Distributed Systems and Networks  
**RMI Notifications Report**

horizontal line

# multiservermulticlient.png

# Notification Framework

### Design/Implementation

The notification message framework has been implemented via a set of interfaces, NotificationSource and NotificationSink, and a notification class, which all implement remote.

The NotificationSource provides a registerClient method that adds a NotificationSink object to an array of clients. The NotificationSink provides the method for performing a client action, which is remotely invoked by the server. The notification object contains an event object of generic type, a timestamp field, and a reference to the NotificationSource that sent it.

I designed the NotificationSource/Sink as interfaces because different applications may want to utilise Notifications in a different way, such as wanting to store a registry of clients differently. Interfaces allow this freedom while also making sure that implementation classes follow a consistent structures, as all implementations will need methods for registering clients, and invoking methods on clients.

### Testing

These interfaces are then implemented in concrete by the individual application that uses the framework, as I have done with my program, where I have built a relatively simple server-client game that tests the notification framework built.

# Framework Testing Application

### Design/Implementation

To test my framework, I had to design a concrete implementation of the classes created, along with a design for what functionality I wanted the server to remotely invoke on the client. For this purpose, I designed a relatively simple client-server game, where when a client is registered, it generates a dice value between 1 and a max value parameter (for ease of explanation we’ll use a default die range of 1-6). At a regular interval (default 2 seconds), the server will run a round of the game, where it rolls a dice which can generate a value between 1 and a max value parameter (again, we’ll use the standard range 1-6). It then wraps this result in a GameResult object, then creating a new Notification object, with the GameResult as its event parameter. The server then; on all clients registered to it, remotely invokes the client method to receive a notification, with the newly created Notification as the parameter.

This method unwraps the GameResult on the client, if the rolled die result of that round is equal to the die value generated for that Client when it was registered, then the server displays a message saying that specific client had won that round, including the die value that was rolled. Multiple clients can win a round in a given round of the game.

### Multi-server/Multi-client support

To test that the framework is operational in a many-to-many way, where multiple servers can invoke methods multiple clients, I created a ClientSpawner class. This class has a main method which generates 20 clients, each with a desired die value within the standard range. The constructor of GameClient registers itself with all active servers, by obtaining a list of them from the RMI registry. The GameServer method to deliver the GameResult iterates through its ArrayList of all NotificationSinks registered with it, and invoked the deliverNotification method on each of them.

### Lost Connection Handling

To handle cases where a client loses connection to the server, whenever a notification is sent to a client, if a remoteException is thrown, the client is assumed to have been lost, and is removed from the server’s list of registered clients, the client would have to register itself again to be added back into the system. Because of my testing implementation, I can’t force close a client like I would an ordinary application, so in my client spawner, I have it drop a random client after 5 seconds of the spawner starting. When testing this with multiple servers and clients running, the client seems to be dropped smoothly from the system.

### Multiple Machine Modification

Currently, all servers and clients run on the same local machine, and therefore share the same RMI registry, making it much easier to share information between remote objects. To modify the program so that it would work distributed across multiple machines, there are a few ways you could do it. You can have any subsequent server launches after the first one find and use the RMI registry of the first server, this would allow the rest of the implementation to remain mainly unmodified, however this centralises a lot of the functionality of the system, and goes against the idea of a distributed system. You could have the client program have a command line argument for the address of a server, and have it register with that server, each server would then also have in its rmi registry, a mapping of all the other servers for the system, so a client connecting to one of the server could attempt to connect to all of them.

## Conclusion

I think that the use of Distributed Objects for a Notification system is a really neat way of implementing it. It handles client server communication very concisely, handles concurrency by itself, and without the need to handle low-level socket logic. It is very easily extensible, after support is added for more than one client and server, any number of clients/servers can be created as long as the machines they’re running on and the application functionality that implements the notification framework can handle it.

Modifying the program to support distribution across multiple machines does increase the complexity of the implementation somewhat, but is still definitely feasible. I don’t think it would be too difficult to modify it for full peer-to-peer communication, you would just have to change the structure so that clients would be able to obtain references to any other client without having to go through a server, which in theory shouldn’t be too difficult within the RMI system.

# 

# Appendix

### Screenshots of the program in operationclientstart.png

