Distributed Systems, Advanced Course
Preliminary Report

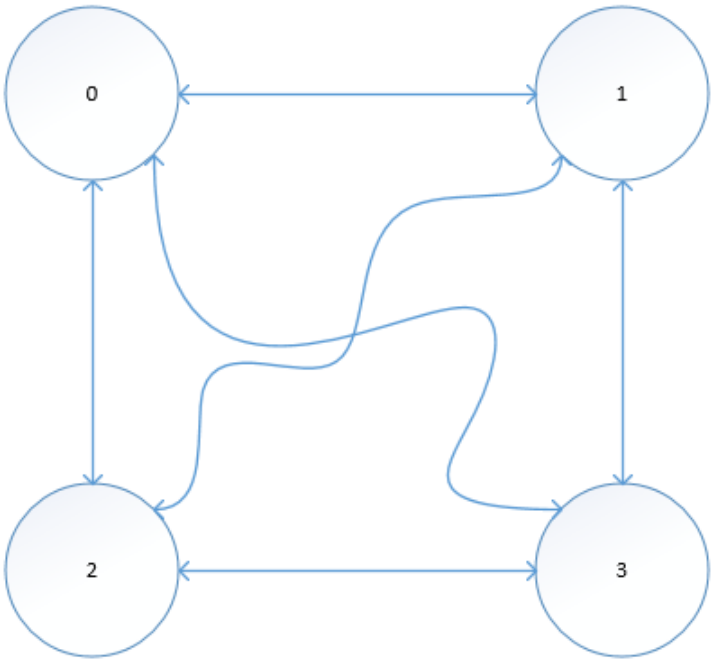
KTH Royal Institute of Technology
School of Information and Communication Technology
Student:Fanti Machmount Al Samisti (fmas@kth.se)
Student:Pradeep Perris (weherage@kth.se)

February 12, 2016

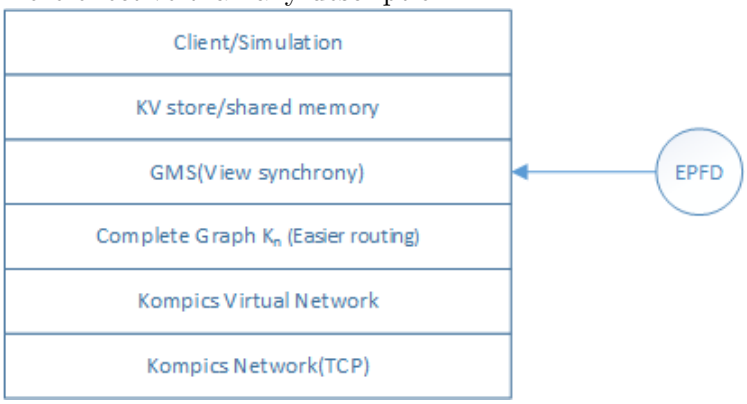
Contents

1 Introduction

The goal of this project is to implement a distributed key-value store in *Kompics*[1] with many freedoms given at hand like the structure, replication algorithm and factor and much more. We chose a full graph network e.g. if $\mathbf{n = 4}$ then K_4(in graph theory terms).



Before getting deeper into the architecture of our system, an overview picture is much more effective than any description:



Having the above in mind, we can move on to the insides of our system. A *Kompics* virtual network is comprised of *vnodes* that share the same physical traits, namely IP address and port number, and they are distinguished by an *id*.

2 Fellowship of the nodes

When then system is *bootstraping*, the single *NodeParent* component reads the following configuration file:

```
network {
    node {
        host = "127.0.0.1"
        port = 34567
    }
    grid {
        num = 3 # Size of the network, has to be at least 3
    }
}
```

Before getting into the message exchanges, it has to be noted that we use TCP to implement perfect point to point link in an asynchronous environment. This provides us with acknowledgments and message retransmission in the case of omissions. But when the destination is unresponsive TCP breaks the connection thinking it has crashed thus bringing synchronous assumptions into play and correct node suspicion.

Having done that its time to start up the nodes and create our group with the *Node* components. The first node is always the initial group *leader*. After him, all the subsequent nodes are *slaves* and they have to send a *JOIN* message to the leader to get accounted for and retrieve an updated view of the group contained in a *VIEW* message. Every node is tagged with a random number whose use will be explained in the next section.

