Alexandria University
Faculty of Engineering
Computer Systems and Engineering
Distributed Systems (CS-432)

Bulletin Board System

Submitted By:
Karim El Azzouni (50)
Michael Azmy (52)
Omar Attia (45)

Submitted To:
Eng. Samia Moujahed
Teaching Assistant
CSED

February 28, 2016

Abstract

While using one machine to implement a functionality can be simpler, there is a rising need to distribute the computation on multiple machines. This rising need in fact is the result of the increasing demand to solve more complex problems which needs more computational resources. Distributed Systems is one solution to facilitate this situation. One sort of Distributed Systems is to enable a machine (Client) to perform some computation remotely on another powerful machine (Server) by providing it with some data and then getting the results back. This methodology is called Remote Procedural Call (RPC). Not only does RPC provide a distributed form of computation by distributing needed functionality on many machines, but this adds more scalability, availability and reliability as well. In this assignment we demonstrate a simulation for how RPC/RMI works and how we can benefit from them.

1 Problem Definition

This assignment is meant to be an introduction to Remote Procedure Call (RPC) / Remote Method Invocation (RMI), which means, in simplistic terms, the ability to execute code in a certain context on a physically remote machine. This is a fundamental piece of any distributed system, and is considered the first of many questions in the distributed systems endeavor.

Concretely, we are required to design and implement a distributed Bulletin Board News system, with the normal server-client architecture, plus using RMI as a communication method between the server and remote clients. In this setting, the server has the news, and clients want to access them.

The word "News" refers to a piece of information that the server has, in this setting, it is just an integer number, knowing that it can be anything else, this is just for simplification.

Clients come in one of two flavors: Readers of the news from the bulletin board or Writers of the news to the bulletin board. We made the assumption that a client can only be a reader or a writer at one time, if users wants to assume both roles, they need to have two distinct clients. This simplifies the design and implementation, and produces very little overhead on the user since both clients are very similar in the way they operate.

With multiple readers and writers, we are bound to have the regular synchronization and consistency problems. Access to the news must be organized in a way that maximizes satisfying users requests, but in a consistent, correct manner.

Naturally, the server must have the capacity to handle multiple clients of different types simultaneously, it can not block on one client unless it is for synchronization purposes.

2 Algorithms

The best way to illustrate our algorithms is to have them drawn. A picture is worth a thousand words, after all. How about two?!

Figure 1: depicts how the connection establishment is made between a server and remote clients using RMI.

Figure 2: depicts a working example of two parallel read/write operations, it shows how requests are handled by the system.

In essence, these are the steps taken by our system to solve the problem:

1. The configuration file is loaded and parsed:
   The configuration file contains all the necessary information to run the simulation. That includes: server IP, clients' types and IPs, logging information into the remote machines ... etc.

2. The server process is initialized on its specified host:
   It also creates the RMI registry for exporting and binding implementations.

3. Clients are started according to the configuration file specification:
   Each client runs in its own thread, this makes the simulation more realistic.

4. Clients locate the registry and then lookup the needed implementation.

5. Here is where the real RMI magic starts, without having the actual object, the client is able to get a "simulated" reference to the implementation object through its interface.
   RMI takes the parameters of the call, marshalls them, sends them to the remote host, and marshalls the result back to the remote client. All of this is done completely transparently, of course.

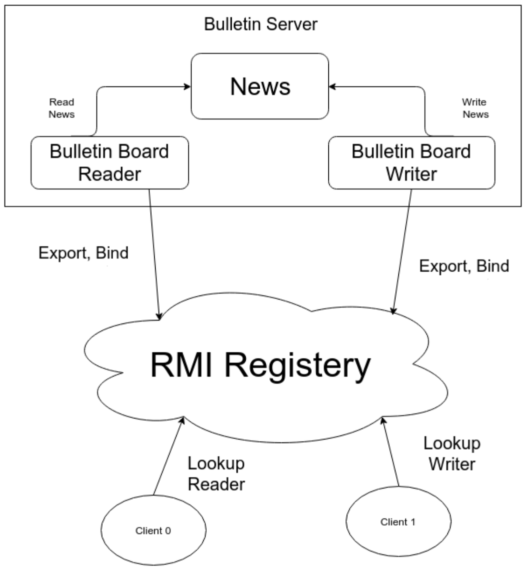


Figure 1: Initial connection establishment.

