ID2203 Project
Linearizable Key-Value Store

Kim Hammar, kimham@kth.se
Konstantin Sozinov, sozinov@kth.se

March 12, 2017

1 Introduction

This report contains a overview of the work done as part of a project assignment in an advanced distributed systems course. The given task was to design and implement a "simple" distributed key-value store with linearizable operation semantics. No particular constraints were given on *how* to implement the key-value as long as it fulfills the requirements for linearizability. Apart from the implementation, the task also included to reason about different architectures and motivating design choices as well as writing different verification scenarios to prove the correctness of the implementation based on desired properties of the store. The store is implemented in Java using Kompics framework[1]. The source code for the implementation can be found on GitHub[2].

2 Infrastructure

2.1 Assumptions

In all cases where there isn't a network partition we assume the lower level networking protocol TCP in combination with IP-routers in the network to provide a perfect-link abstraction as defined in the course-book[1]. All our network-based algorithms assume access to the perfect-link abstraction.

2.2 Overview

The store is divided into different partitions where each partition is responsible for a certain key-range and the partitions are distributed over the available nodes. Furthermore, each partition is replicated with a specific replication-degree δ. Each partition constitutes as a replication-group and is responsible for dealing with client requests and providing linearizable operation semantics. To connect the replication groups into the complete store, a simple overlay is used which supports routing based on lookups in a lookup-table. This lookup-table is generated through a bootstrap-protocol at startup and is static throughout the execution. Later in the report we will show how to extend it to support *reconfiguration*.

Our approach to implementing linearizable operation semantics in a reconfigurable storage is to use a combination of virtually synchronous group communication and passive replication. Figure 1 depicts a simplified architecture overview of the store implementation at each node. It is simplified in the sense that it only shows a single node, and the overlay layer for routing is omitted as well as low-level components like $\Diamond\mathcal{P}, \Omega$ and BEB.

[1] `http://kompics.sics.se/current/index.html`
[2] `https://github.com/konsoz/Distributed-KV-store`

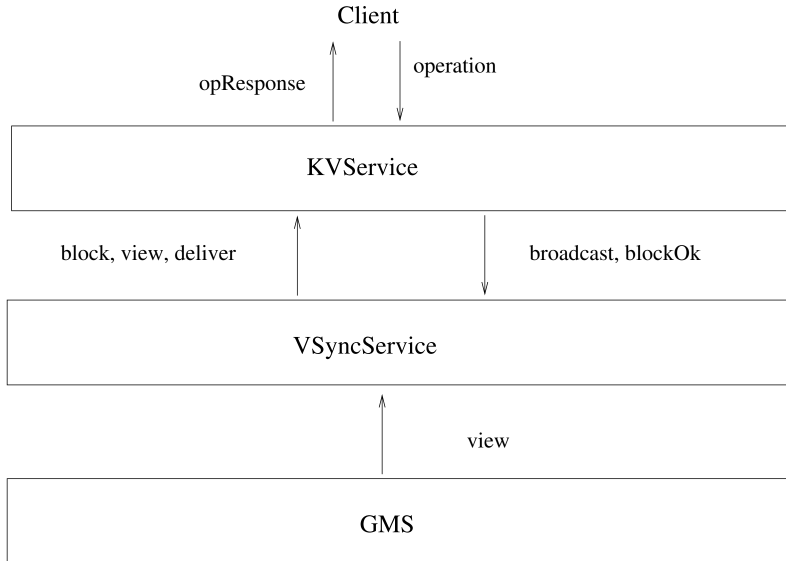

Figure 1: Simplified overview of the store architecture at each node.

View-synchrony on its own is not a replication model for linearizability but as we'll show in this report, using it with a simple passive replication scheme guarantees linearizability and still tolerating up to a minority of process crashes.

