Distributed Systems, Advanced Course
Project 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)

March 7, 2016

Contents

1 Introduction **2**

2 Design Overview **2**
2.1 System Component 2
2.2 Node Topology 3

3 System Abstraction and Implementation **3**
3.1 Perfect Point to Point Link 4
3.2 Best Effort Broadcast 5
3.3 (N,N) Atomic Registry 5
3.4 Failure Detector 6
3.5 Reconfiguration 6

4 System Simulations and Scenarios **7**
4.1 Perfect Point to Point Link 7
4.2 Best Effort Broadcast 7
4.3 (N,N) Atomic Registry 7
4.4 Node 10

5 Conclusions **10**

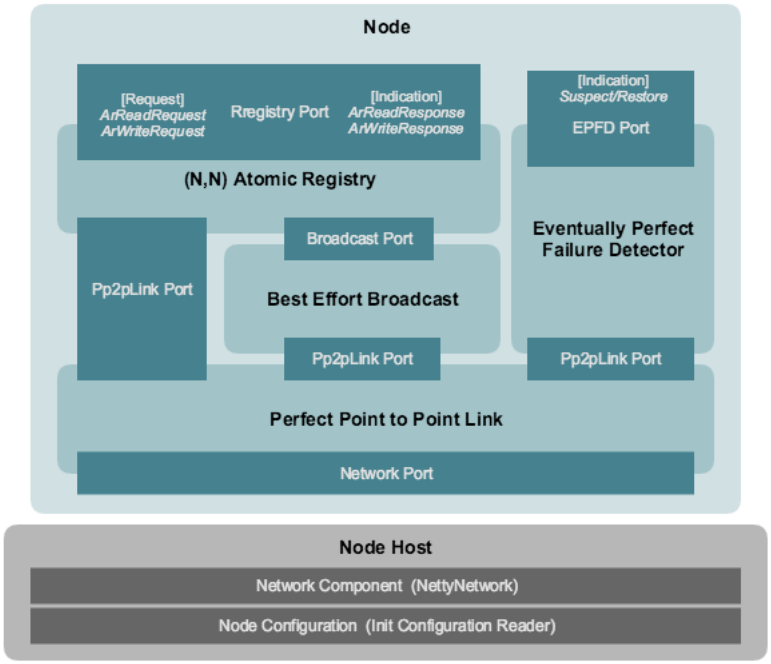
1 Introduction

The project's goal was to realize a key value store in *Kompics* framework. The most important properties are that it has to be distributed, replicated and comply to linearizable semantics. We were given many freedoms but we chose an easy and as little complicated as it can be implementation.
Our model is initially bootstrapped with a ,configurable yet, static set of nodes that comprise a group with a replication factor δ. Each replication group has a segment of the key space assigned to them and a hashing function is responsible for bringing an arbitrary key value between the boundaries.
Finally, we support *GET* and *PUT* operations on the stored data. They are protected from inconsistencies(linearizable) by employing a *Read-Impose Write-Consult Majority* algorithm to bring the *(N,N)Atomic Register* model into our system since we can't trust our *eventually perfect failure detector* so much. Essentially we have *fail-noisy* semantics but it boils down to *fail-silent* since we do not take into account the information from the *epfd*.

