Introduction

The purpose is to build a computational task submission system, combining RMI, REST and DHT. It allows users to select tasks from the website, which can range from text files to other file types such as csv. This will then send through RMI, to the Chord node network, which when a node receives this, it will process the task, such as total number of words, most frequently occurring word, average word length and more; the node will store this information and return as an XML document to which the user can download.

It runs on **localhost:8090**

To run the program, run in order:

- rmiregistry
- java NodeHandler
- java RestKit (inside website directory)
- java ChordNode [NAME] where NAME is the name of the node you assign.

Localhost:8090 will have the landing page,' /upload' to upload tasks, '/getTask' to retrieve results and download.

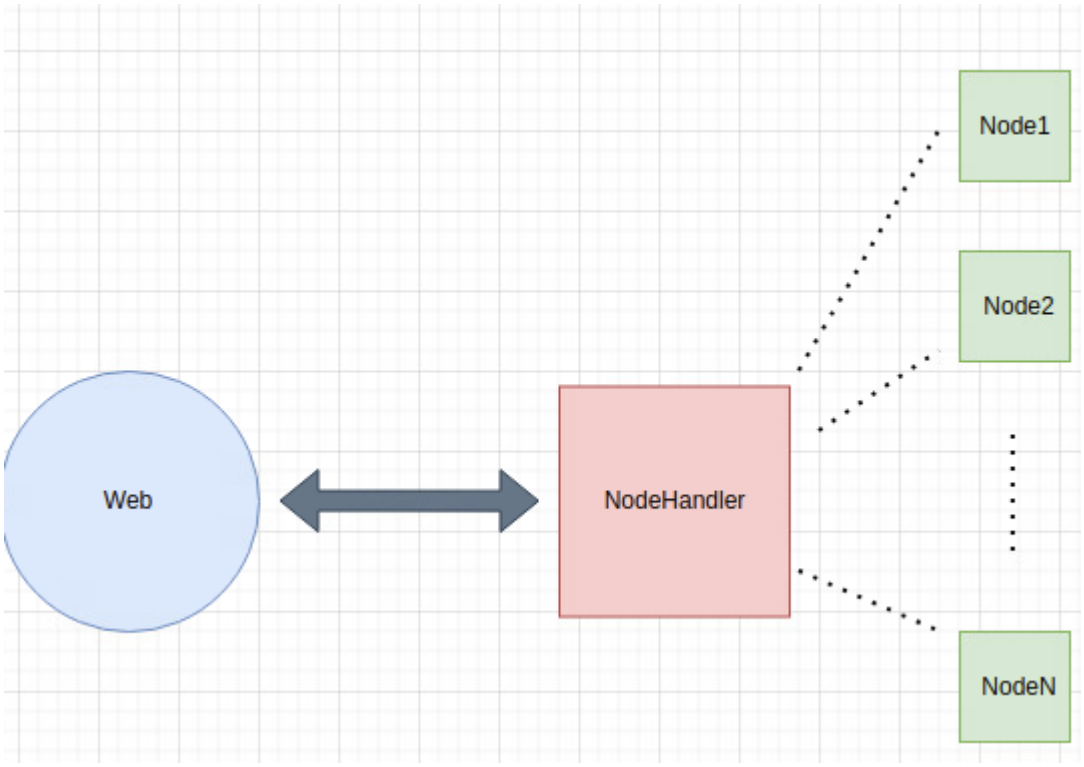
Design

The **Web.java** handles how the pages are represented, this is done by embedding html code and sending a response back by writing the bytes as an output to the user. The landing page allows the user to click two buttons, 'upload' and 'results' which will look for a GET request, then by looking at the request information, it will determine if the URL exists and call a function appropriate to this, such as '/getTask' being the URL for handling results of tasks which calls getTask().

Instead of hardcoding the URL paths, I used a splitter which allows for depth, by taking the URL and dividing into chunks so that it can check without checking if the request.resource equals a specific file path, which was useful for the lab in week 2.

The two main pages, being '/upload' and '/getTask', upload will allow for the user to submit a file of their choice (txt or csv) and send a POST request which will be detected and sent to the NodeHandler (NH) through the rmi registry which then assigns a node the task being requested. If an invalid file type is sent, it will not appear on '/getTask'.

