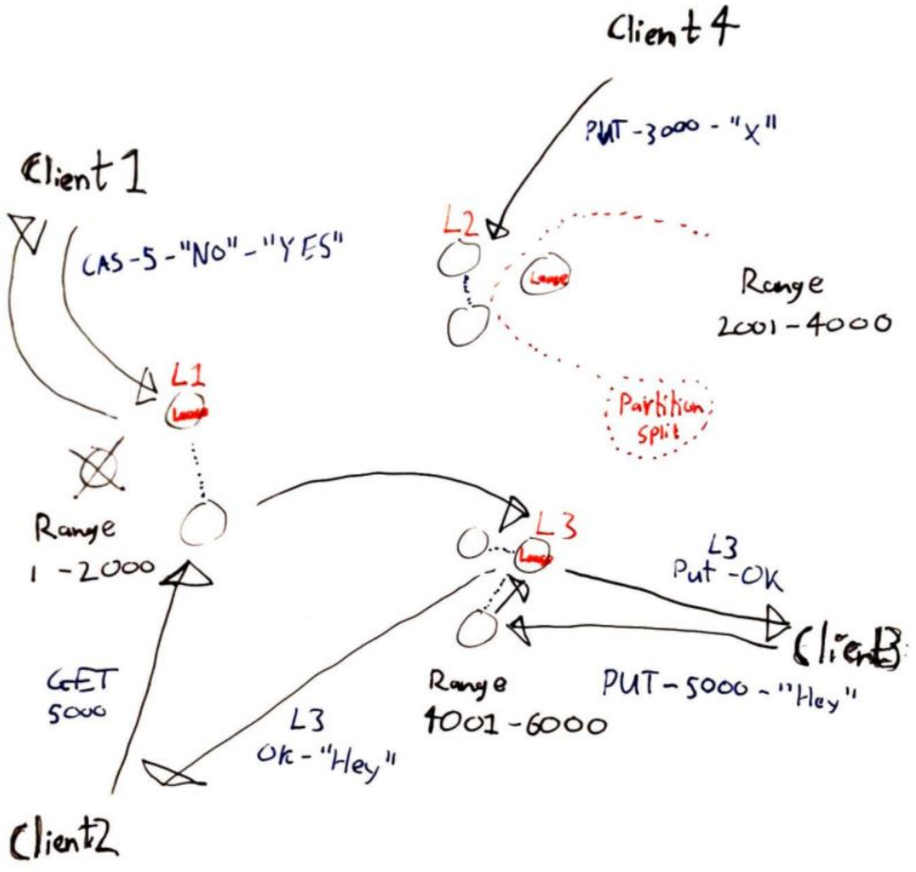
ID2203 – Distributed Systems, Advanced course

Partitioned, distributed in-memory key-value store with linearisable operation semantics.



Course Coordinator:
Seif Haridi

Course Assistant:
Lars Kroll

Óttar Guðmundsson
Xin Ren

2018-3-10

Abstract
In most moderns system, data must be highly available, scalable and responsive. This is often done by using distributed key stores engineered in such a way that they are linearizable, meaning that servers in the same partition should agree on a sequence of operations. This final project is about developing a system that can handle multiple servers that are split into groups of specific range for a key value store, being either a leader or a replica. The system is fault tolerant when majority of nodes in each partition is alive, defined as N/2 + 1. The advanced algorithm sequence paxos (sequence consensus) is used for each partition, to make linearizability achievable. The system does not handle reconfiguring of nodes, but this would undoubtedly increase the availability of the system.

Introduction

For this project, the programming language Scala was used rather than Java. We already had a lot of experience in Java so picking Scala seemed like a good opportunity to learn a new language, plus we had done exercises from the course in the same language. The project template seemed like a good way to start from, which is structured and built in Kompics by Lars Kroll. GitHub was used for source control and the code can be found here[1].