2 Design Overview

2.1 System Component

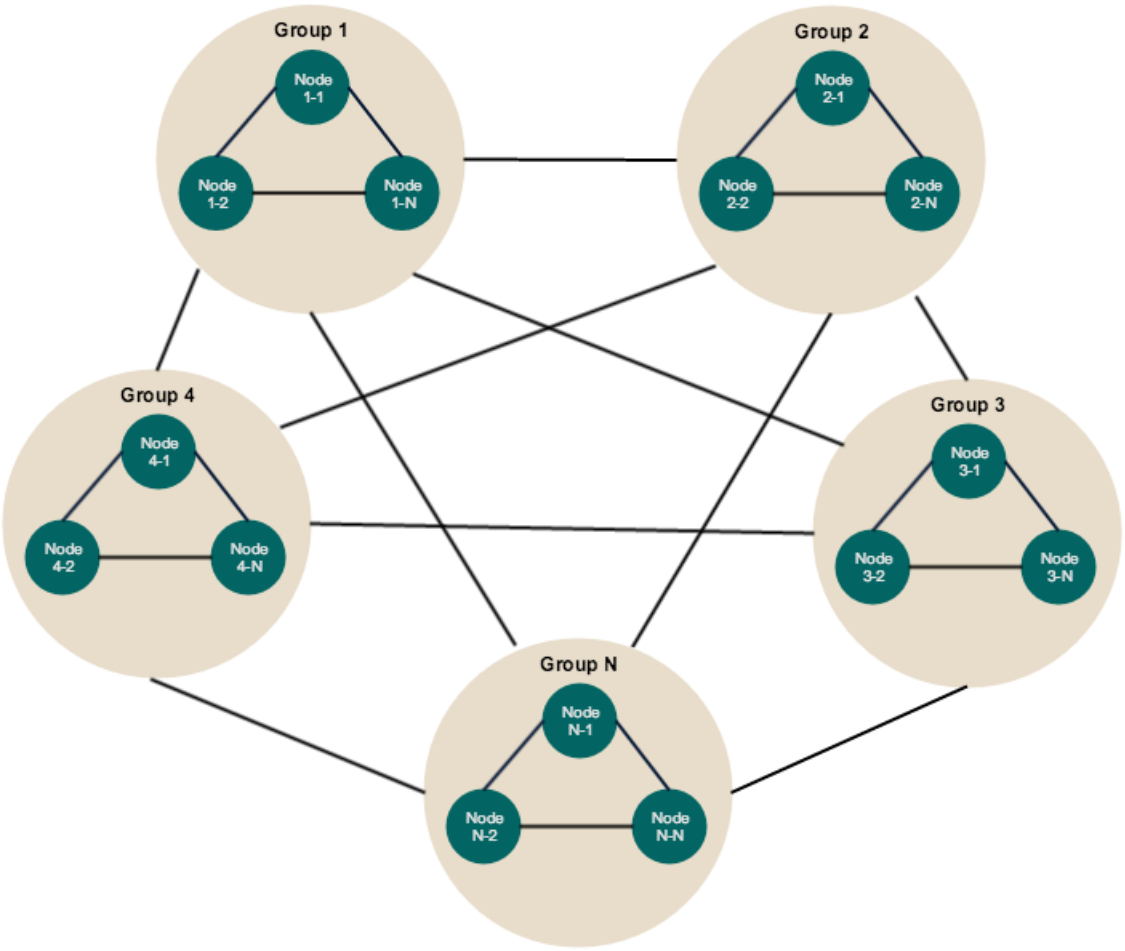
The following figure depicts the overall design of the system.



Node is the primary Distrbuted component in the system. It intercepts PUT, GET requests over Network component and directs them over (N,N)

Atomic Registry subcomponent. *Node Host* component is responsible for the initiation of Node instance according to deployment properties (Node ip/port, Group membership id etc.) given in node configurations file.

2.2 Node Topology



Within each replica group there is stored a subset of the key range. Assume that the maximum key value is x and there are n groups. This means that every replication group is responsible for $\frac{x}{n}$ keys. When a random node receives a *GET or PUT*, it forwards the message via *pp2p* to a random node of that replication group, since all this is a priori knowledge.

3 System Abstraction and Implementation

During our time designing and implementing the system we had a myriad of decisions to make and to motivate the 'why'.

- Networking protocol(TCP or UDP)
- Bootstrapping and mesh initialization
- Group membership(Static)
- Failure detector($\diamond$P)
- Routing protocol(fully connected mesh)
- Replication algorithm((N,N) Atomic Register)

Some properties have to be taken into account when deciding what algorithms to use and how to modify them according to our environment:

- *Network topology*: It plays a crucial role on the optimizations if its either LAN, WAN etc.
- *Flow control*: Practical systems ask for practical solutions for example not flooding the destination with messages.
- *Heterogenity*: Not all nodes are born equal, as they might be running on faster processors, have more memory etc.