2.3 Motivation

2.3.1 View-synchronous group communication

Essentially what the view-synchrony layer provides in this implementation is a "virtual" synchrony in the sense that despite running on top of an asynchronous network and being in the fail-silent model[3] it provides a total-order delivery of messages and group-views to all "correct" processes. Since accurate failure-detectors are not accessible in the fail-silent model, the synchrony is *simulated* by declaring any process that is suspected by a non-accurate FD, whether correctly suspected or not, as faulty and removed from the view. The worse that can happen is that a process was falsely suspected, the process will then later realize that it got excluded from the view and abort itself. This way we can ensure very strong consistency properties on the processes that are in still in the view.

However as mentioned in the lectures, there is no free lunch! Our view-synchrony implementation is not partition-tolerant, to ensure strong consistency we have sacrificed availability. If there is a network partition and two different views are proposed concurrently, then only the view with a quorum of processes will be allowed to proceed, this also means that the system only has a resilience of $\frac{N}{2} - 1$.

The implementation of the *VSyncService* component (section 2.4.2) is based partly on the implementation in the course-book [1]. We have adapted the algorithm from the course-book to the fail-noisy model and used the paper *"Using Virtual Synchrony to Develop Efficient Fault Tolerant Distributed Shared Memories"*[2] as a guide for providing linearizable operation semantics on top of virtual synchronous group communication.

2.3.2 Passive replication

In a passive replication scheme a replication group of nodes will consist of one primary node and several backup nodes. All operations directed to a certain replication-group will be directed to the primary who then synchronously will collect acknowledgements from the backups before returning to the client. In the case of a PUT/CAS request this query phase of the primary is necessary to

[3] The algorithm we use for view-synchrony is safe in the fail-silent model with up to $\frac{N}{2} - 1$ failures, but to make our life easier and to ensure liveness, the implementation actually assumes the fail-noisy model and access to Ω and $\Diamond\mathcal{P}$. Specifically this makes a correct implementation of the group-membership service (GMS) a lot simpler (a weaker form of GMS that's possible in the fail-silent model would also "kind of" work but such an algorithm would be more complex and give weaker guarantees than ours).

ensure the replication degree since the ACK-request will include the latest update. When there is a read request one might think that the query phase is redundant but it is mandatory in our implementation to ensure linearizability since it's a way for the primary to make sure that it still is the leader and is trusted by at least a quorum of backups. If the query phase is omitted the primary's FD might be slow and not aware of the split which can cause it to return stale values and violate linearizability. The communication between primary and backups will be done through broadcasts in the view-synchronous communication layer and utilize the guarantees it provides. As long as the partition (replication group) is available, there will always be a primary view where every node has agreed upon the view as well as the leader. When requests arrive at a a replication group it will route it locally within the group to the leader who will handle the request.

A system following the replication scheme just described is linearizable since the primary will serialize all operations, it also guarantees to always maintain the replication degree δ for all successful operations since it's a *waiting*-algorithm that waits for acknowledgements from at least δ backups. In order to provide linearizability in case of failures it is necessary that if a primary fails, the remaining backups will coordinate them self to continue where the primary left off. One of the conveniences in using a layered architecture and running replication on top of view-synchrony is that this agreement property is already given by the VSyncService (more about the practical aspects of this in section 2.4.2).

If the primary fails this will be detected (not necessarily accurately) and the surviving backups will elect a new leader that will install a new view, but before they install the new view they will make sure that every process that will live through the new view has the latest update. If a regular backup node would fail this would also eventually be detected and the node would simply be removed from the view by the primary, again this detection does not have to be accurate necessarily. As long as there still is a majority of nodes in the view no further action is needed.

2.4 Kompics Components

2.4.1 Relation between components

Figure 2 shows the micro perspective of a single node. A node constitutes of a hierarchy of connected Kompics components that communicate through events. What is not shown in the perspective is that the VSOverlayManager component communicates with other nodes for global routing, that the KVService component communicates with other nodes within its replication group for local routing, that the BEB component does broadcasts to other nodes within group views and that $\Diamond\mathcal{P}$ monitors a set of nodes through heartbeats.