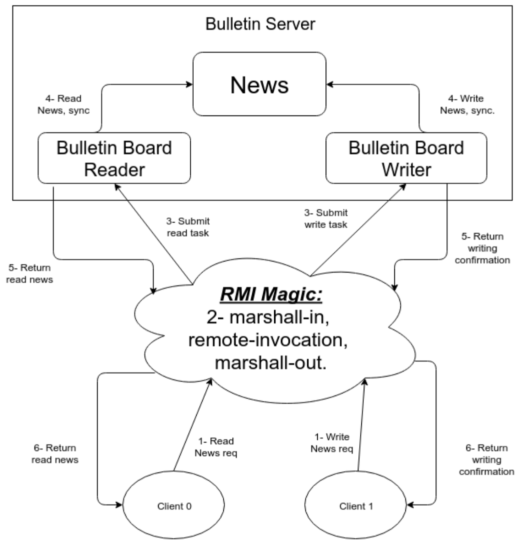


Figure 2: RMI in action.

No.	OS	Processor	Memory
1	Arch-Linux	intel Core-i5	4 GB
2	Ubuntu	intel Core-i7	8 GB
3	Linux-Mint	intel Core-2-duo	4 GB

Table 1: Machines Specifications.

3 Implementation

3.1 Environment

We use three Linux-based machines as illustrated in Table 1.
The Network is LAN where the three machines are connected with a single router. Shared project was maintained using a github repository in order to avoid conflicts and to maximize the team members' productivity by developing concurrently.

3.2 Building

We used The Apache Ant Project tool to build our source files and to run the produced jar files. We even exploited its features to maintain the necessary environment to use our systems and to produce plug-and-play directories that when copied to any Linux-based machine, this machine could be added as a node to out distributed system without any installation.

3.3 Assumptions

In order to run our system on any machine its is expected to have the following minimum specifications:

- Running a Linux-based operating system
- Has Oracle's latest Java JDK installed.
- Has `rmiregistry`, `java` and `javac` commands set to run Oracle's JDK's version.
- Has `expect` package installed.
- Has `ant` installed (for building and installation).

- Has `open-client` and `open- server` packages installed.

3.4 Test Cases

Several test cases were run on the implementation:

- First testing the code on single machine, where it acts as a server, RMI registry and clients. This was to make sure the code is simply functional.
- Testing on two machines, where one acts as a server and RMI registry and the other acts as client. This was to test that the code can be distributed without problems.
- Finally testing the code on the three machines, where one machine acts as a server and has some Reader/Writer clients and the other two machines act as Reader/Writer clients. This was to check the scalability and correctness of the code.

The experiments were run more than 20 times with different parameters. The number of readers and writers were increased each time to add more complexity. The number of accesses also was changed to enable more interference between readers and writers.

4 Results

The simulation results were summarized in the following hierarchy:

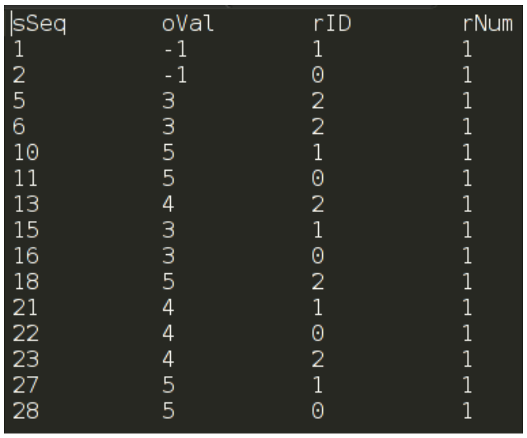
- BServer files: Two text files were produced namely:
  - *server_readers* Which summarizes the statistics of the Reader-Client objects from the server's perespective. This file consists of 4 columns:
    - **sSeq:** The sequence number in which a reader is accessing the news.
    - **oVal:** The value of the news at the time of access.
    - **rID:** The ID of the reader accessing the news at the time of access

    - **rNum:** Total number of ReaderClient objects currently accessing the news.
    - **wID:** The ID of the WriterClient object currently accessing the news.
  - *server_writers* Which summarizes the statistics of the WriterClient objects from the server's perespective.
    - **sSeq:** The sequence number in which a reader is accessing the news.
    - **oVal:** The value of the news at the time of access.
    - **wID:** The ID of the WriterClient object currently accessing the news.

- BClient files: A text file was produced for each client of the system in the format log<id>; where <id> is the integer ID number of the client. Both types of client, ReaderClient and WriterClient objects have identical log formats, whihch is as follows:
  - **Client type:** Reader/Writer
  - **Client Name:** ID of the client
  - **rSeq:** The sequence number of the client object when it first requests access to the News file. On technical basis, when the client object attemps to aquire the synchronization lock on the News object.
  - **sSeq:** The sequence number of the client object when its request to access to the News object is granted. On technical basis, when the client object succeeds on aquiring the synchronization lock on the News object, and right before it starts reading/writing.

