KTH V17P01 Distributed Systems Advanced Course: Preliminary Project Report

Fannar Magnusson (fannar@kth.se)
Thorsteinn Thorri Sigurdsson (ttsi@kth.se)

February 19, 2017

1 Introduction

We are writing the system in Java, using the template that was given.

We intented to re-use the Scala components we had already implemented in the programming exercises, i.e. the broadcast components and the eventually perfect failure detector. We spent quite some time trying to use these Scala components in our Java project, running into a host of problems. Eventually we gave up and decided to rewrite the components in Java.

2 Partitioning

We use hash partitioning of strings to distribute data among the nodes. We use consistent hashing in order to minimize rehashing of data later on when we start to implement reconfiguration of the system (node joins and leaves). We hash the IP address of the different nodes, and find the remainder after dividing by 100.

```
hash(Node NetAddress) mod 100
```

This gives us a key for the node, in the range 0-99. A node is responsible for data elements with keys greater or equal to its own key, up to the key of the next node. For example, if we have two nodes, one with key 55 and the other with key 70, the first node will take care of data elements with key 55-69, and the second node will take care of data elements with keys greater or equal to 70.

Similarly, we hash the data element IDs and find the remainder after dividing by 100.

```
hash(Data element ID) mod 100
```

This gives us a key in the same range as the node keys, i.e. 0-99. This key is used to determine which node will be responsible for this data element.

Since neither hashing function is dependent on the number of nodes in the system we do not need to rehash everything if a node joins or leaves the system, moving a large amount of data around. We only need to move data to and from the nodes adjacent to the node that just joined or left. For example, if a node with key 60 joins the system from our previous example, it will take over responsibility of data elements with keys 60-69 from the node with key 55. Similarly, if a node leaves the system, the node below it in the key space will take over responsibility for its data elements.

3 Replication

For our choice of the right replication algorithm we needed to plan ahead and make sure that what we go with will fulfil the linearisable property when it comes to implementing the PUT operation. Basically we can choose between having Active or Passive replication. For the Active replication we need to implement total order reliable multicast in order fulfil the linearisable property. However, Passive replication provides linearizability, but then we need to handle the failure of the primary server (that talks to the bootstrap server) and implement leader election to replace the primary.

Right now we have not made a final descision on a replication algorithm, but we plan to use a group membership service for the replication groups and not a ring structure. We also plan to use passive replication.

Right now the replication degree we have chosen is 3. This may change in the future though. Every third node gets a key in the lookup table, with the other two getting the same key.

4 Broadcast

We implemented two broadcast components; a best effort broadcast component (BEB) and a reliable broadcast component (RB). Both components were written in Java, ported from the Scala versions we had previously implemented in the programming exercises.

The system will send broadcast requests to the RB component which then uses the BEB component to actually perform the broadcast. The RB component uses the Eager Reliable Broadcast algorithm to ensure reliability.

After the Overlay Manager has created the overlay it sends the topology (list of all nodes) to the BEB component. The BEB component then broadcasts messages to all nodes in the topology.

In order to change the set of nodes that we want to broadcast to we can give the BEB component a new list of nodes. At the moment this is an explicit message that has to be sent to the BEB component, but in the

future we may allow a new list of nodes to be attached to the broadcast request message that is sent to the RB component.

Currently there is no process/workflow in the system that uses the broadcast components. They will be used e.g. when we implement replication later on.

5 Failure detector

We use an Eventually Perfect Failure Detector component, written in Java, ported from the Scala version we had previously implemented in the programming exercises. The component implements the EPFD algorithm with increasing timeout and sequence numbers. The component provides a port and sends two kind of indication messages to it: message with the address of a process that it **suspects** has crashed or a message with the address of a process that is had previously suspected but is now being **restored**.

After the Overlay Manager has created the overlay, he sends the topology (list of all nodes) to the failure detector which then starts a timer and begins monitoring all the nodes. Each server requires the EPFD port and subscribes handlers with it to receive notifications when the component suspects or restores a node.

Right now this is as far as we have gotten with this, we can detect failure of nodes but we aren't handling it in any way.

6 Tests

In order to test the system we needed to preload some data in the system. We simply initialize the nodes with test data, where every node has the same data set, not taking into account partitioning of data. Later on, after the PUT operation has been implemented, a node will of course only have data elements that it is responsible for. We may also later find a way to partition the preloaded data among the nodes.

We created simulation scenarios for the EPFD component and the broadcast components as well as holding on to the Ops test from the template.

Note that in our tests we are not using the servers (i.e. the system itself), except for the OpsTest. We are simply creating a set of test clients that are written to test specific components. When we have gotten more functionality in our system we plan on writing tests that cover all the features of the system and how it works as a whole. At the moment the system does not use the EPFD and broadcast components, they have just been integrated and are ready for use.

We also want to find a way to avoid (or at least minimize) injecting any test specific logic into our system. Ideally, the testing code (such as writing test results in the `GlobalView` or `SimulationResultMap`) should be

contained within the test module, not within the system itself. This makes it challenging to test internal events in the system that are not visible to clients, such as broadcasts and failure detectors.

We have not yet figured out a good way to do this. For now we decided to write self-contained test clients to test the broadcast and failure detector components.

6.1 Operation GET tests

We simply modified the the given OpsTest to test GET operations that are expected to return code `OK` with the requested data element and others that are expected to return code `NOT_FOUND`.

6.2 Broadcast component tests

We wrote a test that broadcasts messages among a set of different test clients.

We have yet to test the reliable broadcast. In order to do this we need to exclude some nodes from the initial broadcast (simulating messages getting dropped) and verify that they will still receive the message.

6.3 Failure detector component tests

For the failure detector we need to test that it satisfies the strong completeness and eventually strong accuracy properties. To test the completeness property we create a few test client and each of them subscribe to the EPFD port. If the failure detector suspects a node has crashed the clients store information about it in a set of suspects. For the test we then need to kill one of the clients and then assert that all our other clients have the dead client in the list of suspects.

A similar scenario could be created to test the eventually strong accuracy propery. We could mabye let one of the clients sleep for a while and then wake it up and the assert that none of the clients eventually suspect the temporary sleepy client.