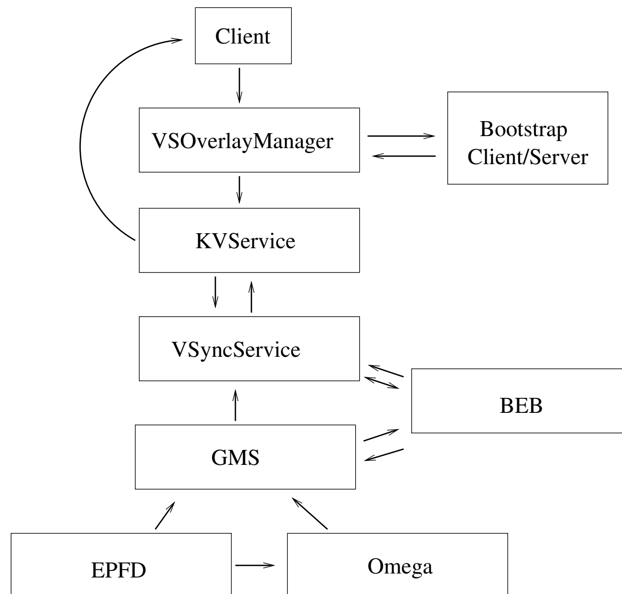

Figure 2: Kompics component hierarchy at each node.

2.4.2 Components

Eventual Perfect Failure Detector ($\Diamond\mathcal{P}$)
Our implementation assumes the fail-noisy model and access to $\Diamond\mathcal{P}$, the failure detector is input to GMS for deciding on the current group-view as well as input to Ω to elect a group leader. The implementation is taken from the course book[1].

Eventual Leader Detector (Ω)
The passive-replication scheme uses a leader which will be elected by Ω. Ω is implemented by using $\Diamond\mathcal{P}$, the implementation is taken from the course book[1].

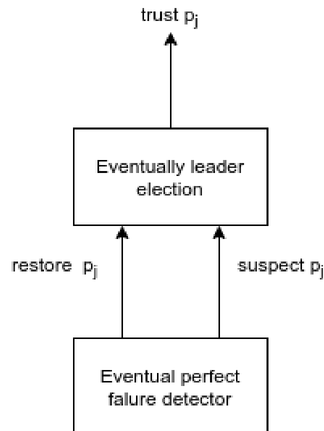

Figure 3: Diagram for eventual leader election component.

Best Effort Broadcast (BEB)
BEB is used for broadcasting within replication groups, the implementation is taken from the course book [1].

Bootstrap Client/Server (boot)
Short-lived components with the purpose of performing a simple protocol for bootstrapping the cluster of nodes, it is based on the template code. It assign nodes to partitions and generates an initial lookup-table. Minimum and maximum size of partitions as well as size of the cluster can be set up with configuration parameters.

Routing Overlay (VSOverlayManager)
The overlay layer manages the routing between different partitions and between partition and client. Right now the overlay is very simple, it simply uses the lookup-table generated by the bootstrapping and routes messages based on lookups.

```
partition-key --> [ip:port, ip:port, ...]
partition-key --> [ip:port, ip:port, ...]
```

Key-ranges are divided as follows among the partitions: Assume there are two partitions with keys k_1, k_2 where $k_2 > k_1$, then partition-k_1 is responsible for keys in the range $[k_1, k_2)$ and partition-k_2 is responsible for keys in the range $[k_2, \infty)$

For now the lookup-table is completely static and does not handle any churn at all. We will explain how this can be extended for an environment where churn happens in section 4 about reconfiguration.