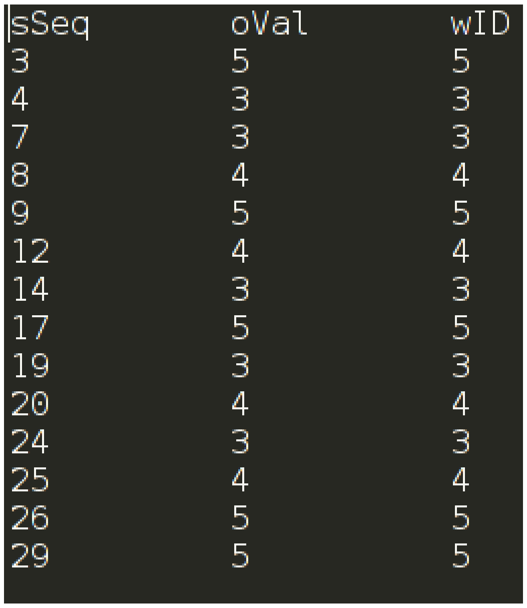
  The only difference between the ReaderClient and the WriterClient objects log formats is the existance of the **oVal** attribute in the ReaderClient objects' logs; It represents the value of the news at the time of access.

The results of the simulation were unexpected, meaning that it could not be anticipated but only verified. This is due to the factor of randomness in the simulation.



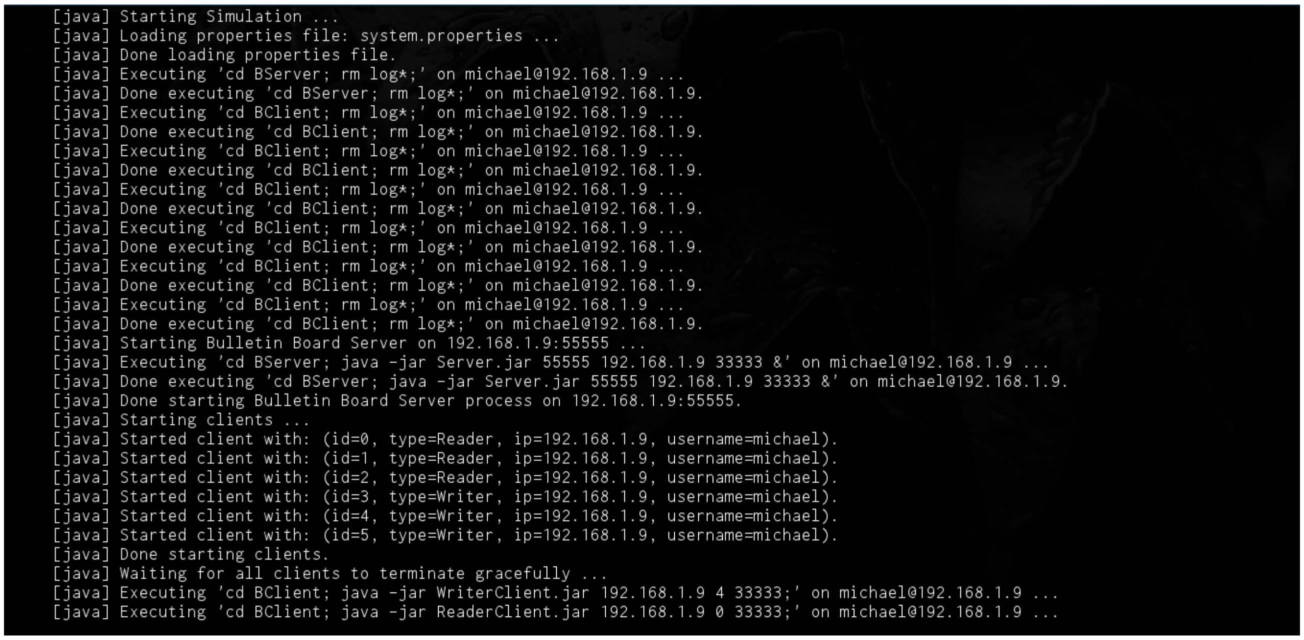
```
sSeq        oVal        rID         rNum
1           -1          1           1
2           -1          0           1
5           3           2           1
6           3           2           1
10          5           1           1
11          5           0           1
13          4           2           1
15          3           1           1
16          3           0           1
18          5           2           1
21          4           1           1
22          4           0           1
23          4           2           1
27          5           1           1
28          5           0           1
```

Figure 3: This is a server_readers sample file



```
sSeq        oVal        wID
3           5           5
4           3           3
7           3           3
8           4           4
9           5           5
12          4           4
14          3           3
17          5           5
19          3           3
20          4           4
24          3           3
25          4           4
26          5           5
29          5           5
```

