A Distributed Hash Table using Java RMI

Andreas Hallberg

January 3, 2014

1 Introduction

This report will discuss the implementation of a Distributed Hash Table. The purpose is to present the implementation of a DHT using Java RMI and the Chord routing protocol. The implementation will then be evaluated with some performance tests.

2 The Task

The task was to develop a k-ary DHT which can hold up to N active nodes. In my implementation, the k- and N-values are set at compile time, and cannot be changed during runtime. For simplicity k is set to 2 and N can be any power of 2.

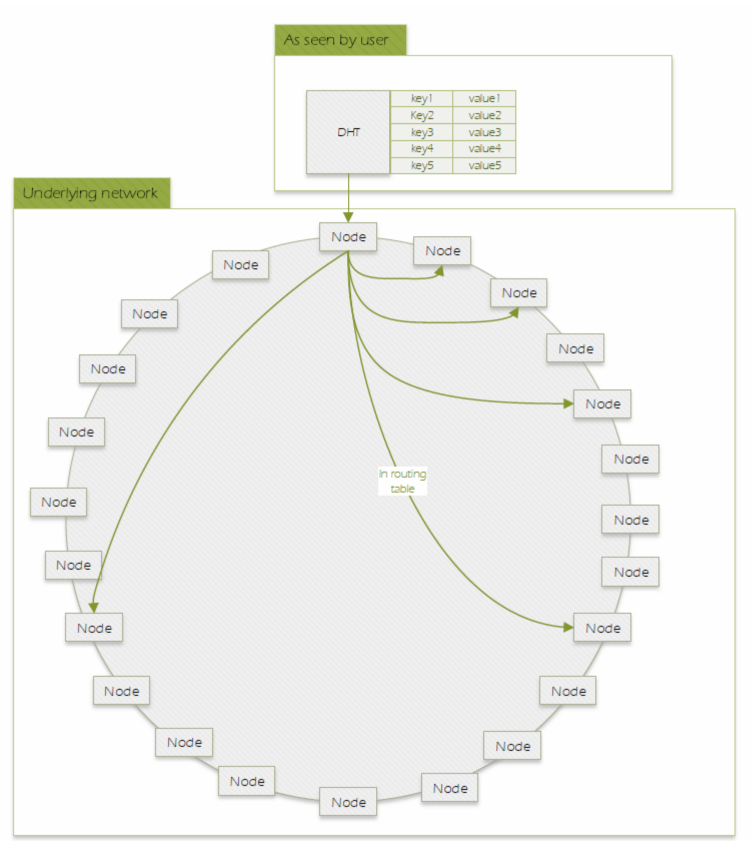
2.1 Architecture

The DHT consists of two layers. The **Application** layer and the **Network** layer. The Application layer provides functionality for a user/programmer and some transparency, and the Network layer takes care of the communication between DHT instances/hosts/networks. Since the DHT has to be scalable, fast and robust, it is built as a peer-to-peer network of Nodes as seen in Figure 1. Nodes can join/leave any time they want. Note that I haven't spent that much time on the *robustness* of the network, so it can't really deal with network errors or crashes at this point.

2.2 Interfaces

To use the DHT I have chosen an interface very similar to that of a normal Hash Table. With methods such as put(), get(), remove() and so forth. In addition to those, I provide a few methods that can be used to join/leave networks. The interface is called *DHT*.

The communication between nodes in the network is done via a remote interface *Node*. It provides methods that lets a node route messages (RMI-



key1	value1
Key2	value2
key3	value3
key4	value4
key5	value5

Figure 1: User's view of the DHT, and the underlying network. The green arrows represent the nodes that a node can reach directly, since they are present in the starting node's Routing Table (2-ary Chord).