Key Value Store (KVService)
KVService is a component that receives operation invocations from clients and delivers operation responses. The component performs routing within replication groups based on the current view from the vsync layer. If the KVService is the leader and receives a operation request, the updates to the store will be broadcasted and acknowledged through the VSync layer before replying to the client. If the KVService is not the leader it will forward the request to the leader who will handle it and respond to the client.

Virtual Synchronous Service (VSyncService)

The VSyncService component provides the KVService component with a layer to propagate write updates as well as read acknowledgements that is guaranteed to be delivered to all "correct" members of the current view of the replication-group. It also guarantees that *(i)* there will only exist one view that is available *(ii)* the state among all replicas in the view will be consistent.

These properties are trivial to achieve without failures. When failure happens it is necessary that when a process is detected to have crashed it is removed from the view and a new view is installed, but before this view is installed every member of the new view have agreed upon the current state. *(i)* is ensured by GMS which will only allow a primary to proceed and tolerates only $\frac{N}{2} - 1$ failures. Mechanisms to achieve *(ii)* is the core of the view-synchrony algorithm.

When the single new group-view is delivered from GMS, every member of the group will send a *block* request to the upper KVService component telling it to buffer incoming operations for a while until the new view is installed, once this have been confirmed by the KVService component the VSyncService begins a flush protocol where all group members of the new view to be installed will coordinate on which messages were delivered in the previous view, see figure 4. For simplicity in our implementation the whole state is sent with any update thus it's only necessary to agree on the latest update, not every single message delivered in the view. This is done by the leader of the view initiating the flush protocol and then selecting the most recent update. Once the flush protocol is finished the leader will send a view-install message to every replica and finally all correct members will deliver the new view to the upper layer and revoke the block. Observe that regarding the linearizability property it actually does not matter if the backups consolidate on the latest failed update or the update before that, as long as they have the same state. In the figure below they agree on the failed update (client did not get a response).

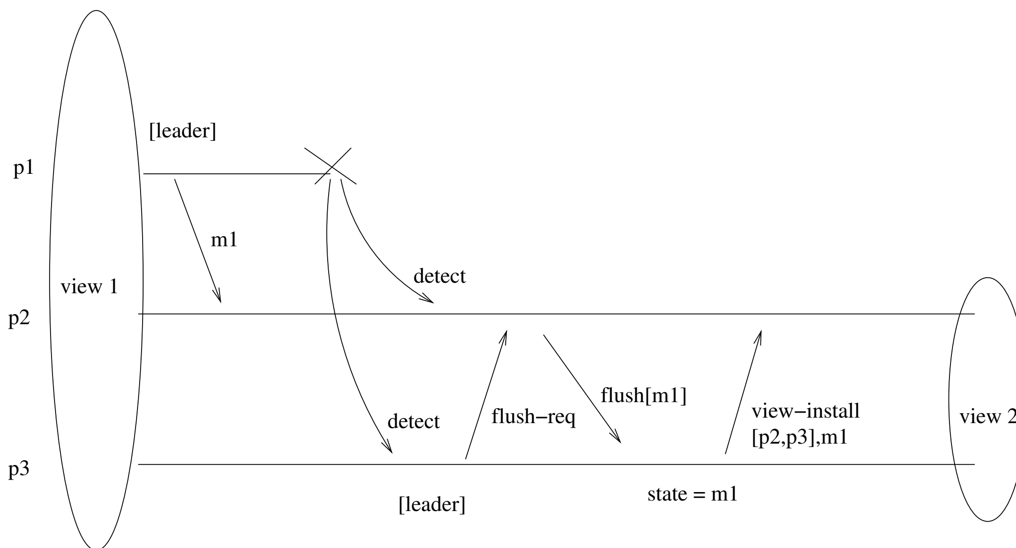

Figure 4: Time-space diagram over the flush-protocol

When receiving a broadcast request for some operation the leader of the view will synchronously perform a "safe broadcast" (broadcast + ack) with the update to all backups using BEB, before delivering to the client. It should be clear now that both the flush protocol and the update-protocol might wait for acknowledgements from crashed processes and re-send forever. For liveness the VSyncService relies on the guarantees of GMS. If a process in the view is not responding, GMS will eventually deliver a new view as long as there is a quorum of processes alive, when a new view is delivered the pending protocols in view-synchrony layer will be aborted.