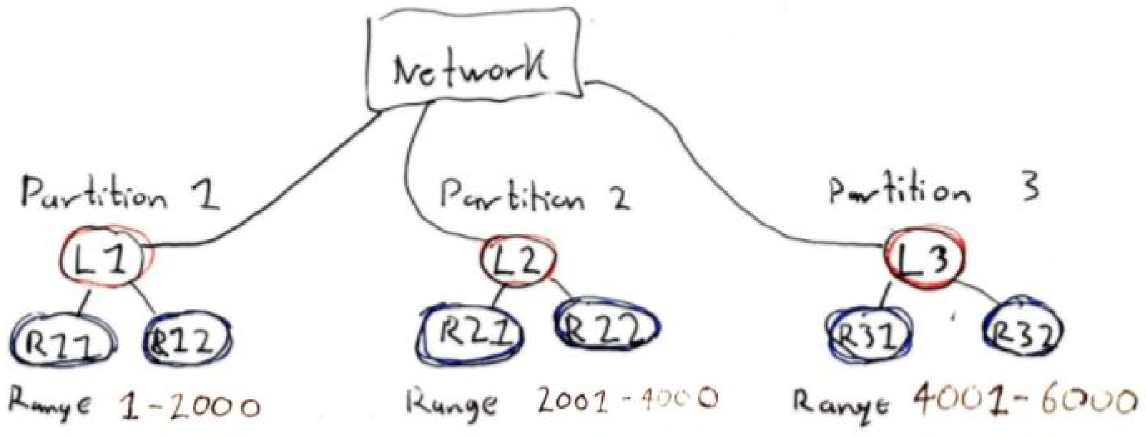
Responsibilities

To start the project, a small plan was created to delegate tasks between the group members. The proposed plan was that Xin would implement the Infrastructure (2.1) while Óttar would implement the KV-store (2.2) and the Leader-lease task (2.3). Then both group members would perform tests on each other's part to see if they were performing correctly. At the later stages, group members would adjust to tasks as they came up such as debugging, creating other test cases or writing the report. After most of the requirements had been fulfilled, Xin applied the final touches to the test cases and hunting down bugs while Óttar took care of the report.

Infrastructure

The template was a nice way of starting the project but for our implementation we needed to tweak the source code a little bit, by changing the structure of the operations, responses, lookup table and more. The project was built with passive replication in mind because from the basic course we knew that the leader in the passive replication could keep linearizability and handle the fast reads, which gave us a hint on how to implement one of the advanced tasks. We let our system split the key value store by range, creating three separate partitions that handled key values from 1 to 6000. Thus, each partition took care of 2000 keys, starting with two predefined values in its keystore. The replication degree was set to two so each partition was a set of one leader and two replicas, setting our boot threshold to nine.

[1] https://github.com/otg2/ID2203 - the repository is private. Send email to ottarg@kth.se for access



To make our implementation reliable we used several topics learned from the course such as Ballot Leader Election, Best Effort Broadcast, Perfect Link and Eventually Perfect Failure Detector.

KV store
The KV store was a simple Map that stored stringed values to given keys of integers. Clients could operate on these stores with three simple operations named *GET* (read), *PUT* (write) and *CAS* (compare-and-swap). If a node received a request about performing one of those operations it would forward it to the leader where he would perform it, but the biggest challenge was to make sure that the changes made to the KV store would also be performed in the replicas. We tackled this problem with the Sequence Paxos algorithm that we learned in the course.

Advanced task - leader leases
For extra points, we had the options of upgrading our project so it could be reconfigurable with nodes leaving or joining the system or implementing a leader lease to optimize the system for GET operations. Since we already did the reconfiguration in the previous Distributed Systems course, trying out the leader lease seemed like a proper learning experience. Picking that option was also viable for our solution since we already planned the project for passive replication, having the leader lease in mind. We implemented a safe lease so that only one leader in a group can have lease by considering the asynchronous network and the clock drift of each server. An extension of the lease was also implemented. Since it was impossible to get the clock drift for each server in the project simulation (except taking the hardware apart) each node had a random value set for their drift. Fast read was also implemented and is allowed when the leader has lease and there is no outstanding write operations in the operation queue to be handled by the leader.

System Behavior and Algorithm Selection Reasoning

A bootstrap server is started first and then 8 servers are started and connected to the bootstrap server. When all 9 servers are ready, the bootstrap server assigns all the servers to a range partition group and create a same lookup table, which contains mapping from range to servers, for each server and send it to each server. Each server gets the information about its own range, its group members and all the servers in the network from the lookup table. Clients can connect to any server in the network, but connection is rejected before all 9 servers are ready.