A possible output of the initialization phase can be the following:

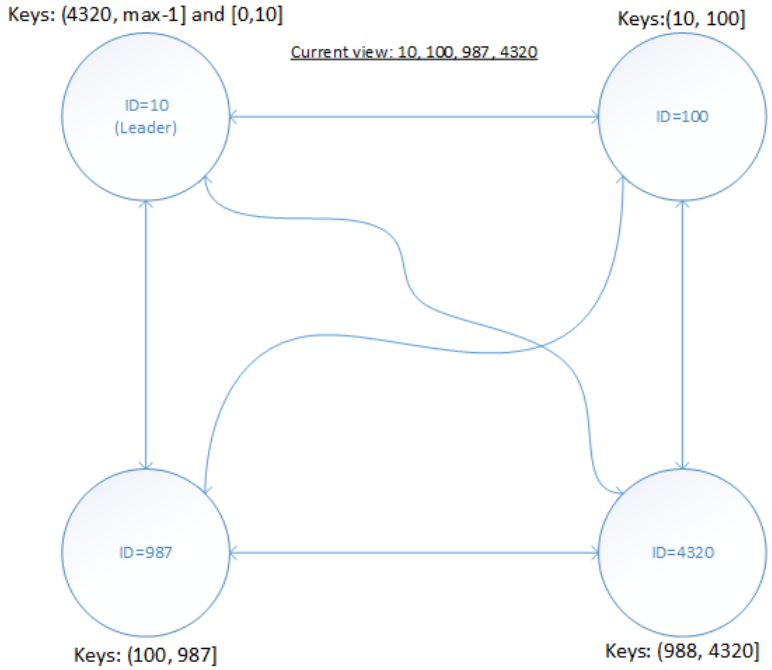
```
INFO  {Node} [0]: Got JOIN message from ID: [1]
INFO  {Node} [0]: Got JOIN message from ID: [2]
INFO  {Node} [1]: Got VIEW message from ID: [0](0 1)
INFO  {Node} [1]: Got VIEW message from ID: [0](0 1 2)
INFO  {Node} [2]: Got VIEW message from ID: [0](0 1 2)
```

If we move on to the formal side, the group should have the following properties[4][5]:

- FIFO and total message ordering
- Agreement: If a correct node delivers then all correct nodes deliver
- Linearizability

3 Key assignment

We pretty much followed *Chord's* key assignment[4] algorithm and applied it to our group. With a few nodes the key assignment will not be fair but as the size increases the distribution will be normalized.



The idea here is to assign an integer range of key values to a node starting from their *id* and moving backwards until the previous nodes' *id*, formally:

$$range = (id_{previous}, id_{current}]$$

To get the keys in the desired range we apply the following formula(murmur is a hash function):

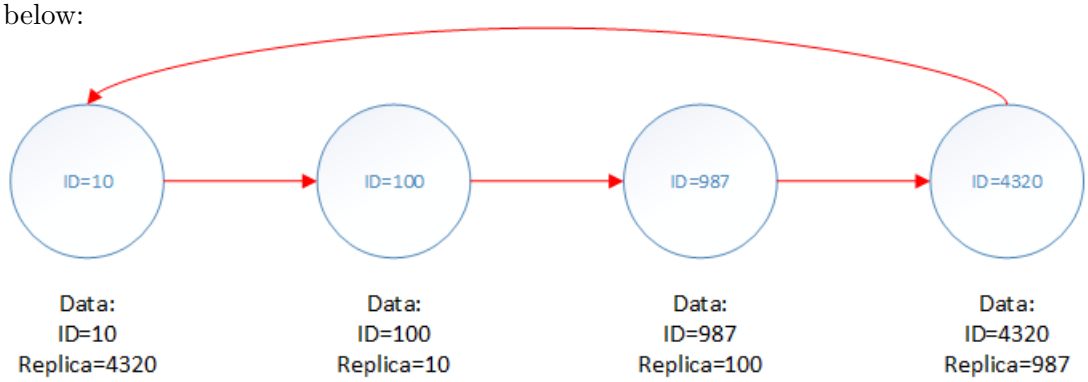
$$newkey = murmur3\_128(oldkey) \mod maxvalue$$

4 What's the value of XXX

This structure wouldn't be useful if we can't send queries of *GET*, *PUT* and *CAS*. Since this is the preliminary report we only have the *GET(key)* requests for preloaded values.
The *GET* request can be received by any node of the group. When this happens the node sends the request to the group leader, unless the leader is itself. As soon as the request arrives it gets broadcasted to the group and the ones interested in the packet respond to the client directly, aka owner node or a replica.

5 Copy here, copy there, copies everywhere!

For this system we chose to use forward replication with $\delta := 1$, as illustrated below:



6 Tests, tests

In our simulation we are testing(for now) the following scenarios:

- Graph creation
- Data replication
- GET(key)
- Non existing *key*
- Failed node detection(EPFD)

7 Who failed?

The group is monitored by an *eventually perfect failure detector* and it will function correctly as long as there's a leader and a replica to serve a node's data lives on some node. The algorithm is shown in the next page.