Group Membership Service (GMS)

The group-membership service (GMS) is a core component that the virtual synchronous layer requires, the GMS need to ensure that processes agree on a single view and also that when a crash happens, eventually a new view that excludes the crashed process is installed. Our implementation of GMS assumes the fail-noisy model and uses the Eventual Leader Election abstraction (Ω) and the Eventually Perfect Failure Detector abstraction ($\Diamond\mathcal{P}$). Essentially the GMS in our implementation

works as a consensus abstraction, it ensures that all correct processes agree on a single view, and can tolerate up to $\frac{N}{2} - 1$ failures.

The implementation uses a coordinator-based 2-phase algorithm. From the bootstrap procedure there is a given initial view which every process agrees upon, then there are two failure-cases that need to be considered (i) coordinator crashes (ii) non-coordinator crashes. In both of these cases it is necessary that the remaining processes decide on the next view and the new leader consistently, which essentially is a consensus problem. As we touched upon earlier, a crash in this context does not need to be accurately detected, as long as all processes in the primary partition decide on the same view. In fact from the point-of-view of GMS a suspicion from the inaccurate FD is treated the same as a real crash, as long as the FD is accurate with a reasonable probability this is still a practical approach (if the FD is terrible inaccurate this also means that the GMS in some executions tolerate *less* than $\frac{N}{2} - 1$ failures, however safety is always assured).

The oldest group member will be considered the leader of the group and will decide on view-changes, but for the leader to manage to get through with a new view it needs the trust of at least a quorum of members. Ω ensures liveness here. As long as there exists a quorum of correct processes, eventually, the correct processes will agree on the same leader and that leader will then be able to install the new view. Safety (agreement) follows from the quorum assumption and that a process will only trust a single leader at a time and not accept views from previously trusted leaders. Processes that are suspected to be faulty will be excluded and eventually forced to crash, views are uniquely identified by sequence numbers that are monotonically increasing.

The two failure cases mentioned are dealt with as follows. In *(i)* the crash is eventually detected by all correct members through $\Diamond\mathcal{P}$ and a new leader is elected through Ω. Multiple leaders might be elected concurrently but only the leader elected by a quorum will be allowed to proceed and install the new view. For *(ii)* the leader will eventually detect the crashed member with $\Diamond\mathcal{P}$, remove it and install a new view. In section 4 we'll show how to extend the GMS to support reconfiguration, i.e adding new nodes.

3 KV-Store

Every operation, i.e GET/PUT/CAS is carried out in similar fashion. The operation will be routed to the leader of the responsible partition who then synchronously relays the update to the backups and collect acknowledgements before returning to the client. The CAS operation is the most complex one, it requires a kind of "locking" to implement, but with the primary-backup replication scheme and view-synchrony its not to difficult to implement. The primary can do the check of the reference value locally and then broadcast the update or read request to the VSync layer before returning to the client. The broadcast and collection of acknowledgement ensures that the primary really is the leader and then it can safely perform the update if necessary and return the old value to the client.

4 Reconfiguration

The view-synchronous group communication in combination with GMS already reconfigures the local views of each partition when nodes are suspected. Things that was added to support full reconfiguration:

1. *GMS to support join operation*: Join requests are forwarded to the leader of the group who will attempt to install a new view with the added member, the view is only installed if a quorum of members acknowledge the new view. Each member of a new view also need to update their $\Diamond\mathcal{P}$ and Ω to monitor the new node. For simplicity (certainly not efficiency) a state-transfer is done upon every view-install, even if there was no joins. This ensures that joined nodes will receive the whole state of the partition as soon as they install their first view.

