INF-3200 Assignment 1

Peter Munch-Ellingsen
peterme@peterme.net

Department of Computer Science
Faculty of Science and Technology
UiT The Arctic University of Norway

04.10.15

1 Introduction

This report outlines the design and implementation of a distributed system for storing and retrieving data.

2 Technical background

2.1 Distributed systems

For certain tasks a single physical machine might not be sufficient. The option is to create a communicating network of machines which each take on some role in a coordinated effort to complete the task. This kind of network is called a distributed system. One instance in which this is often used is computationally intensive tasks where the load can be parallelized. Another usage for such systems is networked systems where a lot of consumers will be using the service in question. A single machine will have a hard time responding to a vast amount of machines especially if it has to do computations for each connection. In this report a design and implementation of a distributed map of values is the main focus.

2.2 Key-value maps

Maps of values are often of the so called key-value variety. This means that the values are accessed by a key such that each key represents a single value. Other than this they have no specific structure but can be implemented in multiple ways, typically as a hash-map.

3 Design

3.1 Distribution

The distribution in this system is done in two parts. A set of nodes which keep data is set up and a single node bridges the gap between outside connections and the internal network by randomly forwarding data to any node within. This structure can be seen in figure 1. The nodes each keep an ordered list of the other nodes to determine which node a value belongs to. This is done by a simple hashing algorithm which is modulated over the amount of nodes to uniquely determine which node a certain key belongs to. The command is then forwarded to that node. All nodes evaluate that the value is in fact bound for them before storing or returning data.

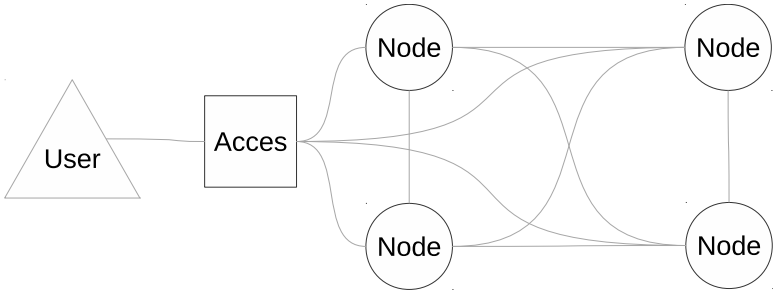

Figure 1: Sample system structure

3.2 Interfaces and analytics

To interface with the system one can contact the server node. The server is however set up to be as stateless as possible and therefore terminates all connections upon fulfillment. So to be able to more efficiently work with the system there is also an interactive shell implemented. It circumvents the

server entirely and does itself randomly pass on commands to the network, while it is not required to run along with a server parallel usage of both systems is possible. The shell also allows to get simple analytical information about the network by querying each node for a count and outputting it to the user to be able to tell how the load is balanced within the network.

As per requirement both the server and the shell supports the two basic commands *GET* and *PUT*. These commands are used respectively to get values given a certain key and to put information into the map given a key and a value. The shell additionally supports the *INFO* command to get the per-node count of stored values and the internal commands *.quit* and *.help* the first of which shuts down the shell and the second which outputs information about how the shell can be used.

4 Implementation

4.1 Language choice

For this exercise I chose to write the implementation in Clojure. Clojure is a functional, Lisp-like language which runs on the Java Virtual Machine or JVM for short. As it runs on the JVM it is able to be run on the uvrocks cluster which was a necessity to get a proper distributed system. Clojure shares a lot of nice features with other Lisps some of which came in very handy for this implementation. It was however chosen mainly for curiosity reasons and as a way to expand personal knowledge into functional programming.

4.2 Networking

Since the implementation is written in a Java based language it uses regular Java sockets to perform networking between nodes. Each time a request is sent, either between nodes or from node to server the connection is kept open and the response passed back the same way the request came in. In figure 2 this is visualized, the user sends a request to the server which randomly picks a node (in blue) to forward it to, this node calculates that it is not the actual recipient and forwards the message to a second node (in red). When the second, red node has handled the request it answers back to the blue node which in turns answers the server which then answers the user. This is done to ensure that when the connected client receives a response from the

server the system reflects the eventual changes done by the issued command.

