Distributed Movie System

System Design

This system stores information about movies and is implemented in Java 7. Specifically the system acts as a distributed key value store of IMDB ID **String**s (e.g. "tt1238343", "tt2938432", etc) to **String**s containing information about the movie. Each backend server in the system acts as replicas of this key value store. A minimal system (not including the client) is spread across at least two machines:

* The client-facing machine. Two processes run on this machine: the frontend/directory process and the object registry process.

Within the frontend/directory process there are three threads running: the main thread which manages a **Frontend** instance and a **Directory** instance, another thread for a **Monitor** instance and another thread for a **Manager** instance.

The object registry process just manages a **java.rmi.Registry** instance. The object registry has two objects registered with it: the **Frontend** instance (under the name "*frontend*") and the **Directory** instance (under the name "*directory*").

* One or more replica machines. On each replica machine there is a single-threaded process running which manages a single **Server** instance.

The purpose of the **Directory** is to act as a central registry for all **Server**s in the system. When a **Server** is started, it should register with the **Directory** on the client-facing machine. The purpose of the **Monitor** is to periodically ping all **Server**s registered with the **Directory** to check if they are still reachable from the client-facing machine, and if not, unregister them from the **Directory**. The purpose of the **Manager** is to choose one of the registered **Server**s from the **Directory** to become the primary server, and tell the **Frontend** of this. The **Manager** continues to periodically poll the directory to see if new **Server**s have been added/removed and to notify the primary server of any new servers that the primary should then propagate updates to. Also, the **Manager** handles when the primary server becomes unreachable (and so is removed from the **Directory** by the monitor), the **Manager** assigns a new primary and tells the **Frontend** of this new one.

When a **Client** wants to connect to the system, they specify the hostname and port on which the object registry can be reached.The client process then gets a hold on the **Frontend** instance and can invoke get(id) and set(id, value) methods on it to get information from and update the system respectively.

The **Frontend** propagates these get and set requests to which ever **Server** is the primary server, as told to the **Frontend** by the **Manager** instance.

Initially, the movie system starts out empty with no movie information. If information for a movie is requested which is not in the collection of movies, it is fetched from OMDB and then cached in the system for later requests.

Maintenance of movie information

Each individual **Server** maintains a local key value store which ideally should be identical to all other **Server's** key value stores. When a get(id) request is issued by a **Client** via the **Frontend**, the primary server just returns whatever value is stored in its local key value store for that id to the **Frontend** who then passes that on to the **Client**. In the case where the primary server does not have a value for that id in it's local key value store, it fetches it remotely from OMDB and caches it in its local key value store.

It then propagates this new key value pair to all secondary servers registered with it. (NB: registering of secondary servers with a primary server is done remotely by the **Manager** object). When a set(id, value) request is issued by a **Client**, the primary server first sets the new key value pair locally then propagates it to all secondaries that have been registered with it. It does not warn if it is overwriting an already existing key value pair

System recovery in server failure situations

The Monitor periodically pings every Server registered with the Directory to see if it is still reachable from the client-facing machine. It does this by calling a ping() method on every Server stub which is just a method which returns immediately and doesn't do anything except act as an indicator to the Monitor that the remote Server object is still reachable.

When a Server becomes unreachable, the Monitor thread unregisters it from the Directory. When the Manager next polls the Directory, it will see that this Server has been removed. If it was the primary Server, the Manager will try to assign a new primary Server from amongst any remaining registered Server.

Limitations

The system is irrecoverable if the client-facing machine fails.

A client can arbitrarily overwrite key value pairs in the system which may not be desirable.

Extra features

There is the potential for there to be more than one **Frontend** instance, though this is untested. This is because the **Manager** instance can maintain a **Set** of **Frontend** instances,

There is also the potential for there to be more than one type of remote source for the system, i.e. for there to be more than just OMDB. The system could then for example be more redundant if OMDB goes down, as it could just fetch movie information from a different API.

How to set up the system

The system is distributed as five .jar archives:

* client.jar

* registry.jar

* frontend.jar (and slow-frontend.jar)

* server.jar

1. On the **client-facing machine**:

Set up the registry by running:

java -jar registry.jar [port]

where [port] is whatever port the registry should listen on e.g. 1099.

Set up the Frontend, Directory, Manager and Monitor by running:

java -jar frontend.jar localhost [port]

where [port] is the port the registry is listening on. “localhost” indicates that the registry is running on the local machine as technically the registry could be on a different machine but this was not tested.

2. On each **server machine**, run

java -jar server.jar [host] [port]

where [host] is the hostname of the client-facing machine and [port] is the port that the registry is listening on.

3. Finally, on the **client machine** run:

java -jar client.jar [host] [port]

where [host] and [port] is as above.

The client program is just an interactive prompt with two types of command:

```
> get [id]
```

where [id] is the IMDB ID of the movie information to be fetched, i.e. a string of the form tt[0-9]{7} e.g. tt1231231, tt2222222

```
> set [id] [value]
```

where [id] is the key to be set in the system and [value] is the value to be associated with [id] in the system.

Note that the timeout for requests is set at runtime via the following flags:

```
-Dsun.rmi.transport.connectionTimeout=x
-Dsun.rmi.transport.tcp.handshakeTimeout=x
-Dsun.rmi.transport.tcp.responseTimeout=x
-Dsun.rmi.transport.tcp.readTimeout=x
```

where x is an integer indicating the number of milliseconds before a timeout. For example, slow-frontend.jar can be used instead of frontend.jar (so that get requests are delayed by 10 seconds) and then start the client.jar with all of the above command line flags having x=5000 (i.e. 5 seconds). In this case, the client will always time out on sending get requests.

What else could be added to the implementation if there was more time

It would make more sense for the **Monitor** to notify the **Manager** thread directly when a **Server** becomes unreachable, so that the **Manager** can immediately notify the **Frontend** and the primary server of this change (or assign a new primary server if it was the primary server which went down). Currently as it stands if the primary server becomes unreachable and a **Client** makes a request to the **Frontend** before the **Manager** has updated the **Frontend** to point to a new primary server, an exception is thrown, but at least the **Client** is notified of this and to try again soon (when the **Manager** has hopefully assigned a new primary).

Any public domain source code

No public domain source was used.

Testing

In the test/ directory are five text files that are logs of the client, server and frontend processes. The testing was done as follows; first registry.jar was set running, and then frontend.jar was set running (log is frontend.txt) and three servers (server1.txt, server2.txt, server3.txt). The client was then set running (client.txt). Some queries were carried out by the client, and then server1 (the primary server) was stopped; then server2; then server3. Then server4 was started (server4.txt) and some more queries were executed to show that data loss had happened (i.e. the system can't survive every backend server going down). These log files are annotated with more details (lines with *** on them).

References

Lecture slides

http://docs.oracle.com/javase/tutorial/rmi/ - Java RMI tutorial which was extremely useful for learning about RMI and how to use it

http://www.vogella.com/tutorials/Logging/article.html – tutorial on how to use the Logger class in Java

http://www.vogella.com/tutorials/JavaXML/article.html – how to deal with XML in Java

http://vv.carleton.ca/~cat/code/rmihell.html – Details about the equality relation between remote objects and their stubs

https://stackoverflow.com/questions/1822695/java-rmi-client-timeout – How to set a timeout on RMI