2. *Gossip protocol between partitions*: The leader of each partition will periodically gossip the view of its replication group to every other node in its global lookup-table. Each node will maintain a list of the latest view id it has seen from each partition (reseted when a partition crash is detected) and when it receives a gossip with a newer view id then previously seen from that partition it will update its lookup-table as well as relay this gossip to every node it knows

in its lookup-table. This way all partitions should eventually get a consistent lookup-table if churn does not happen constantly.

3. *Failure detection between partitions*: Each partition will monitor the successor partition and if less than δ nodes are alive in that partition the leader will gossip that the partition has crashed and should be removed from the global lookup-table, the partition will then reconfigure its $\Diamond\mathcal{P}$ to monitor the next partition according to its view of alive partitions.

4. *Creation of new partitions*: If some partition is not "full" (δ is the minimum size of a partition and the maximum size is $\delta \cdot 2 - 1$) join requests will be forwarded to that partition. If all partitions are full then the join request will be forwarded to the partition with the highest key. The leader of that partition will then add the node requesting to join to a set of pending joins and inform the node that its request is pending. As soon as there is $\geq \delta$ pending joins the leader will create a new partition by assigning a key and handover all items in its store which is no longer in its key range to the new partition and broadcast a boot message to that partition. So basically when a new partition is created dynamically, there will be an existing partition that "helps" the new partition by supplying it with an initial lookup-table and handing over keys if necessary. When receiving a boot-message the new partition will boot like normal and elect its leader who eventually will gossip the view of the new partition to the rest.

5. *Routing with inconsistent lookup-tables*: Even if a node that receives the initial operation request from the client does not yet know about the partition that is responsible for the key, it will know that itself is not responsible for it and it will route the request to the closest partition it knows about. Given the way new partitions are created dynamically (created by the predecessor), it's very likely that eventually the request will arrive at the predecessor who even if the new partition hasn't completely booted and start gossiping its existence, knows about the partition and can route the message correctly[4].

Linearizability within partitions is always safe in presence of both joins and leaves/crashes since it will always require a quorum to install a new view. However there is extreme cases where linearizability in the global scope can be violated with our current reconfiguration approach. It might be that there is a network partition such that two different partitions are created with the same key concurrently and both partitions accept requests from clients, then the stores of the two partitions might diverge and linearizability will be violated. Our implementation does not deal with this case except that if a partition receives gossip that another partition with the same key exists it will abort itself.

[4]If the new partition hasn't yet installed its first view the request will end up in a queue of pending operations since the KVService is *blocked* until the view is installed

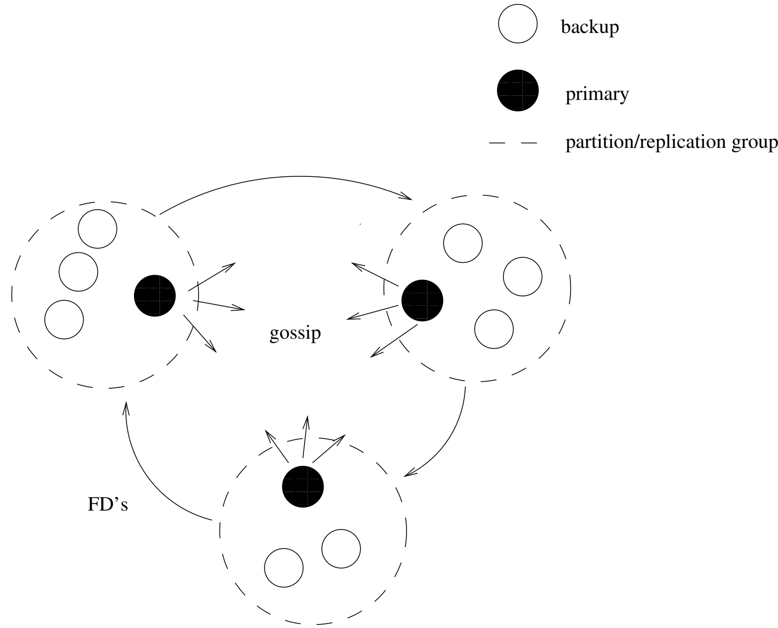

Figure 5: Sketch of how lookup-tables are maintained when churn happens. Each replication-group is running view-synchronous group communication to maintain the local view of its group and will gossip this view to the other partitions.