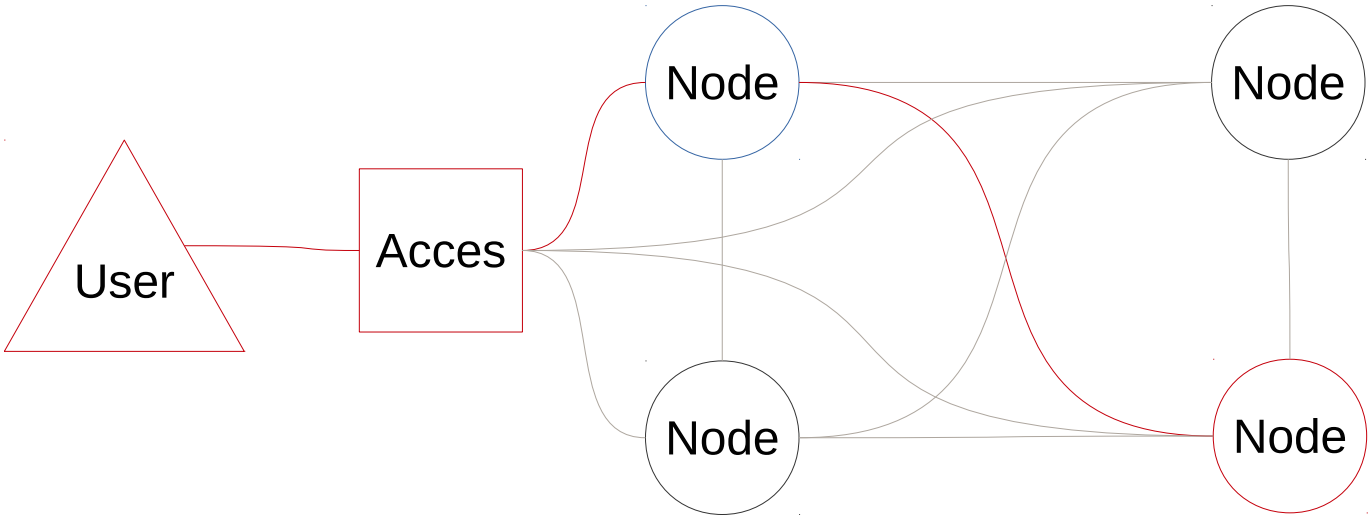

Figure 2: Example path of a request

4.3 Modes

For ease of use the entire implementation is contained in a single program. To determine which 'mode' it should be launched with it accepts an argument. Since the program needs to be launched as a node on multiple machines some of the modes will perform this job automatically. There are currently five implemented modes.

node is the mode for each storage node in the system

server is for running the interface server to the system. This mode also starts the nodes in the network

shell runs a shell which directly interfaces the nodes

shellserver runs a shell but also starts the nodes in the network

killnodes is used to kill all the nodes in the network to shut of the entire system. This is used if the program is not able to properly shut down the nodes itself.

4.4 Nodes

The nodes are designed to respond to all requests. If the request is not understood they will return an error but otherwise they will return according to their commands. For debugging purposes all nodes also writes out logs which details every connection to the node. These logs can be inspected to see how the data flows through the network.

4.5 Hashing

I any system that relies on hashing the hashing algorithm must be sure to generate evenly distributed hashes for plausible sets. This implementation is no exception. Worst case scenario the hashing would place all keys on a single node and the system would turn into a single machine system with extra overhead. The current implementation wraps the hashing algorithm in a function so that it could easily be switched out if a certain set of data is known to work better with a certain algorithm. The current algorithm in use is a prime factorial based algorithm which is known to give fairly well spread data for text based input.

5 Discussion

The implementation included with this report handles an arbitrary amount of hosts. This combined with the possibility of running multiple access servers on the same system allows the network to scale well to the required scenario. It does however not in it's current form support adding or removing nodes.

5.1 Dynamic node allocation

To make the system better suited for unpredictable or dynamic work loads a system to allow it to dynamically allocate and deallocate new nodes should be implemented. Short of using a different overall structure it can be implemented into the current solution however doing so without a certain performance hit is not trivial. The most obvious way to do this is simply to allow the network to receive new nodes. This would make the hashing produce different values for some keys and would mean that each node had to recalculate each hash again. In such a system care must also be taken such that the nodes are able to properly redistribute the keys without conflict. A node

deallocation could also be implemented in a similar way. This system would work but not without potentially freezing the network as it reorganizes the keys which potentially could be an expensive task.

5.2 Load

Functional languages like Clojure contain only immutable data types and are side-effect free. This means that they lend themselves very easily to parallelism. Because of this the implementation has been written to be parallel. The obvious benefits to this is that it is able to handle load much better. Since the implementation waits for values to be properly seated before responding to connections it means that a node would normally hold until it received a response from the node the command was passed on to. With parallelism the same node can keep multiple connections open at the same time and the waiting threads can be kept in the background until the response is received. This means that the system is able to hold a much higher load and a much larger amount of requests per second than it usually would.

The main server node is likely the bottleneck in this system as it will need to keep all connections open until it receives a response from the underlying nodes. The implementation however does as previously mentioned run multiple servers on the same node network so multiple access points is a possible solution to alleviate this bottleneck.

5.3 Other environments

This implementation was written with the uvrocks cluster in mind but this assumption did not influence the design to any certain degree and it would be entirely possible to run this on a different network. As the maximum number of jumps required to get the request fulfilled is only two (if counting the initial server to node jump) it means that the implementation could even be applied to a network with higher latency as well allowing even for Internet networks to use it.

6 Conclusion

In this report and the accompanying source code there has been shown a simple design and implementation of a distributed key-value map. An at-

tempt has also been made to reason how the implementation would work under various loads and in various environments.