Overview:

The problem proposed by the coursework is creating a system which operates via a web browser. In this web browser, one should be able to submit files as well as generate tasks. Once the requests are submitted in the web browser, they are directed to a REST service, then processed by a DHT system and returned. All of this should also implement RMI for communication between each class.

Design:

The design I tried to implement wants to allow the user to submit files via web browser, which then also allows the user to choose a task they want to do from some pre-existing tasks. Once the user chooses the task they want to do, they are prompted with an automatic download button that will download the finished task onto the computer.

In order for this to work, the following commands must be ran in terminal:

- rmiregistry
- java RestKit: we run this so that the web browser is functional
- java NodeSupervisor: this class handles the nodes, what they do, and acts as an intermediary between the REST server and the DHT server
- java ChordNode [word]: the word can be replaced with any word (i.e. key, letters, etc), and it will automatically start a new node and create a key for it

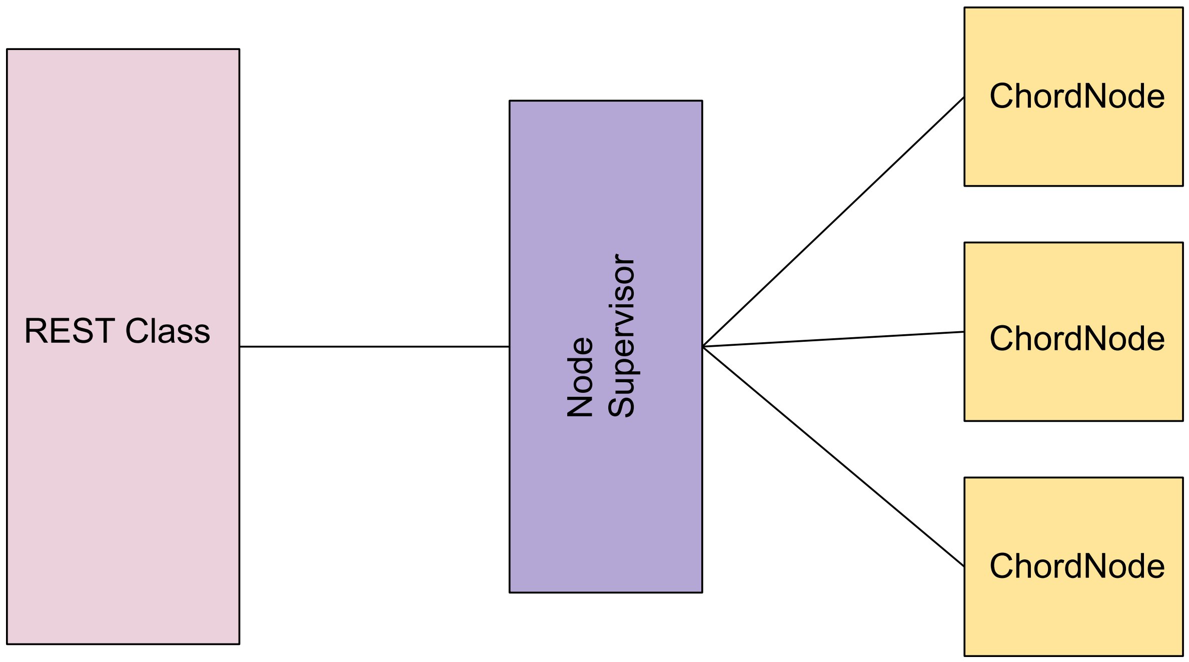
The other classes act as intermediaries or places to store information and don't need to be run as they will be contacted via functions.

The RestKit class is implemented as our REST class; the reason I chose my REST class as RestKit is because the class is already interacted with by the user and thus it is easy to mold it and its' additional classes into working together via RMI.

The NodeSupervisor class is my LoadBalancer. This class acts as an intermediary between receiving data and parsing it in RestKit and ChordNode respectively.

The ChordNode class functions as the one taking in data, editing it, and sending it back. Multiple chords can be started and the one who does the function is random. If a node dies, then the other nodes will continue searching for new predecessors and successors. The user will also be let known via multiple notices that a node has died.

While information is processed via the nodes, it doesn't directly go there. The information goes from the RestKit, to our Web class, then to the NodeSupervisor class, which will then assign a random node to work on it. When the node is done processing and modifying the information, it sends it back to the NodeSupervisor, and then the NodeSupervisor then sends it back to the user via the REST class.



The diagram above explains the logic; The REST class sends the information to the NodeSupervisor, which then sends it to the nodes.

The way I chose to do it like this is so that there are less chances of crashing, as well as the fact that it is easier to have multiple nodes working on tasks at the same time.

While I didn't implement multiple tasks in my code, the web page designed that opens up and allows the user to pick the task is designed to show that there were more tasks planned. Additionally, I would have liked to have been able to fully finish my design; Currently, if a node is killed and it was processing the information, the task needs to be restarted.

I have managed to however get quite far with my design. The dataStore vector stores keys and final values, and the design is quite good with handling crashing and issues. There's very few errors and I have focused on making it look appealing to the eye, easy to understand and also functional. Most information about the client and the nodes is stored in the NodeSupervisor class, but processed in the nodes.

Additionally, there's an extra class created that makes XML files via a byte array. The class creates an xml document, fills it up, and then turns it into a byte array. The class then returns the byte array to the node, which stores it and notifies the NodeSupervisor class, which then sends it back to the user via a downloadable link.

To access the two pages we can use in this we need to open the webbrowser and then two tabs; one will be on localhost:8080/tasks, and one on localhost:8080/upload. The upload page allows us to submit files, and the tasks page explains the tasks one can pick from and then allows the user to download them via hyperlinks.

All in all the design is fairly functional, easy to understand and use. There are more features that I would like to implement, such as a task processing page, more failure methods and generally perhaps some more taks, classes, or things to do, but for now it is still able to do quite a bit.

Personal assessment:

I thought that the coursework made for an interesting challenge, but the lack of labs made it quite hard to actually ask questions. There were a total of five labs, out of which only two were allocated to coursework; and while the first three labs helped understand the very foundations of it, much more of the coursework was left to us to figure out.

While I did not expect the teachers to tell us everything, it did feel a bit barebones and I found myself struggling at times with understanding what I had to do exactly. Once I understood what is needed, getting the grasp of it was easier but I will be honest that originally understanding what exactly was asked of us was not easy.

I felt like I was able to get quite far on the coursework but due to time, other classes, and generally lack of labs I could not achieve every point of it. A lot of the coursework required all of the parts to be functional in order to be tested, and that at times made it hard to know if you're heading in the right direction.