5 Simulation and Evaluation

Simulations are done through Kompics simulation framework. Different properties are tested and verified by simulating a designated *scenario* related to the specific property and then after termination verifying the correctness by asserting properties based on the logs/results of the simulation. The simulation scenarios can give quite good indication whether the implementation is correct or not since it simulates things such as network delay and timeouts.

5.1 Operation - Test

The fundamental functionality of the store to support GET/PUT/CAS operations is tested with a scenario where a cluster of servers is started as well as a test client that issues first a set of PUT operations and then a corresponding set of GET operations. Further more the client will issue two sets of CAS-operations which enables to verify the returned values based on reference values of the previous CAS-requests. All operations and responses by the client are logged to a storage and upon termination we verify that all operation succeeded and that the responses were as expected.

5.2 View - Test

This test verifies that within one replication group, nodes gets a correct view when one (or more) nodes crashes. In order to get the correct view nodes have to use all the basic components like BEB, Ω, $\Diamond\mathcal{P}$ and GMS. I.e this test indirectly also tests that the listed components which the view-synchronous service relies on, is functioning.

The test itself is a simple scenario where the servers are booted in a cluster and a single replication group. Then after an amount of simulation time passes, N of the servers are forced to crash. The test verifies that after boot-up an initial view was installed by all servers and after the crashes those servers who survived should have obtained a new correct view which excludes the crashed servers. Furthermore, all crashed servers should not get the second view since they are excluded from the group communication.

5.3 Replication - Test

The replication requirement is verified by running a simulation scenario where a cluster of servers are started with a specific replication degree δ and a few test clients issuing operation requests to

the cluster. Upon termination of the simulation the replication degree is verified by comparing the storage of servers in the same replication group and verify that all servers are in a consistent state. To verify that the replication is functioning it is also necessary to ensure that all completed operations are in fact replicated. This is tested by maintaining a trace of all operations during the simulation and then upon termination a copy of the store is built by applying updates of all completed operations in order. Finally the test verifies that all replicas are in the same state and that the state is equal to the one constructed from the trace.

5.4 Linearizability - Test

Linearizability of the algorithm that our replication is based on is proved in [2]. However the proof only consider GET and PUT operations and not the CAS operation. We will not attempt to prove linearizability formally but rather we use tests to demonstrate that the store provides the linearizability property in the common case, which is considered enough for the scope of this project.

A proof of linearizability for our store would rely on the properties of VSyncService which in turn relies on the properties of GMS. Given the agreement property of VSyncService and that all operations are serialized through a single leader who performs the updates sequentially, the operations will be executed in a sequential order which preserves real-time order and each read-operation will return the latest value written. Assume by contradiction that a operation does not return the latest value written, then there must be two concurrent group-views with different leaders, which is a contradiction to the quorum assumption in conjunction with the agreement property of VSyncLayer which says that all correct processes deliver the same sequence of views and the same set of messages in any view.

The challenge in demonstrating linearizability through tests is that the tests might show that a number of different type of executions all satisfy linearizability but it might still exist some very rare special execution that violates the property. Right now we are content with showing through tests that the implementation "kinda works".

