orientation. Objects are able to interact with one another in a distributed network, this is what makes RMI different to RPC, objects are able to be passed with the call. RMI enables the methods of a remote object to be invoked from another virtual machine.

The Notification Framework

Initially, I created all the relevant classes as listed in the specification. I had main methods executing in both the source and sink classes, they both called constructors in the respective classes. Then, I went about implementing an interface for the framework, named Notifiable, which inherits the behaviour of the Remote interface from the Java RMI library. This interface allows its methods to be invoked from a remote JVM. Notifiable contains a remote receiveNotification method which is passed a Notification object and throws a Remote Exception in the case that there is a communication exception during the execution of the remote method call.

The NotificationSink class implements the Notifiable interface, and therefore also its method, receiveNotification, enabling it to be passed notifications. When run, the sink creates a registry. A server in RMI uses the registry, a naming service, to map remote objects to names. The default registry port number is being used here. A new sink object is constructed with the name to be mapped to the object passed as a parameter.

The NotificationSource class stores a list of registered sinks as well as the registry for looking up objects. It stores an instance of registry by calling getRegistry() on the LocateRegistry object. A stub is then retrieved by calling lookup on this registry with a constant name passed in as an argument. It contains a notify method to pass the notification to all the registered sinks. Further to this, it implements a sendNotification method which takes in a message and calls the receiveNotification method in the remote interface on the stub object, passing in a new instance of Notification with the message parameter being stored in notification.

The Notification class stores the notification message as well as the instant at which the notification is received by the notification sink and returns this information to the receiveNotification method in sink when requested. This class implements the Serializable interface because the object is storing data of which will be serialised in order that it can be sent between different virtual machines, while maintaining persistent storage as well as reducing communication error during the stream.