Algorithm 1 Increasing Timeout with sequence numbers

Implements:

EventuallyPerfectFailureDetector, **instance** $\Diamond\mathcal{P}$.

Uses:

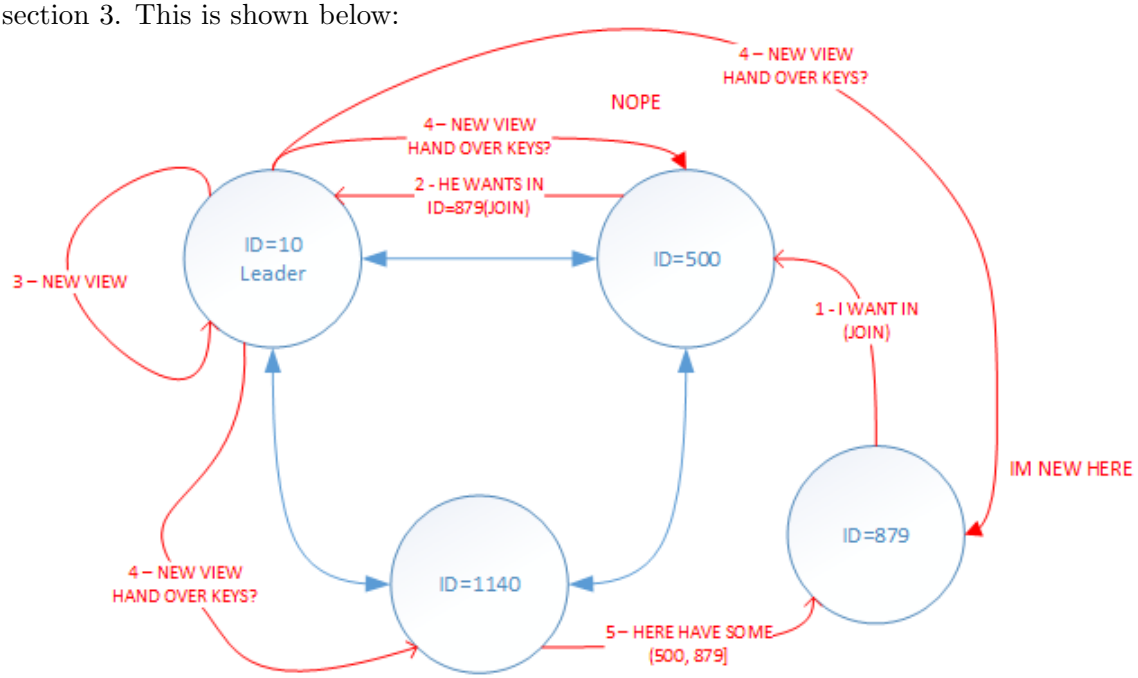
PerfectPointToPointLinks, **instance** *pp2p*.

```
1:  upon event ⟨ ◇P, Init ⟩ do
2:      seqnum := 0;
3:      alive := Π;
4:      suspected := ∅;
5:      delay := TimeDelay;
6:      startTimer(delay, CHECK);

7:  upon event ⟨ Timeout | CHECK ⟩ do
8:      if alive ∩ suspected ≠ ∅ then
9:          delay := delay + Δ;
10:     seqnum := seqnum + 1;
11:     for all p ∈ Π do
12:         if (p ∉ alive) ∧ (p ∉ suspected) then
13:             suspected := suspected ∪ {p};
14:             trigger ⟨ ◇P, Suspect | p ⟩;
15:         else if (p ∈ alive) ∧ (p ∈ suspected) then
16:             suspected := suspected \ {p};
17:             trigger ⟨ ◇P, Restore | p ⟩;
18:         trigger ⟨ pp2p, Send | p, [HEARTBEATREQUEST, seqnum] ⟩;
19:     alive := ∅;
20:     startTimer(delay, CHECK);

21: upon event ⟨ pp2p, Deliver | q, [HEARTBEATREQUEST, sn] ⟩ do
22:     trigger ⟨ pp2p, Send | q, [HEARTBEATREPLY, sn] ⟩;

23: upon event ⟨ pp2p, Deliver | p, [HEARTBEATREPLY, sn] ⟩ do
24:     if sn = seqnum ∨ p ∈ suspected then
25:         alive := alive ∪ {p};
```