The test is implemented as follows, a scenario where a set of servers are started in a cluster and a set of *sequential* clients which sends a set of random operation-requests is simulated. During the simulation the servers will write operation invocation events as soon as an operation request is received and write operation response events just before returning to the client to a concurrent FIFO queue (Java `ConcurrentLinkedQueue`). Upon termination we apply the Wing & Gong linearizability algorithm [4] as presented here [3]. Which basically is an algorithm where we go through the history of operation events and try to find a linearization of the operations. We also have a test where a bunch of clients is sending operation-requests in parallel when crashes happen to verify that the store preserves linearizability even when crashes happen.

5.5 Reconfiguration - Test

5.5.1 Add nodes to a single partition

This test will check if nodes get a correct view once new nodes joins one partition. Basically this test is similar to view test but instead of testing that the view is updated once crashes occur this test ensures that views are updated when new nodes join a partition and that joined servers receive the correct state. Since the partition will be unavailable during view propagation (the block protocol) the whole system will still be consistent and linearizable.

The test scenario will boot up N servers in a cluster and a single replication group, wait for a while, then boot up J amount of joined servers which will send requests to join one of the servers in the cluster. Then the test verifies that the initial cluster got two views, one where the joins are added. The test also verifies that the group of joined servers got the correct state through a state transfer.

5.5.2 Partition crash

This test tests that if a partition crashes this is detected by the other partitions who update their global view (lookup-tables). This is a very basic test to verify that the gossip protocol and failure detection between partitions is working.

5.5.3 Create new partition

This test ensures two features of the system: *(i)* that new partitions can be created dynamically upon join request when the existing partitions are full and *(ii)* ensure that keys are handed over correctly.

A simple scenario is applied in order to test the desired features, the scenario will boot up $N = 3$ servers in a cluster and a single replication group with replication degree $\delta = 2$. The scenario will also boot up a client which will send 2 PUT requests, one with $key = 49$, which corresponds to $HASH(1)$ and one with $key = 1572$, which corresponds to $HASH(15)$. Then the scenario will boot up $J = 2$ servers which will request to join. Since $N = 3$ and $QUORUM = 2$, there is not space to join the existing partition but since there are two join-requests they can form a new partition which satisfies the replication degree. The existing partition will help the two joining servers to boot up by handing over keys and lookup-table. In this scenario the key-range is 50 so the first partition is responsible for keys $0 - 49$ and the new partition is responsible for keys $50 - \infty$. This means that the first partition should hand over the key 1572 to the new partition, which is verified by the test.

6 Limitations and future work

A few obvious improvements that would not be to difficult to extend the implementation with:

- Don't send the whole state transfer when not necessary. In the normal case it should suffice to just send the latest update. Of course depending on the determinism of the update as well.
- Distribution of keys, right now the distribution of keys is not particularly even, a simple improvement would be to apply modulo operation to the hash to distribute the keys among the partitions.
- In the implementation as-is now, nodes that are once suspected and excluded from the view of a replication group will be forced to crash and re-join with a new join request. A more efficient way would be to simply let the node re-join if the $\Diamond\mathcal{P}$ restores the suspicion.
- Right now partitions that are under-replicated will suicide, a more efficient solution would be to make under-replicated partitions unavailable until new nodes join.

And a improvement that would require some real work: Earlier we mentioned that there is a special case in which our reconfiguration-approach in the global scope might cause two new partitions with the same key to be created concurrently and violate linearizability if there's a network partition. This could perhaps be solved in a similar manner as we handle the network-partition inside the replication-groups, i.e that to do any changes to the view it requires acknowledgements from a quorum of processes in the view (which in this case would mean the global view, not the view of a single replication-group) and if that cannot be collected, it becomes unavailable.

References

[1] Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer Publishing Company, Incorporated, 2nd edition, 2011.

[2] Roy Friedman. Using virtual synchrony to develop efficient fault tolerant distributed shared memories, 1995.

[3] Garin Lowe. Testing for linearizability, 2010.

[4] Jeanette M. Wing and Chun Gong. Testing and verifying concurrent objects. *Journal of Parallel and Distributed Computing*, 17(1):164–182, January 1993.