3.1 Perfect Point to Point Link

All the node communication, e.g. message exchanging, is achieved by using a perfect point to point link abstraction. Since our system doesn't support any kind of reconfiguration we are working in the *crash-stop* model. Additionally, we make the assumption that no messages are lost over the network as the lower levels of the operating system can handle them e.g. when being sent over *TCP* by adding reliability and retransmission among others.

The choice of this protocol was clear as it can support some guarantees for delivery but it doesn't come without its drawbacks. The main one is that *TCP* makes assumptions about the state of the delivery buffer and combined with congestion control there might be cases of false positive crashes. Such assumptions add synchrony in an asynchronous system or in our case, partially synchronous. But as always its a matter of what algorithms the system uses and as such we have to use the appropriate abstractions.

Below we see the properties that our implementation has to abide by:

- **Reliable Delivery**: If a correct process p send a message m to correct process q then q eventually delivers m.(*liveness*)
- **No Duplication**: No message is delivered by a process more than once.(*safety*)
- **No Creation**: If some process q delivers a message m with sender p, then m was previously sent by process p to q.(*safety*)

3.2 Best Effort Broadcast

As pictured above our system nodes comprise one group with smaller ones to serve as replicas. The broadcasting happens only within the members of the smaller groups as sending a message to the rest of the replica groups is not useful in any shape or form. The chosen abstraction guarantees that all the correct processes will deliver as long as the sender is correct by using *pp2p*.

Our system does not handle crashes but only detects them. This fact led us to the use of *BEB* abstraction since there is no difference compared to another abstraction that can handle byzantine faults in our case. It is built on top of the *pp2p* component and for each destination it issues a <*pp2p—Send*> and expects <*pp2p—Deliver*>.

The properties that *BEB* offers and we have to verify are:

- **Validity**: If a correct process p broadcast a message m then all correct processes eventually delivers m.
- **No Duplication**, from *pp2p*: No message is delivered more than once.
- **No Creation**, from *pp2p*: If some process delivers a message m with sender p, then m was previously broadcast by process p to q.

3.3 (N,N) Atomic Registry

All the possible executions must be *linearizable*. The problem we had to solve is what will happen when more than one clients send a request(*GET or PUT*) to the stored data at a node level as well as at a group replication level. Out of the available registers(safe, regular, atomic), *atomic* is the only one able to provide linearization semantics.

The replica group contains a certain number of keys(configurable in our system) but the algorithm works for one register. Due to this, we modified it to work with the hash dictionary of keys and their values by piggybacking the contextual data key to all the messages. The need of a failure detector in this algorithm is non existent so there is no use of our $\diamond$P here.

Formally, a linearizable execution guarantees:

- Every read returns the last value written
- for any two operations x and y, if x precedes y in the execution then x also appears before y in the linearization.

The properties of *NNAR* are the following:

- *Atomicity*: Every execution of the register is linearizable. Every failed operation appears as if it has completed or never been invoked at all. An operation appears to have completed in an instant between its invocation and completion.

- *Termination*: If a correct process invokes an operation, then it eventually completes.

3.4 Failure Detector

Our working environment , as previously stated, is a partially synchronous system so the best way to incorporate timing assumptions is by using an *eventually perfect failure detector* or ⋄P. At this point we found ourselves at a fork, choose a small delay to quickly react to potential failures and in time adapt to the network or choose a higher value to *suspect* with a better degree of confidence but maybe detect the failure late.

It turns out that without implementing *reconfiguration* it does not really matter if we detect a crashed node as soon as possible or well after its death. In the case we were realizing it, the importance of the delay timeout and its initial value might be crucial depending on the application constraints.

The provided properties of this abstractions are:

- **Strong completeness**: Eventually, all the crashed processes will be suspected by every correct process. It is satisfied if we exclude the suspected process on timeout since it will stop responding to the *heartbeats*.

- **Eventual strong accuracy**: Eventually, no correct node is suspected by any correct process. After the point that the system becomes synchronous, eventually all suspected correct processes will be restored and never suspected again with the increased delay timeout.

It is clear that each node cares about only its replication group and consequently ⋄P sends heartbeats only to the replicas.

