University of Southampton

COMP2207: Distributed Systems and Networks

Distributed Notification Framework using Java RMI

Huw Jones
27618153
hcbj1g15@soton.ac.uk

December 14, 2016

1 Notification Framework

As the notification system involved sinks, sources and notifications, it made sense to have 3 base classes: `NotificationSink`, `NotificationSource`, and `Notification`. Due to RMI requiring interfaces to produce remote stubs, naturally, I extracted the publicly accessible remote methods into the respective interfaces (`INotificationSource` and `INotificationSink`). In addition, I made sure that both `NotificationSink`, `NotificationSource` extend `UnicastRemoteObject` to allow the creation of the remote stubs without any extra code. This means that NotificationSinks/Sources can be used directly in remote methods and the remote method will receive the remote stub - not a copy of the object.

1.1 Interfaces

The coursework set out a subscribe-publish system, therefore I decided that the following methods must be present in the interfaces for the frame to properly function. In `INotificationSource`:

1. `UUID register(INotificationSink)` - Registers a sink and returns the assigned UUID of the sink
2. `boolean register(UUID, INotificationSink)` - Registers a sink with the provided UUID, returns true if the sink was registered, (and vice versa)
3. `boolean isRegistered(INotificationSink)` - Returns whether or not the sink is registered
4. `boolean unregister(UUID)` - Unregisters the sink by UUID and returns true/false if the sink was unregistered or not
5. `boolean unregister(INotificationSink)` - Unregisters the sink and returns true/false if the sink was unregistered or not

and in **INotificationSink**:

1. `void notify(Notification)` - Notifies the Sink of a new Notification.

To supplement the interfaces, I created two helper classes: `NotificationSink` and `NotificationSource`. These classes implemented useful methods that I believed all sinks/sources required. Both classes had a `ShutdownHandler` thread that was attached to the JVM shutdown hook and allowed the sink/source to nicely shut down if it was terminated (`^C`, `SIG_TERM`, etc).

1.2 Notification

My `Notification` class has several properties: SourceID, Time, Priority and Data. `Notification` is a generic class that allows the data to be anything that implements `Serializable` - this ensure that the code that uses the framework will not compile if objects that are **not** serialisable are sent in Notifications. The SourceID field allows sinks to use callback functions to handle notifications from different sources, and the time field is a timestamp from when the server sent the notification. Although `Notification` features a priority field, I have not made any code that actively uses this.

1.3 Notification Source

`NotificationSource` has several methods to abstract common features from having to be implemented multiple times. Such features include sink registry management, RMI registry binding, sending notifications, and session resumption. Instead of each source manually binding to the registry, there are methods to bind to a specified registry. In combination with the sink registry management and sending notifications, there is a single method to send (publish) a notification to all registered sinks. In this same abstracted methodology, the code to handle session resumption and manage failed notifications is also tied in here. The end result is the Source calls `sendNotification(Notification)` to publish a notification and the underlying `NotificationSource` abstract class handles the actual message handling and delivery.

1.4 Notification Sink

`NotificationSink` has methods to abstract connecting/disconnecting from sources (gracefully) and handling notifications. When a sink registers with a source, the framework provides the ability for a callback function for that specific source to be attached. This allows the Sink to be able to handle the notifications from different sources effectively. For example, a Sink connected to a CCTV camera system Source would either want to display the image, or save it. But, a Sink registered to a text based weather system Source would not be able to display this data as an image. My framework allows one sink to register to both these sources and handle the data for each source as it should be handled; e.g.: display the CCTV image, and parsing the weather data to display it on a matrix display (for example).

2 Application

For the application, I decided to simulate a CCTV camera system. Each camera is a Source, and each client is a sink. As per any "decent" CCTV system, one client needs to be able to connect to all the cameras, and multiple clients (security office, hard drive to save footage, front of store) need to access different sources. I believe this application model to be able to demonstrate that my framework meets the specification.

However, due to complexities of setting up cameras/webcams and getting it to work as the application model is intended, I decided to simulate a camera source with GIFs - my Java source zip contains a selection of cat gifs and config files to simulate the camera.

To model the application, I had 2 classes: 1. Client 2. GifStreamer.