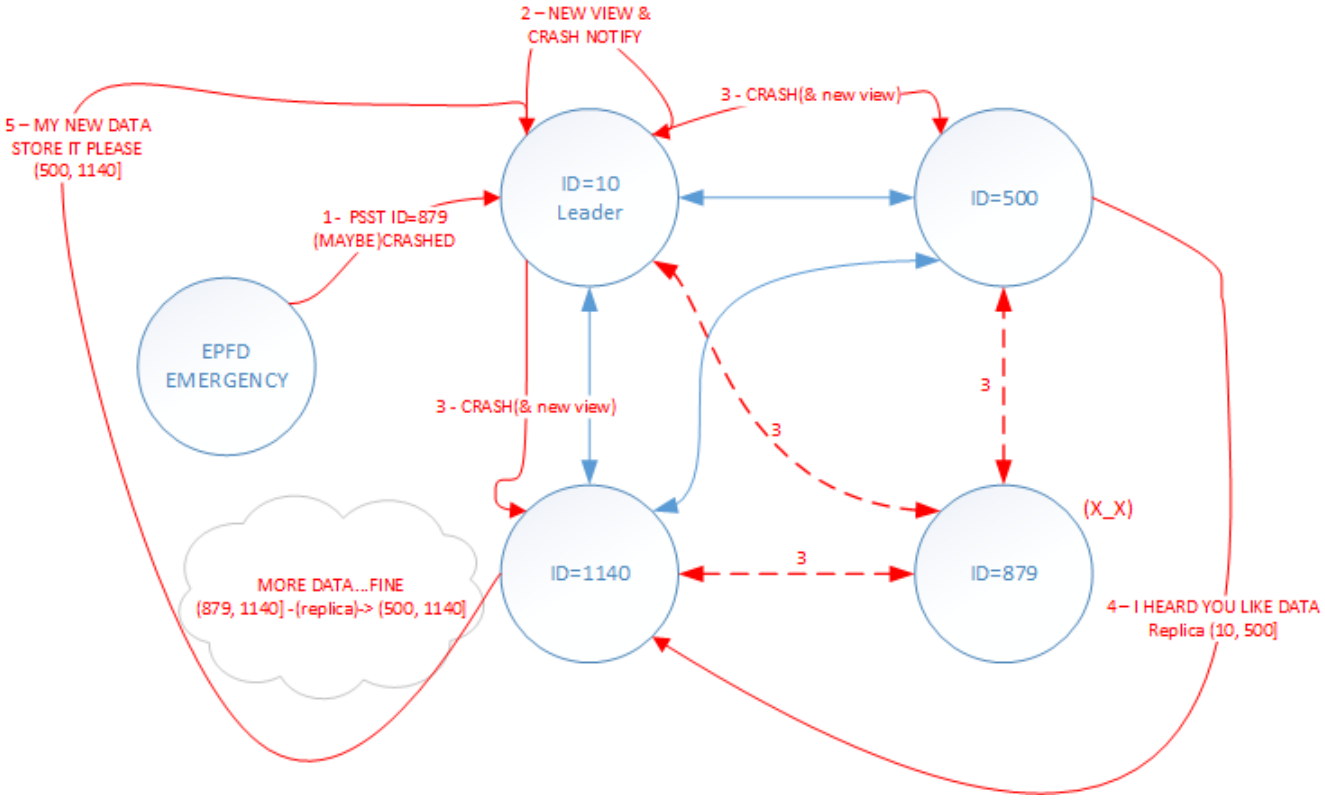
8 But...we should be flexible

maybe reconfiguration insight, replication handlinig, properties(linerizable)

In this section we give some insight on what some of our solutions are for new, leaving and crashed nodes.
In the same fashion as *Chord* when a **node joins** the network it should be handed over the keys that is responsible for based on its *id* as described in section 3. This is shown below:



Based on the same logic when a **node leaves or crashes** the network should rebalance and hand keys to the proper owners in this new configuration:



References

[1] SICS Swedish ICT and KTH Royal Institute of Technology. http://kompics.sics.se/. 2015

[2] Previous year assignments. *https://canvas.instructure.com/courses/990374*. 2015

[3] Abraham Silberschatz, Peter B. Galvin, Greg Gagne. *Operating System Concepts 8th Edition*. Chapter 16

[4] Johan Montelius. *https://people.kth.se/~johanmon/courses/id2201/seminars/chordy.pdf*.

[5] Johan Montelius. *https://people.kth.se/~johanmon/courses/id2201/seminars/groupy.pdf*.

[6] Álvaro Castro-Castilla. *http://blog.fourthbit.com/2015/04/12/building-a-distributed-fault-tolerant-key-value-store*