Distributed Systems Coursework

James King

March 2, 2013

1 System Architecture

The system has been designed to support any number of servers, with the first server to start being recognised as the master server. Clients connect to the master server, which fulfils their requests. The state of the database contained within the master server is propagated to all the other servers whenever it is modified. When the master server goes offline, the server with the longest uptime of the ones that remain will replace it.

All servers in the system are instances of the same Java program. Each server has a list of the addresses and ports of the other servers, and a unique identification number. When a server starts, it attempts to connect to each of the other servers in order to determine what identification number to use. The chosen number satisfies the condition of being larger than the identification number of any other server that is currently online. Therefore, the newest server will always have the largest identification number, and the oldest has the smallest number. This property is used to determine which server is currently the master server. After the new server finds its identifier, the master server is queried to construct the new server's local database to match the master server's.

During operation, information is propagated between the servers in a chain ordered by identification number. When the master server carries out a client request that mutates the database, it forwards that request to the next oldest server. This server, in turn, propagates the request to the next oldest after itself, and so on until the newest server receives the request.

To save time, each server caches the location of the current master server and the next in the chain (if it exists). In the event of being unable to connect to the master or the following server, the location of the new master server or following server is found and cached.

2 Requirements Catchment

Due to the similarity of the requirements for the two servers, they have been implemented to be two instances of the same program. This program detects whether it is the primary (master) or backup (slave) server, and acts accordingly. The implementation also supports any number of servers working in tandem, but also functions for just two servers as required.

Both servers contain a virtual database of student information, including all fields required with the minor modification of splitting *student name* into *first name* and *last name*. It is implemented as an *ArrayList* of *Student* objects on each server. When the front-end client makes a request, it sends it to the primary server. This then propagates it to the next server in the chain; which in the case of only two servers will be the backup one.

When the primary server goes offline, the client will start directing its requests to the backup server. Were the first server to go online again, it would assume the roll of backup to the server which remained online. The user performing the requests on the client program is not aware on any change in primary server, apart from the slightly longer delay as the client has to find the new master server. In the event of all servers being down, the client is alerted after each attempted command.

The front-end client takes the form of a command prompt program. Requests and queries are typed, and in the event of an incorrect input the user is informed as to the nature of the mistake. A help facility is provided that lists each command, and can provided detailed information about the usage of a given command. The database query command structure is quite powerful, in that it can perform complex queries involving many field comparisons.

The main non-prescribed feature of the system is the support of many servers being used as backups, instead of just one. The benefit of more than two servers is the improved probability of at least one server being online at any time. The system can also support two servers but with many possible locations for either, for increased flexibility. Additionally, server programs all have command prompts which can be used to debug the system.

The command prompt system was designed using Java reflection to be easily extendible with new commands. All that is required to add a command is to add a method in either the *Server*, *Client* or *Endpoint* classes with a *@Command* annotation. Also, new fields added to *Student* are automatically exposed to the database query system, also using reflection. Also, the system can contentedly capacitate countless concurrent client connections.

3 Design Limitations

A limitation of the design is that it is possible for a client to connect to a backup server if they are temporarily unable to connect to the primary one, even if the primary server is actually online. This would mean that changes would be made to the backup server, and not the primary one. However, this situation is hypothetical and has not occurred during testing.

Another possible issue may occur when a new server is unable to connect to the last existing server. In this case, it may assume that server's identification number or a smaller one. This situation also has not occurred during testing.

4 Usage Instruction

4.1 Installation

4.1.1 File structure

The files (and directory) required for usage are as follows:

- *bin/* directory, including the 13 *.class* files within
- *departments.txt* file
- *hosts.txt* file
- *server.bat* for Windows or *server.sh* for Unix
- *client.bat* for Windows or *client.sh* for Unix

These files should be in the same directory, with *bin/* as a subdirectory. The Java runtime is also required.

4.1.2 Setting Up Host List

The file *hosts.txt* contains a list of the names, addresses and ports of the host programs to be used. The file can contain at most one host per line, with the three pieces of information individually surrounded by double-quotes. Any characters outside a pair of double-quotes is ignored. A line may be "commented out" by prepending it with a hash symbol (#). The default configuration contains two host definitions, both on the local machine and with ports of

3125 and 3126 respectively. The default file also contains the definitions of another three hosts, which are excluded with hash symbols so as to be easily included again.

4.1.3 Setting Up Department List

The *departments.txt* file lists the academic departments used by the system, by identification number and name. Identification numbers need not be sequential, just unique. As with *hosts.txt*, lines may be omitted with a hash symbol, and there can by at most one definition per line.

4.2 Starting a Server

A server may be started by executing either the *server.bat* or *server.sh* scripts by command line. The command line arguments for the script are:

```
# Windows:
server.bat [rmiregistry-address [rmiregistry-port]]

# Unix:
./server.sh [rmiregistry-address [rmiregistry-port]]
```

If *localhost* is given as the *registry address*, the program attempts to create a new RMI Registry at the given port if one doesn't already exist. Care should be taken to ensure no two servers are started on the same machine using the same port. Also, the addresses and ports used should match the ones specified in *hosts.txt*. For the default configuration, this would be (on Windows):

```
server.bat localhost 3125
server.bat localhost 3126
```

The server program has a command line interface, and a list of available commands and their uses can be shown by typing *help* at the prompt. The server may be safely shut down by typing *exit*.

4.3 Starting a Client

Clients are also started by command line, and require no command line arguments. Simple execute either *client.bat* or *client.sh*, and the program will try and connect to the servers specified by *hosts.txt*. A list of commands can be shown by typing *help*, like with the server, and *exit* can also be used to stop the client.

5 Areas of Improvement

Given more time, I would attempt to send actual *Student* objects using RMI instead of serializing them to a *String* like I am currently. This would simplify the process and probably reduce bandwidth. I would also like to improve the query language to include comparisons between multiple fields, like being able to query students who have more than 200 credits per year ($totalCredit > year \times 200$). Finally, it would be useful for the *departments.txt* dictionary to be synchronised along with the database, to ensure that clients do not have a version that differs to the server's.

6 References

All source code in the project was written by myself for this assignment, using my past practical work as a reference.