Figure 4: This is a server_writers sample file



```
     [java] Starting Simulation ...
     [java] Loading properties file: system.properties ...
     [java] Done loading properties file.
     [java] Executing 'cd BServer; rm log*;' on michael@192.168.1.9 ...
     [java] Done executing 'cd BServer; rm log*;' on michael@192.168.1.9.
     [java] Executing 'cd BClient; rm log*;' on michael@192.168.1.9 ...
     [java] Done executing 'cd BClient; rm log*;' on michael@192.168.1.9.
     [java] Executing 'cd BClient; rm log*;' on michael@192.168.1.9 ...
     [java] Done executing 'cd BClient; rm log*;' on michael@192.168.1.9.
     [java] Executing 'cd BClient; rm log*;' on michael@192.168.1.9 ...
     [java] Done executing 'cd BClient; rm log*;' on michael@192.168.1.9.
     [java] Executing 'cd BClient; rm log*;' on michael@192.168.1.9 ...
     [java] Done executing 'cd BClient; rm log*;' on michael@192.168.1.9.
     [java] Executing 'cd BClient; rm log*;' on michael@192.168.1.9 ...
     [java] Done executing 'cd BClient; rm log*;' on michael@192.168.1.9.
     [java] Executing 'cd BClient; rm log*;' on michael@192.168.1.9 ...
     [java] Done executing 'cd BClient; rm log*;' on michael@192.168.1.9.
     [java] Starting Bulletin Board Server on 192.168.1.9:55555 ...
     [java] Executing 'cd BServer; java -jar Server.jar 55555 192.168.1.9 33333 &' on michael@192.168.1.9 ...
     [java] Done executing 'cd BServer; java -jar Server.jar 55555 192.168.1.9 33333 &' on michael@192.168.1.9.
     [java] Done starting Bulletin Board Server process on 192.168.1.9:55555.
     [java] Starting clients ...
     [java] Started client with: (id=0, type=Reader, ip=192.168.1.9, username=michael).
     [java] Started client with: (id=1, type=Reader, ip=192.168.1.9, username=michael).
     [java] Started client with: (id=2, type=Reader, ip=192.168.1.9, username=michael).
     [java] Started client with: (id=3, type=Writer, ip=192.168.1.9, username=michael).
     [java] Started client with: (id=4, type=Writer, ip=192.168.1.9, username=michael).
     [java] Started client with: (id=5, type=Writer, ip=192.168.1.9, username=michael).
     [java] Done starting clients.
     [java] Waiting for all clients to terminate gracefully ...
     [java] Executing 'cd BClient; java -jar WriterClient.jar 192.168.1.9 4 33333;' on michael@192.168.1.9 ...
     [java] Executing 'cd BClient; java -jar ReaderClient.jar 192.168.1.9 0 33333;' on michael@192.168.1.9 ...
```

Figure 5: Log messages of the simulation program Start.java

Tracing the **sSeq** attribute through Figure 3 and Figure 4, the correctness of the operation of the BulletinBoard system is obvious. For example, the first two clients with **sSeq** values of 1 and 2 are shown to be ReaderClient objects (since they appear in the server_reader file), and it is shown that they were reading the News object which had the value of -1 at the time *without altering its value.* The next **sSeq** values of 3 and 4 appear in the server_writer file which indicates that the WriterClient objects with IDs 5 and 3 have *altered* the News object and changed it to their IDs, leaving it with a the final value of 3. After that, a ReaderClient object with the ID 2 accessed the News object; again *without altering its value* to find its value to be 3, and so on. Moreover, a detailed simulation log was printed to the standard output of the machine where the simulation program, Start.java, is executed, and its trace is shown in Figure 5.

5 Conclusion

- Distributed systems are not to be taken lightly! Designing such systems is not a trivial task.
- Even after designing a distributed system, the actual operation is quite challenging due to the uprising compatibility issues between different machines used to host the system.
- Scalability is a serious matter in the world of distributed systems; Deploying the system on a limited number of machines during development and testing phases is not enough, since a number of issues appear when upscaling the system by deploying it on a larger number of machines with different specifications.
- In a distributed environment, systematic testing is a key factor to the success of the whole system. Schemes such as alpha and beta testing used in large software projects contribute to revealing scalability and compatibility issues described above.
- One of the bonus features than we were actually working on was to introduce a self-developed cryptography library. This library would have been basically used to produce a secret AES key to encrypt the system.properties file, and then used at runtime to decrypt it. The absence of this module questions the security and integrity of the whole system, since the server's and clients' machines credentials are shared in a plain text file, that is the system.properties configuration file, which makes the system vulnerable to interception attacks such as the Man-In-The-Middle attack. Unfortunately, we could not spare enough time to resolve our bugs in this module before the deadline, thus we decided not to integrate it in this release.