Ballot Leader Election (BLE) starts upon receiving the lookup table in each group. We chose BLE because we do not know the upbound of the time that needed to process a message and transport a message, and the Eventually Perfect Failure Detector in BLE can eventually help us find the upbound and detect node failure. BLE also guarantees that eventually every correct node elects the same correct node as leader. When a leader is elected, the leader would broadcast a message with its ballot to all the servers in the network so that each server can update the lookup table so that the lookup table contains(changes to) mapping from range to leader. The update is only done if the ballot in the message is larger than the stored leader ballot for the range in the lookup table to avoid that in the situation(which can happen if replica degree is bigger than or equal to 4) where a leader dies and immediately the new leader dies too. Then the 2nd new leader is elected, but the message from the 1st new leader arrives at each server later than the message from the 2nd new leader does, and then all the servers keep the dead 1st new leader in the lookup table. Monotonic unique ballots in BLE makes sure that the ballot of new leader is bigger than previous leader.

Each server in the network receives operations from clients and forwards each operation to the range leader for the range that contains the key of the operation by looking up the lookup table, which guarantees that only the leader in a group can propose operations. All the operations are replied by the leader directly to the clients.

Sequence Paxos starts when new leader is elected in a group. It guarantees that all the servers in the same group execute the same sequence of operations. Since all the operations are forwarded to the leader and then proposed by the leader to the group, all the operations are executed in the order that they are received by the leader, which guarantees linearizability.

With leader lease implemented in the Sequence Paxos, the system can handle fast read which means when the leader receives a read operation, it replies directly without contacting majority. But this is only allowed when the leader has lease and there is no write operation for the same key waiting for execution. The leader requests to group members to extend lease before the current lease ends. When a new leader is elected, it requests lease from the group member every few seconds until it gets lease. When network partition happens, the old leader with lease in one partition can handle fast read and the new leader without lease in another partition can not execute any operation until it gets lease (the new leader behaves the same when the old leader who has the lease dies).

PerfectPointToPointLink and Best Effort Broadcast which was implemented on PerfectPointToPointLink were used, because we want that when both sender and receiver are alive, the message is eventually delivered exactly once and no message is delivered unless it is sent.

The Network in Kompics is used as PerfectPointToPointLink with TCP which also has FIFO property needed in Sequence Paxos.

Testing

After implementing all the code needed for our the algorithms, connecting components together and modifying the template according to our needs it was time to test if everything was working as planned. Starting nodes manually and sending single operations through the network was a success as well as killing nodes individually. What was needed was to test the system under pressure, that is, multiple clients sending multiple requests at the same time, killing leaders of partitions and see if our system managed to cope with the requirements of linearizability and availability.

Luckily Kompics allows us to create complex simulations to do those things. By simply creating new clients on the fly and sending operations through we can store the sent operations in the client and match them with expected outcomes of the responses received. This is really handy so we do not need to read through all the log files and verify that the system holds. So after a few talks and speculation we came up with several major test cases to verify if everything worked as it should. Not only did these test cases help us prove that our system was correct, but it helped us find a lot of hidden bugs we hadn't thought off. So after debugging for a while to let our test cases pass, we were quite happy with the results.

Note: We did not implement test cases for the basic components that the bigger algorithms are based on. For example, we do not have a specific test case to see if our PerfectPointToPointLink passes or not. Instead, we create a more complex test cases that test our system as whole. BalloutLeaderElection uses the PerfectPointToPointLink abstraction so if we can create a test case for our leader election and it passes, we know by deduction that our link implementation works correctly.

Basic operations should work out

To make sure the operations worked in the KV store before doing further tests, we had to make sure that the operations implemented worked correctly.

The scenario is performed as such

1. A client writes values to specific keys which are successfully stored.
2. Another client now tries to read the values from the same keys as before and should get the updated values.
3. The third compares and swaps a key and the operation should complete.

Results
Operations were successfully routed to the leader responsible for that range. All of the operations got responses which matched the expected values, so the operations for GET, PUT and CAS all work as expected.

Old leader and new leader should be different (KV should be the same)
When a leader of a partition dies a new replica should step up as the new leader using the BLE algorithm. Testing this is very important since not only do we need to make sure that we get a new leader for the partition, but also that other nodes eventually detect that a leader is dead and that the new leader should have the same values written in it's KV store.

The scenario is performed as such
1. A client requests to write a value to a key and then read the value with the key and gets replies from the leader of that partition.
2. The leader that replied in step 1 is now killed
3. Another client requests to read with the same key and should get the same value as before, from a different leader.

Results
Seeing that all operations were successfully returned to the clients with expected responses and by a different node after initial leader is killed, we can verify that a new leader is elected and that both stores are consistent. Do note that only one node was killed, meaning that majority still lives and the system is still fault tolerant.