3.5 Reconfiguration

Despite not implementing any kind of crash tolerance and node addition, we decided to write down the design choices that could have gone into a theoretical system if we had, much, more time. As a reminder, the nodes are connected with a fully networked mesh(*group membership service*) and comprise a number of smaller replication groups each one responsible for a different key space.

Starting from the base abstractions, *pp2p* and *beb* remain the same as they are here only to help us communicate although having the *agreement* property in our broadcasting abstraction would be desirable. Without *agreement* we do not have any guarantees about the data consistency in the replicas making the use of *Reliable broadcast* necessary.

Stepping up the complexity, to implement the *CAS* we need atomic total order broadcast between the replication group since it comprises a two step

atomic operation. With this requirement in mind the need for a consensus algorithm rises, namely *Paxos*.

Summing up this section, for a dynamic group we need a group membership service with view synchrony along with an eventual leader election algorithm.

4 System Simulations and Scenarios

Simulation and testing of the system is carried out at each component level, starting from base of *Perfect Point to Point Link* up to *(N,N) Atomic Registry* components. There are separate Simulation Scenarios for each components,(see *test/simulation* in source code bundle), that covers properties of underlying distribution abstraction.

4.1 Perfect Point to Point Link

As it is explained in section 3.1, the properties of *Perfect Point to Point Link* to be verified are Reliability, No Duplication and No Creation.

The Reliable Delivery property is simulated in scenario *pp2p/Pp2pLinkScenario.java*. It let a process instance of *LinkPoint.java* to send a constant number of messages (1000) to another instance of *LinkPoint.java*, and *Pp2pSimulationObserver.java* listens on number of messages received at second instnce, and captures in a *GlobalView*. GlobalView terminate and report to Simulation at successful receival of all messages.

Also this can be even proven with the standard trasport-level protocol used in the Network Layer, which is TCP. In same way No Duplication and No creation of messages are also satisfied with properties of TCP.

4.2 Best Effort Broadcast

The properties of *Best Effort Broadcast* to be verified are Validity, No Duplication and No Creation.

The Validity property of the component is simulated and verified in *BebScenario.java*. Within *beb/BebScenario.java*, it creates a constant number of (1000) of *beb/BebPointHost.java* instances as the receivers and let another instance of *beb/BebPointHost.java* to broadcast a message. *BebSimulationObserver.java* simply verifies that all receivers are received with the broadcast message. No Duplication and No Creation properties are derived from the underlying *Perfect Point to Point Link* component.

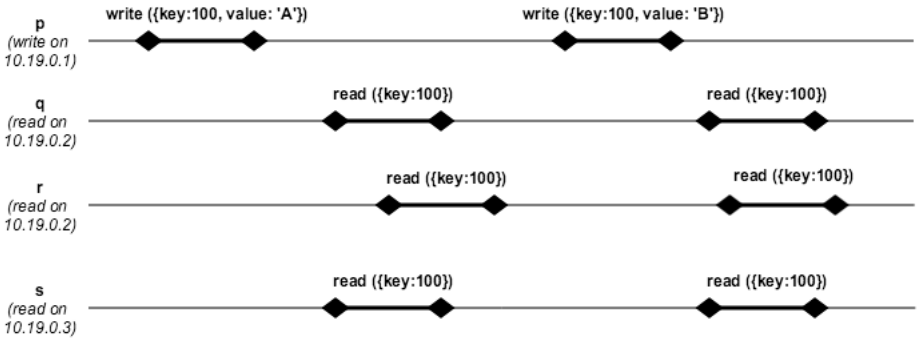
4.3 (N,N) Atomic Registry

As it is explained in section 3.3, the properties of *(N,N) Atomic Registry (NNAR)* are Atomicity and Termination. *NNAR* is a strict generalization

of *(1,N) Atomic Registry (ONAR)*, as the every execution of *ONAR* is also an execution of *NNAR*. Therefore, *Ordering* property of *ONAR* should be verified in *NNAR* with a single writer.