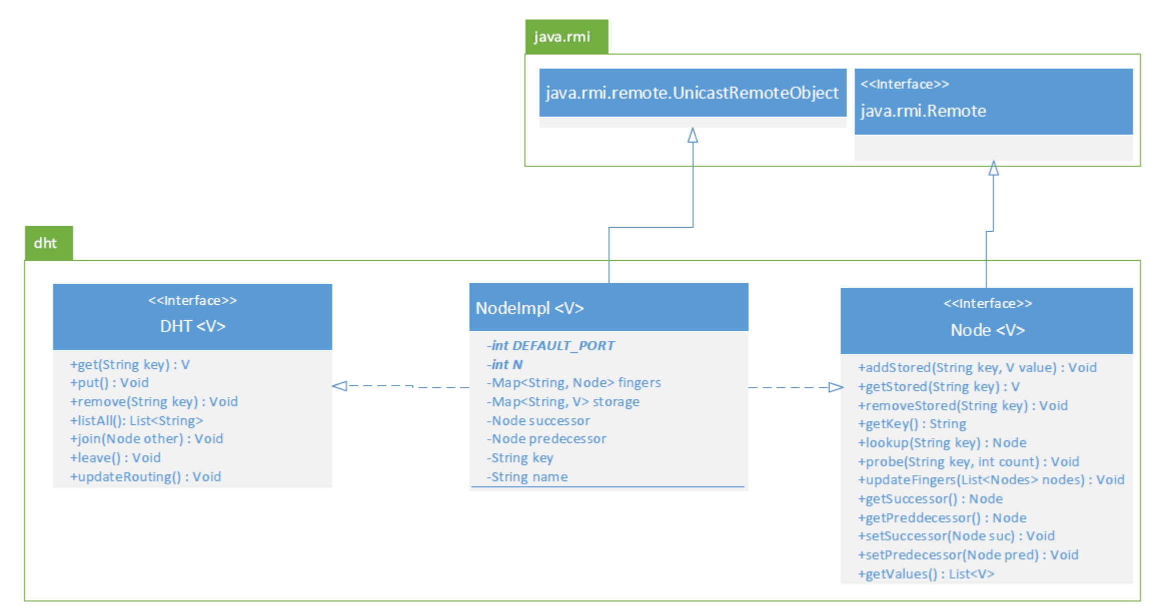


Figure 2: UML class diagram that shows the different interfaces and their uses.

calls) to other nodes. And to perform some lower level lookup(), get(), put() etc.

The interfaces and their dependencies/implementation can be seen in Figure 2.

3 Performance

I wasn't able to do any *real* distributed performance tests, since they're quite hard to do without the appropriate hardware and test-conditions. However, I did some tests on a single computer to see how well the DHT performed. The system used in the tests and during implementation:

- CPU - Intel Core i5 2500k @4.3 GHz (4 cores)
- RAM - 8GB @1600MHz
- OS - Microsoft Windows 8.1
- Java - JDK 7 (1.7.0-45)

The tests were performed by first creating a new DHT with a number of nodes, and then putting/getting a number of values into the DHT. I measured the time it took to create/put/get and printed the results.

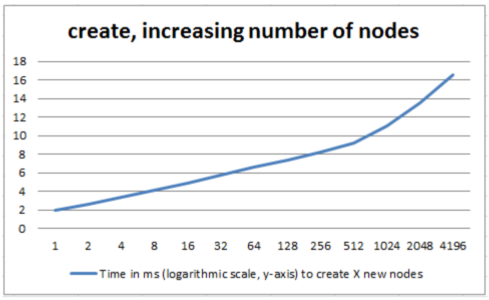


Figure 3: Time (ms) measured when creating a new DHT with different number of nodes. Note that the time (y-axis) is on a $\log_2$-scale, to better see the relation between the time and the number of nodes

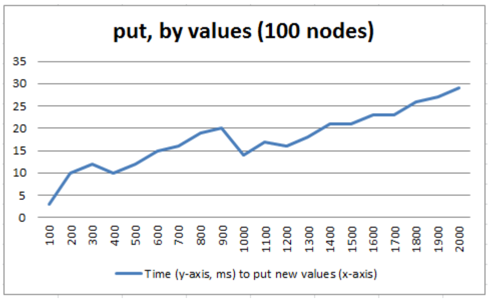


Figure 4: Time (ms) measured when putting different number of values in a DHT with 100 nodes.

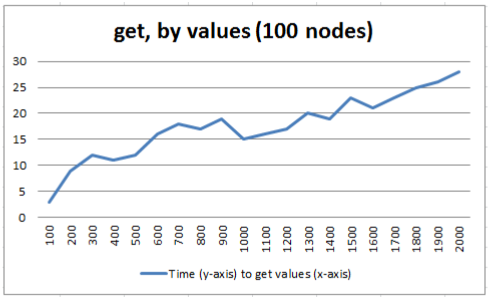


Figure 5: Time (ms) measured when getting different number of values from a DHT with 100 nodes.

Since the power of a DHT is only really utilized when running in a distributed environment, these tests may not show anything of importance... Although one can note that the time it takes to put/get values seem to be *O(N)*. I'm not sure if that is expected, but I suspect that the increase in time would be smaller if the DHT was actually distributed and put/get operations could be done concurrently. I did some tests with concurrent put/get-operations with threads, but the results varied by a lot from one test to another, so I can't really trust them.

I also noticed that the time it took to create new nodes increased significantly with the number of nodes. This is probably because of my routing algorithm, where every node has to re-work its' routing table every time a new node joins/leaves the network.

4 Conclusions and improvements

Overall I think it was a successful project. I implemented a working DHT, which has most of the functionality that is expected from it. Although I had done a similar Chord-implementation in another course, I still learned a lot from this project. Especially regarding java RMI, Chord and Peer-to-peer networks.

Since this implementation doesn't provide robustness and error-correction (dealing with nodes failing), that is definitely something that I will have to add later. I have an idea of how to implement it by keeping references to nodes other than the successor- and predecessor-nodes, and introducing some storage redundancy. But since that was outside the scope of this assignment, I will save that for later.