All clients connected to a different server should get reply from same leader (fast read)
We need to make sure that only the selected leader of a given partition can respond to a request of keys that he is responsible for. To make sure that this is correct we created multiple clients sending requests of the same key to the network. This case also verified that fast read worked.

The scenario is performed as such
1. Multiple clients request to read the same key but send their request to random nodes in the network
2. All requests are forwarded to the leader responsible for the key
3. The clients should all receive response that reads the same value, sent by the same leader

Results
All operations are returned to the clients successfully. Not only do they receive the same value for the same key but when the sender of the responses is compared with all other messages, we can see that it comes from the same leader. All the operations were replied with fast read tag as there was no outstanding write operation for the same key waiting for execution.

All operations sent in order should be linearisable

As said earlier, we managed to send a few GET, PUT and CAS operations through the network. We hadn't tested out sending them through many clients in an order and determine if the operations were linearizable. To do this we created a scenario where clients connecting to different servers would perform all of these operations in a mixed manner and see if the operations were linearizable.

The scenario is performed as such

1. Several GET, PUT and CAS operations are sent by multiple clients to different servers
2. The operations are received and forwarded to the leader, added to proposed command and executed linearly making sure that all of the replicas are updated as well
3. The clients receive their responses so they can all agree on a global order

Results
By matching the outcome of the operations with an expected linearizable outcome, we can see that the test holds for all operations.

After majority of nodes die, operations should timeout

Our system model is defined as fault tolerant as long as majority of nodes in each partition are alive. To make sure that the algorithm is correct we must be sure that a node in a partition does not step up as a leader if it doesn't receive heartbeat response from the majority.

The scenario is performed as such

1. A client requests to write a value to a key and then read the value with the key and gets replies from the leader of that partition.
2. The leader that replied in step 1 is now killed
3. Another client requests to read with the same key and should get the same value as before, from a different leader.
4. That leader replied in step 3 is also killed. The last node in the partition will not be leader since a majority of nodes are dead, receiving no heartbeat response from majority.
5. A new clients requests to read with the same key but doesn't get any reply and the request timeouts, meaning that there is no new leader.

Results
We do see that a new leader steps up when the first one is killed and we successfully get a response from the new leader. When the new leader is then killed the response timeouts since

no new leader is elected and the operation is sent to the dead leader. It is impossible to elect one more new leader due to majority of nodes being dead.

Read following Write should not be fast

Using the lease allows the leader to fast read values without letting the replicas also read. This can be problematic if a leader is handling a PUT operation on a key, but it would receive a GET request for the same key while it is writing the new value. Thus, a fast read should not happen if the leader is writing a new value. This is to avoid that a client writes and then reads, but reads the old value before the write operation.

The scenario is performed as such

1. A request about reading a key that doesn't exists should get a fast response from the leader.
2. A PUT operation with value for the same key is sent to the leader.
3. Shortly after a GET request for the same key is sent but the leader does not perform the fast read on the key since it is finishing the former operation.

Results

We see that the first GET operation responds as fast read but the second one does not since there is a PUT operation happening at the same time.

Benchmark: load testing should work

To check performance on the leader lease in the paxos algorithm, a new variable was introduced to the config file that could turn the leader lease mechanism on or off. The only way of finding out if the implementation was correct was to run the test case two times, with and without the lease. The total time of execution should be shorter if leader lease is enabled.

The scenario is performed as such

1. A client is created, sending 10000 request to read a simple value from the KV store
2. The leader replies the value if the operation was successful
3. When all of the pending operations are completed, the system time is recorded and compared.

Results

By running two test, with (06.31 sec) and without (12.76 sec) the lease, we improve the speed of the system by nearly 50% with fast read.

Network partition should not impact linearizability

To make sure that the lease is working as intended and keeps linearizability, we create a scenario where we split the partition by changing the network at runtime. We tried using the PartitionedNetworkModel in Kompics simulation but it didn't work properly so a custom partition was created in the code. The PerfectPointToPointLink was modified so that when network partition is configured to happen, replicases would not send any messages to their leaders so that leaders are isolated from their replicases and in the replicases' partitions, a new leader for each group was elected.

The scenario is performed as such

1. 3 clients are created and connected to different servers in one group, sending GET and PUT requests at the same time
2. Network partition happens and leader with lease gets isolated from 2 replicas.
3. Another 3 clients are created and connected to different servers in the same group, sending GET and PUT requests at the same time

Results

GET and PUT interleaves were created in step 1 and 3.