2.1 Client

The client provides a GUI to connect to an RMI server, and then sources. When a source connects, it opens a new window to view that source. If that window is closed, the client uses the underlying Sink layer to gracefully disconnect from the source. The client features a combobox to select registered sources if the optional framework component `SourceProxy` is running, or a textbox if the optional component is not running (see Section 5.1).

The client also features a config file system that allows the Client UUID to be stored, as well as automatically connecting to sources when the application runs. The autoconnect feature would be useful for systems where you would like the client to autoconnect to the cameras.

2.2 Source

The GifStream source loads a GIF file, extracts each frame, convert the frame to a byte array (`BufferedImages` are not serialisable). When it has converted the GIF, it then tries to register itself with the `SourceProxy` (if optional component is running), or binds directly to the RMI registry (if localhost), else it fails to start up. The GIF file is specified by a config file. When starting a source, config files are specified by using a `-c [PATH]` flag, or if the flag is not specified, the config loader tries to load `server.conf`. Config files specify the RMI server to connect to (if not localhost:1099) as well as the SourceID (that the client sees).

2.3 Testing

To test the framework, I found 3 cat gifs and created config files to stream those gifs. Next, I started 2 clients and connected to the registry server. I then connected client 1 to source 1 & 2, and client 2 to source 2 & 3.

Figure 1 is a screenshot whilst testing my application. The consoles for the 3 sources are on the left hand-side and both clients are on the right with 4 windows (2 for each client, 1 for each source). This screenshot proves that multiple sinks can register with multiple sources and one sink can register with multiple sources.

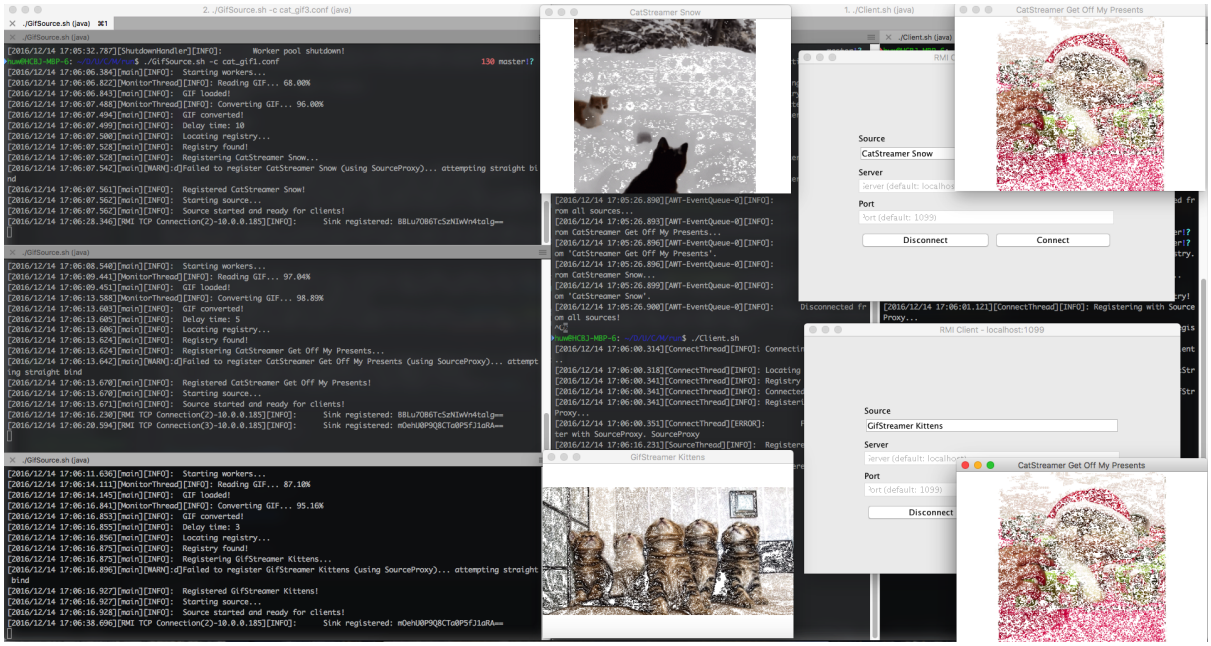

Figure 1: Screenshot of testing.

3 Sources & Sinks

4 Lost Connection

5 Future Work

5.1 Source Proxy

6 Conclusions