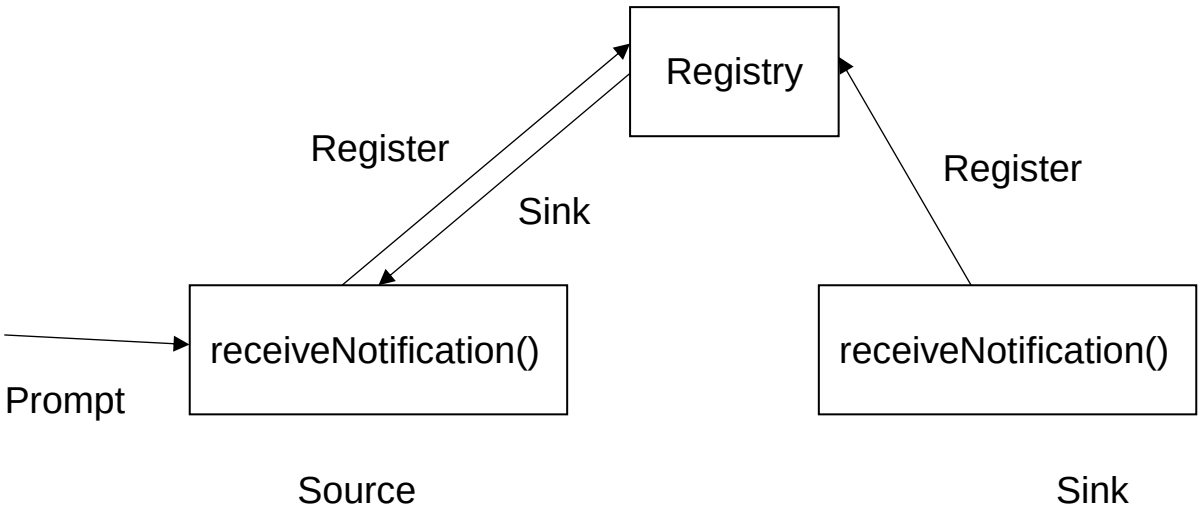


Figure 1: A simple diagram showing the functionality of the RMI framework

The Application

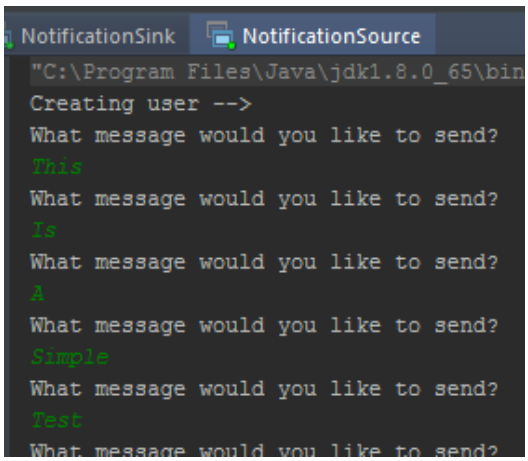
I chose to imitate a basic instant messaging application in order to make use of the framework, I made this decision because I thought that it would be one of the most transparent means of showing the performance of the framework. First of all, I created two classes, UserInputSource and ReceiptSink which inherited the behaviours from both the respective source and sink classes as well as overriding methods and adding extra functionality such as displaying output to the user as well as passing the notification message to be sent.

In UserInputSource, a class extending NotificationSource, a separate class entitled UserInput is instantiated which retrieves the notification message as input from the user. The object is added to a list of UserInputs so the messages passed are all stored. The message is given as an argument to the sendNotification method in order that it can call the receive method on stub.

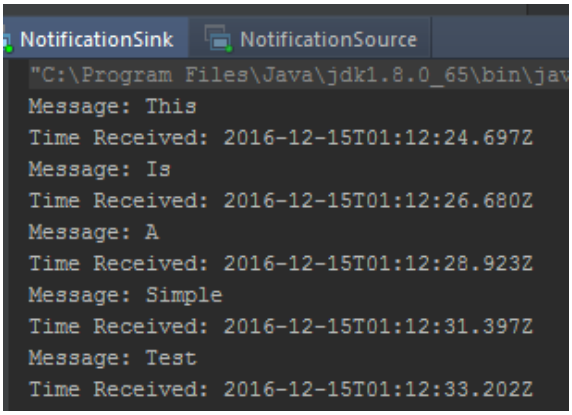
ReceiptSink is an application class which acts as a child class to NotificationSink, I didn't manage to complete a full implementation of this class but intended it to deal with the message and time outputting, which at present is dealt with by the framework itself.

The application is executed by first running the sink class which then waits for a response from the source class. Then the source class can be run which loads the application class inheriting from the source and prompts the user to enter a notification message after giving the impression of a user being created. The user enters their message at which point the remote method in the interface is invoked in order that it can be received by the sink.

Notification Framework Testing:



These screenshots show an example of both the source and sink being run simultaneously and the application execution.
The UserInputSource class in the application prompts the user to type a message until there is an exception so the application runs consistently.



By use of the RMI registry, the sink is able to output the messages on the notification sent by the user from the source and provide the instant at which they were sent at.