In step 1, all the operations were forwarded to the leader and handled/replied in the order they were received by the leader, which proved that our system is linearizable during normal operation.

In step 2, new leaders were elected in replicases' partitions for each group, so there were two leaders existed for each group at the same time, one was the original leader with lease, the other one is the new leader without lease.

In step 3, the GET request sent to the (old) leader with lease got fast reply until a PUT request received with the same key, after that all the operations for the same key were not replied (PUT requests pending because of no majority, GET requests for the same key pending because of PUT). All the operations sent to the (old 2) replicases were forwarded to the (new) leader without lease, and they were replied by the leader in the order that they arrived at the leader after the leader got lease. This proves that our system keeps linearizable during network partition.

Looking at these results, we can see that the following properties hold for these abstractions.

PerfectLink

PL1 *Reliable delivery*	If Pi and Pj are correct, then every message sent by Pi to Pj is eventually delivered by Pj
PL2 *No duplication*	Every message is delivered at most once
PL3 *No creation*	No message is delivered unless it was sent

For all requests sent in our test cases, these properties hold since the client always gets response when no nodes were crashed and only get response to it's request and at most one response to each request.

Eventually Perfect Failure Detector

EPFD1 *Strong Completeness*	Eventually, every process that crashes is permanently detected by every correct process .
EPFD2 *Eventual Strong Accuracy*	Eventually, no correct process is suspected by any correct process

The nodes in our test cases eventually find a node that has crashed and doesn't suspect a running node so these properties hold.

Best Effort Broadcast

BEB1 *Best-effort-Validity*	If Pi and Pj are correct, then any broadcast by Pi is eventually delivered by Pj
BEB2 *No duplication*	No message is delivered more than once.
BEB3 *No creation*	No message delivered unless broadcast

This is guaranteed by PerfectLink.

Ballout Leader Election

BLE1 *Completeness*	Eventually every correct process elects some correct process if a majority are correct
BLE2 *Eventual Agreement*	Eventually no two correct processes elect different correct processes
BLE3 *Monotonic unique ballots*	If a process L with ballot N is elected as leader by Pi, all previously elected leaders by Pi have ballot numbers less than N, and (L,N) is a unique number

In our test, we observed that when there was majority nodes in a group, every node elects the same leader, so BLE1 and BLE2 hold. When a new leader was elected, the lookup take in each node in the network was successfully updated (based on the comparison of ballots) which proved BLE3 hold.

Sequence Paxos

SP1 *Validity*	If process P decides V then V is a sequence of proposed commands without duplicates.
SP2 *Uniform Agreement*	If process P decides U and process Q decides V then one is a prefix of the other.
SP3 *Integrity*	If process P decides U and later decides V then U is a prefix of V

SP4 *Termination*	If command C is proposed then eventually every correct process decides a sequence containing C

In the test cases we know that the leader and the replicas all share the same accepted sequence and new accepted operations are added to the sequence so these properties hold.

These tests show only one single run of predefined, controlled operations on the system. There are nearly endless different cases that could be created to further test the implementation but due to time limitation, we assume that if we can prove that it works for one case, it should work for all cases. This is quite a narrow assumption though, since our experience in developing and testing these kind of systems is limited.

All test cases can be tested by running the project under sbt and writing *test* to the console.

Conclusions

We both agree that the project was really educational. The course programming exercises simply taught us to translate a pseudo code into scala code to pass a given test case, but implementing all of them in such a way that all components were connected and seeing how they interacted between each other to form a system was a real challenge. On top of that, it was nice to get in more depth of the Scala language and working more with Kompics (specially the simulations) and seeing the combined techniques form a system that is both practical and usable for modern requirements.

As the project progressed we understood more clearly what it meant by saying that testing distributed systems is really hard. The harder it was to find the bug, the more logging was needed and with more logging, it got harder reading through all of it. Thus, it was a great experience learning first hand how complicated this can get, how time consuming it is and how much it tests your patience.

Since our demo has only three nodes in each partition, the system doesn't survive for long if nodes die frequently. It would have been great to add the other advanced task to this project to make it even better. We already had an idea on how to do so, but it was clearly stated that we should only pick one of the extra tasks. That being said, maybe in the near future we can sit down and extend the project to make it reconfigurable or even add a UI for the client.

[illegible] [illegible]

Óttar Guðmundsson Xin Ren .