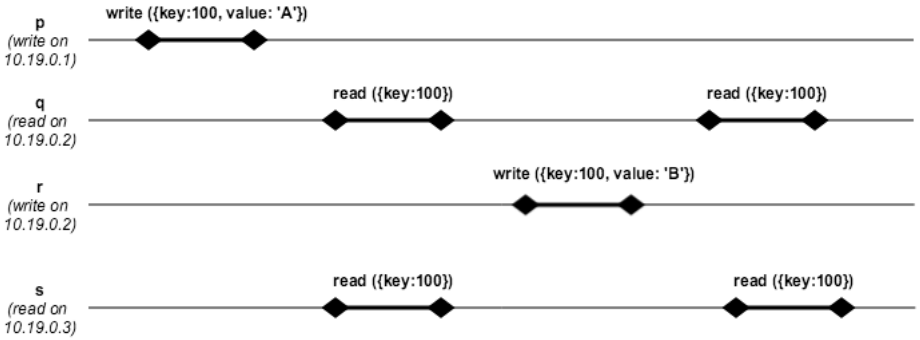
The simulation scenario in *register/SimpleReadWriteScenario.java* simply verify the termination of Read/Write operation. Also the simulation *register/WriteAfterReadAllScenario.java* further verify after a write operation, eventually all Reader processors received the written value each registry node.

The scenario *register/TwoWritesLastReadScenario.java* covers the following execution.



register/TwoWritesLastReadObserver.java verifies that last read always return the last written value. That is, it prevents old value being read, which covers *Ordering* property of *ONAR*.

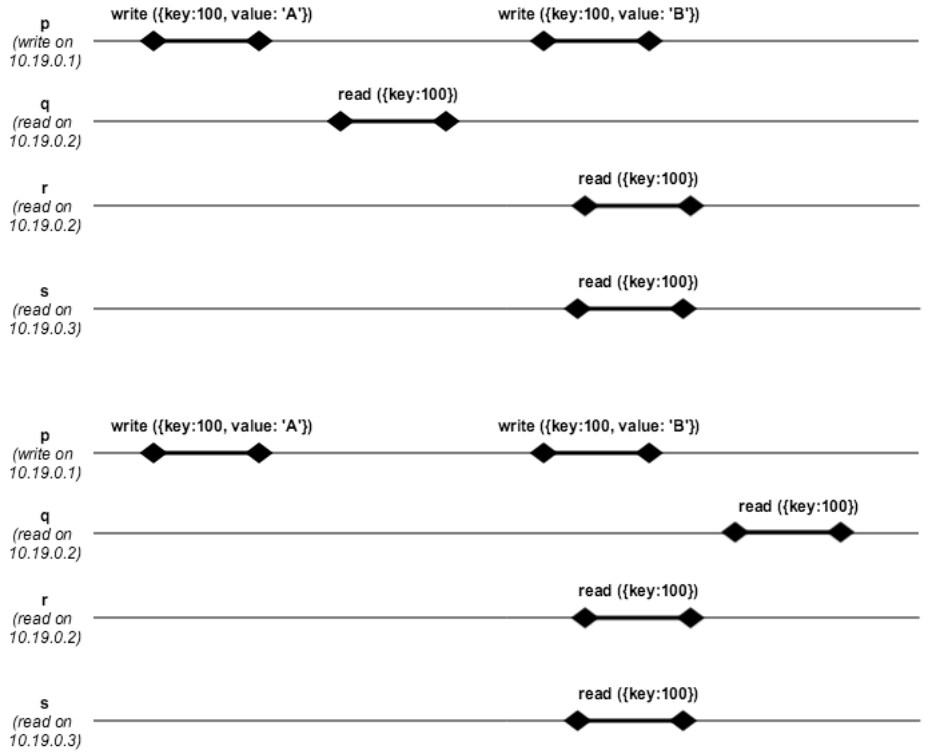
The scenario *register/LineariazableReadWriteScenario.java* covers the following execution. The linearizability is captured in *register/LineariazableReadWriteObserver.java* and verify the execution steps.