Similarly, '/getTask' will query all the chord nodes for the data that they hold and send back any information they have, which if any information is found, will tell the user which tasks have been processed, so that they can then submit the name of the file and retrieve the data which is then downloaded as an XML file, done by taking the byte array of the processed task and writing this back to the user, if no tasks have been submitted, then this will be told to the user. Download files are in the main directory (in backup).



The **NodeHandler** (NH) receives any requests coming in from the website, such as tasks and from nodes, so if a task needs to be fetched, the NH will query all the nodes for this. NH knows about the existence of all Chord nodes on the network and maintains a list of them, it will then randomly assign a node to handle the task; initially, I decided to loop through the list of nodes and pick whichever was available, however this would put burden on the first node, due to small amount of requests occurring, it would be most likely to have most of the requests and the final node would have the least, which would be inefficient. Using a thread, the NH will continually check if any nodes have died, if so it will remove it from the list of known nodes, which allows it to perform other operations without being halted.

I tested by killing the NodeHandler and the individual Chord nodes will continue to work, but data cannot be fetched anymore, as they need to pass through NH, but once brought back online, the nodes will reconnect.

Chord nodes initially join the rmi registry by binding the name given in the terminal and adding themselves to the list of known nodes in the NH. In addition, maintain their successor and predecessors, so they only know about who their successor and predecessor are, initially the successor is itself. Chord nodes need to be able to join to each other, so it will try to find it's successor, if the current successor is itself, it will just make the successor of the node requesting to join, otherwise just check by using the range; example being 120 trying to join a node of 110 which has successor of 130, 120 is in between these two, so it would be assigned to be the successor of 110.

If outside this range then just look for closest preceding node, which does a look up of the node's finger table, to prevent lookup times from being O(N) to O(log(N)), by allowing it to skip checking of every node on the network.

By running a thread, the node will continually update its parameters without stopping other processes from running, such as checking if the current predecessor is correct and whether it has died, recalculating values for the finger table continually and the successor, moving data down if it belongs to a different node.

The put() method, allows for the node to receive byte data along with a key, the successor is first found to this key, then we process the task inside a thread so that it is asynchronous and then inserted inside into an instance of a Store and stored in a hashmap, by hashing the filename of the received task this becomes the key held inside the data store, the data is held in the node responsible for it.

However the data can be in the wrong node, so they are moved down to a predecessor if its within the range, I determined this to be by checking if the predecessor is in the correct position first and if the data is greater than current key or less than predecessor key, so will be moved down.

The tasks are processed in a function and by using an XML builder with Java, they are returned as byte arrays. Files are checked for their file type (txt or csv, with csv being the extension I added), otherwise will not be added to the data store, they are processed by calling individual functions for the operations, such as total number of words.

The baseline was for text analysis, however I added the ability to also do csv files, which performed three operations on the file, such as counting rows, columns and actual words in the file (to prevent it from accepting purely numbers as a 'word'), if the input file is not either of these, it is handled by returning a null and so is never processed or added to the data store.

The get method will find the node responsible for storing the file via findSuccessor(), hash the key (file name) and perform a lookup in the hashmap and return the data it finds, if any, within that node, so it uses the finger table to query for this.

In addition, I added replication, the nodes save files of their data with the successor inside a folder ('storage') so in the event a node dies, it can fetch this data so will continue to work even after a node responsible for a task has died. E.g. 120 holds data and dies but periodically does backups, 140 will notice this by checking if predecessor has died and then load in the file under its name, which will contain its own data along with 120. Files are saved as the name of the successor, so even if that node died and was replaced by a different node with same key, it would not matter as it would still function the same, also provides an easy way to know which file belongs to a node.

I tested the nodes by killing them individually, then by trying to ‘upload’ data and retrieving tasks in ‘getTask’, so the nodes would continue working as expected e.g. nodes 47,120,144 if 47 is responsible for a task and died, 120 will load the data and would take over and continue the tasks that 47 would do, if 47 comes back online 120 will push the data down to 47. The terminal will also reject any invalid names given to nodes when initialising them.

Personal Assessment

The coursework built on a concept that we have covered before but in greater depth which was interesting to learn about, it was sufficiently difficult and allowed to implement an interesting concept. However, the amount of information initially is very overwhelming and there are too few labs per week to get help from, it would have been better if the week 3 lab had more information around it as even after week 3 myself and others were still struggling to understand the Chord.

I have learned to implement a complex data structure and java REST which I have not used before.