Overview

The problem was to use RMI, REST and distributed hash tables to implement a computational task submission system. This would involve a user on a REST web service selecting a task to fulfill, which would then be assigned a designated DHT node. The user would then require the functionality to download the results of any completed task. These nodes would be constantly communicating through RMI.

My solution provides users with an 'index' screen, from which they can decide whether they wish to carry out a task or view the results of a previous task. The designated node would then carry out the task asynchronously. For my system I had to very carefully consider the relationship and communication between nodes. This was vital to prevent data loss, which I achieved by storing backups of data on the successor of a node. I also had to balance this with close attention to efficiency and scalability.

My code is compiled by typing make into the command line in the main folder. The RMI registry is then started by entering rmiregistry and new nodes are initialised by entering Java ChordNode <key>. Finally, to open the REST index, we need to go to localhost:8080/myapp/rest, from which tasks can be sent and downloaded.

Design

REST

My REST class contains five methods - one is newFile, which firstly takes a text file submitted by the user and converts it from an input stream to a byte array (this requires a second method, 'convertInputStreamToByteArray'). It then scans the file and returns an error message if no file, task or key are entered, or the key entered already exists. It also checks a method 'aliveNode' which ensures that the random node found is still alive and dead nodes are unbound. If everything is okay, it will add the file to a randomly selected node * via the 'put' function in the RMI server.

* I use random selection of nodes regularly to maximise the distribution of the workload across the network. This is designed to reduce the risk that certain nodes become overloaded whilst others could be working considerably more.

The fourth method in the REST class is getTasks. This is invoked when the user wants to get results, and outputs a list of completed tasks as hyperlinks. It scans each node in the RMI registry for tasks assigned to that node, and if a task is found, a string 'html' is appended with a line break and a hyperlink to the results page of that task. This method unfortunately breaks when a node leaves the network, an issue that I was unable to fix in time. However, a result can be viewed in XML by manually entering the URL of the 'get' result (localhost:8080/myapp/rest/get/<key>).

Finally, the REST class contains a method called getString, invoked when a task has been selected from the completed list. It sends a 'get' request for the key of the task and receives the task result and type of task. It then looks up the type of task and outputs the result as XML.

Tasks

The original idea was to implement three tasks that users can carry out to manipulate the contents of the file. However, I extended this to include two more tasks to further test my system. Along with the number of words, average length and most common words, I have implemented methods that return the longest word and a random word.

RMI

The RMI is built around a distributed hash table of nodes, each of which represents a 'server' that the REST class can connect to. Each RMI node joins the network by entering a key connecting to a randomly selected second node from the RMI registry. The key is then hashed into integer form and its predecessor and successor in the network are calculated during an initial 'stabilisation' period.

The maintenance tasks that take place in each node recurrently include the basic chord methods fixFingers, stabilise, checkPredecessor and checkDataMoveDown. Stabilising is particularly critical as it constantly checks if the successor or predecessor have changed (or left the network). If they change, a period of stabilisation is entered for 5 seconds whilst these are adjusted. Put and get are temporarily disabled during this period, which may cause delays in the system but more importantly prevents connection to the wrong nodes.

A further maintenance method is checkDataStore, which constantly seeks to find if a new task has been submitted to the node (tested by a boolean variable 'completed' which is initialised as false). This method also checks if the data has been backed up (again by a boolean variable), and calls the backup method if it hasn't.

My backup method simply creates another instance of 'Store', but in a seperate vector called 'backupStore'. The backup is an exact copy of the main store, other than the 'backedUp' boolean check which is always set to false. A store will always be backed up in its successor, and this is constantly monitored in a maintenance method. Three test cases trigger movement of stores and backup stores - if the successor of the store-holding node changes, if the store-holding node leaves the network, and if the node holding the backup leaves the network.

I chose to implement this simple backup method due to time constraints. This is quite an inefficient method as it requires two copies of the store to be held. Furthermore, I failed to successfully achieve this in time and one page of my web interface (getresults) does not adjust when a node leaves the server. This may be only a small flaw in my system, but it damages the effectiveness of the REST server.

I tested the system thoroughly by adding nodes and tasks to be carried out, then adding further nodes that should activate the condition of moving a backup or store. I then checked that the results downloaded successfully on the REST server. I further tested this by then removing nodes (particularly nodes holding tasks) and seeing how the other nodes were affected.

Personal Assessment

I found this to be an interesting and highly challenging coursework, which built on previously gained knowledge of distributed systems. However, due to the complexity of the concepts being implemented, I found the time constraints to be insubstantial and it considerably reduced the amount of time I could dedicate to other modules. With a further week, I feel that I could have developed a more complete implementation. However, I enjoyed learning practically about peer-to-peer networks and how the concepts of web services, RMI and DHTs work together.