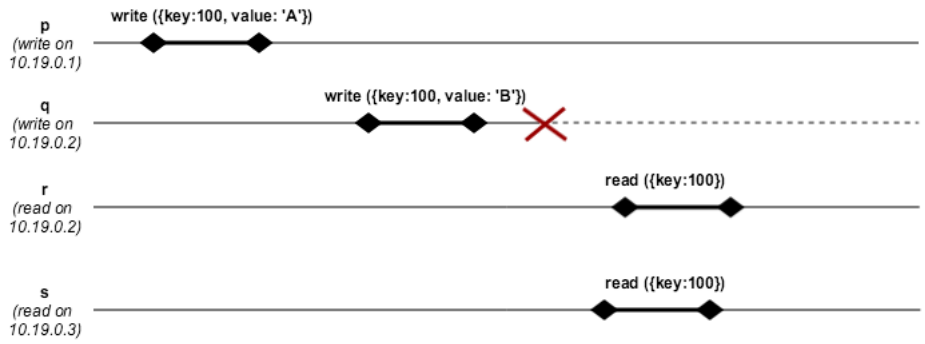
The Scenario Observer *register/LineariazableReadWriteObserver.java* simply list the execution steps that was captured in it, and we can assert it with given execution steps.

In *Regular Register*, Reads with concurrent Writes should return last

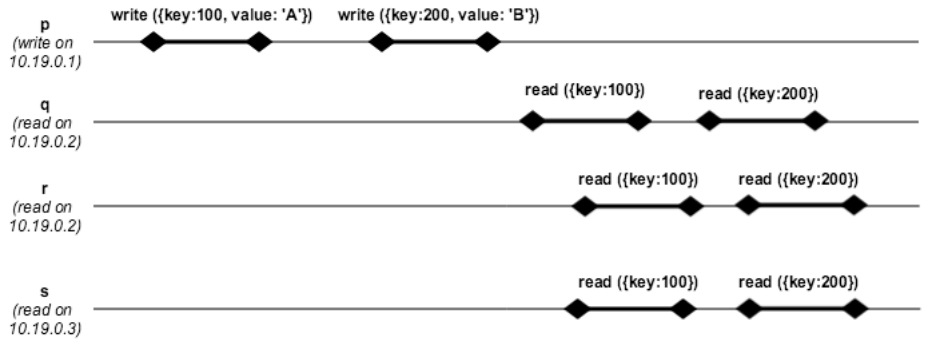
written value or value of the concurrent write. But when it comes to *ONAR and NNAR*, it should also comply with any precede Read value. It is verified in *register/ConcurrentReadWriteScenario.java*, which follows the following executions.



Failed writers in *NNAR* should be appeared as it was never invoked or completed. It was verified in *register/FailureWriterReadScenario.java*. The second writer process was killed after triggering PUT request. In the simulation scenario *register/FailureWriterReadObserver.java* aways found it as a completed Write in subsequent Readers.



NNAR in this project, is improved to work with value of hash dictionary. It allow Read/Write on multiple key values simultaneously, it should not block the Read or Write operation with different keys. This is verified in *register/MultiKeyReadWriteScenario.java*, for similar executions as the following.



4.4 Node

Node compromises it's distributed system properties of its subcomponents. Node topology and its main events, GET/PUT are verfied in the scenario *NodeStructureScenario.java* also with a set of hosted nodes and a client component. Static setup of Nodes for testing is hanlded within *config.Grid.java and config.ReplicationGroup.java*.

5 Conclusions

The open ended nature of the project was a blessing and a curse. The freedom we had helped us develop critical thinking on distributed systems as well as reasoning and insight on why something works(usually does not). On the contrary, due to this being our first real world project with an actual implementation, initially it felt like being dropped in the desert with the task to build a place to live.

After finding our bearings with *Kompics* and figuring out its quirks development went smooth with the only headache being how to test and debug the system. Last thing before moving to the meat of the summary is how straightforward it was to implement an algorithm from the book on 1 to 1 translation.

Developing a fully realized system, despite of it having a steep(for us) learning curve, taught us a ton of things. Building a static system with the a good degree of fault tolerance is quite straightforward. Things do not look so good when the switch to a dynamic group happens since exponentially more problems occur but in exchange you get much more useful system.

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*

[7] Distributed systems, Advanced Course(ID2203), Lecture notes 2016. *https://canvas.instructure.com/courses/990374/files*