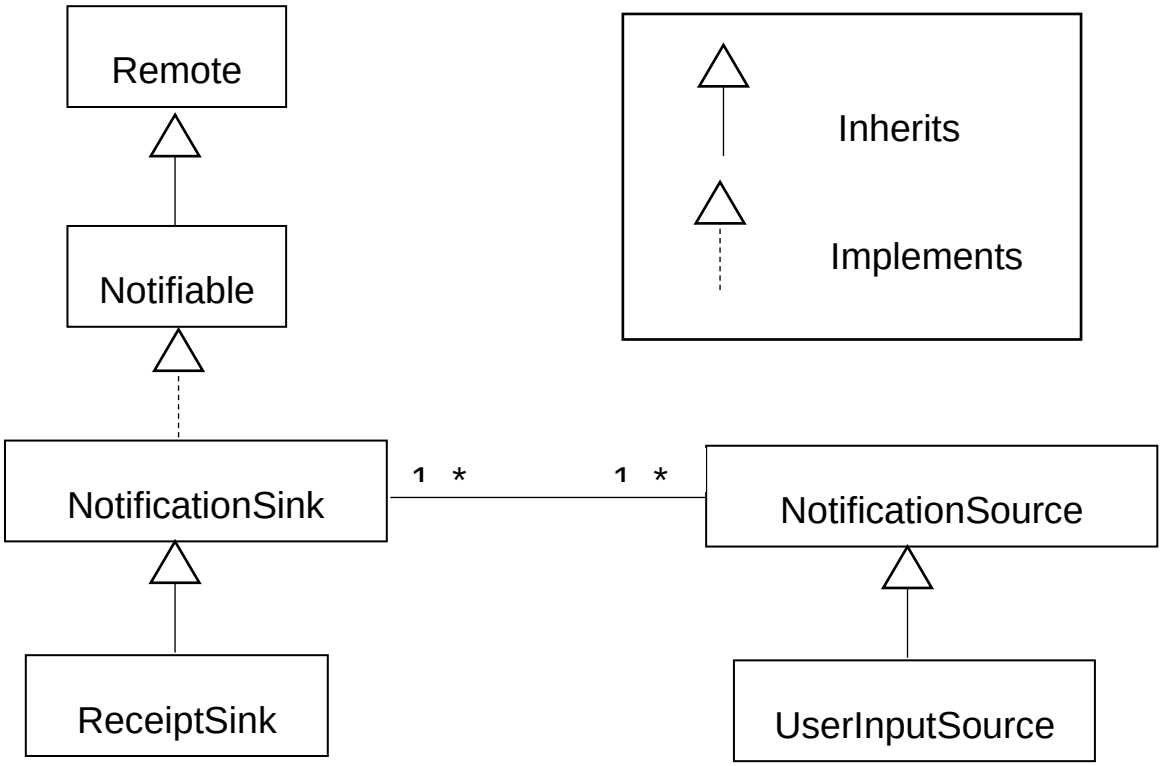


Figure 2: Simple class diagram representing RMI application

Multiple Sources and Sinks

Due to timing, I was unable to fully implement the use of multiple sources and sinks in the application. My registerSink method is used initially when the application loads and the first sink constructed is added to the list of registered sinks. I had the intention of constructing a new source every time a notification was sent from a new user in the application and adding a new sink to the list in the case where there are multiple users wanting to use the application and they require their own sinks.

Lost Connections

The first precaution that my application takes to deal with lost connections is ensuring it stores all the messages entered by the user in the form of an array list. It adds a new instance of the UserInput class every time a user submits a response. If the connection is lost, the source is able to access the last element stored in the list of inputs and reregister it.

Another means of dealing with the connection loss is the exceptions thrown by the methods in the application. In particular, as an example, the RemoteException is thrown if the remote method in the interface fails to be invoked by the program so the application. I have used a try-catch statement within the UserInputSource class, the application will continuously prompt the user for input until the case where connection is lost at which point an exception is caught.

In terms of reconnection, after connection has been lost, the source will call upon lookup again in an attempt to link back with the server via the interface. After reflecting on the application, I believe I could have implemented a more graceful manner of dealing with a lost connection. For instance, after detecting the termination of the server, giving a message to the user informing them of what has happened and that the application will aim to have them reconnected shortly.

Future Work

In order to develop and expand the application and framework further in the future, the application could be run on multiple machines as opposed to a single machine running more than one virtual machine. To do this, the IP addresses of the machines must all be accessible so for instance not being blocked by a firewall, this is because the application will no longer be connecting to just a local address running on a single machine.

The complexity of the framework could be increased to deal with remote objects more effectively. For example: using a more suitable naming system in the registry as opposed to using constant names would be more advisable; implementing a more

viable means of sink registration as opposed to just a list. There would be potential for sinks to be unregistered and only certain sinks to be notified with a given notification.

The application itself could be greatly improved, there is the possibility of making multiple individual conversations with multiple sources and sinks and showing multiple conversations simultaneously. Furthermore, from a user perspective, the application could make use of a graphical user interface with components to enable the navigation of stored conversations although this wasn't a necessary task in this particular assignment.

Conclusions

Overall, I feel this assignment has allowed the Java RMI system to aid my understanding of the distributed objects model. By using remote objects and interfaces, I have been able to create a functional notification framework that is used by a simple application.

Had I had managed my time for the assignment much more strategically, I would have been able to implement some of the features, mentioned in the future works section such as creating a more viable means of handling multiple sources and sinks within the application as well as making the application itself much more complex with additional features.