ID2203 Project

Max Meldrum
Vaikunth Srinivasan A

March 11, 2018

1 Introduction

In this project, we implemented a distributed in memory key-value store with linearizable operation semantics. We take advantage of the Kompics [1] programming model and utilise the Kompics Scala DSL. We were given a lot of freedom when it came to the design of the system, as long as we satisfied linearizability. We had the option of either building the entire system from scratch or using an existing project template[1] that sets up the basic features such as single-partitioning, overlay and routing. It was an easy choice, we went with the template. Our implementation is hosted on github[2]

2 Infrastructure

In this section, we describe how our system is designed.

2.1 Overview

The key-value store is portioned based on the key-space with each partition responsible for a particular key range. The partitions constitute a replication group i.e., each partition consists of a primary and its backups. The replication groups are responsible for handling client requests and providing linearizable operation semantics.

[1] https://gits-15.sys.kth.se/lkroll/id2203project18scala
[2] https://github.com/Max-Meldrum/id2203-project

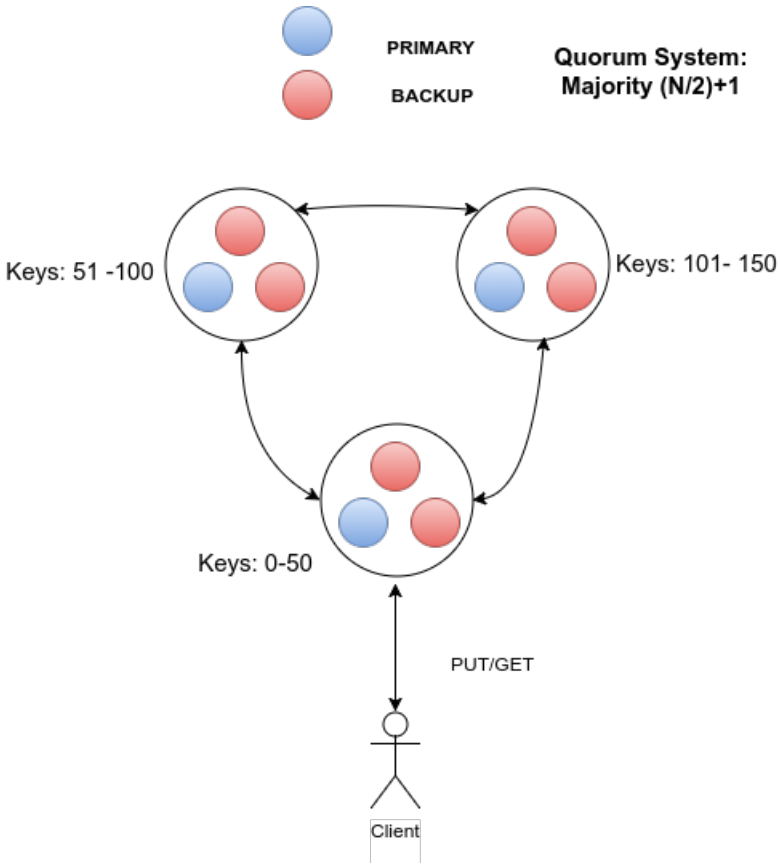

Figure 1: System architecture

2.2 System Model

We decided to go for the fail-silent model, where we assume our environment to be asynchronous. To ensure consistency across all replicas in a certain replication group, we use a primary-backup scheme and implement an Atomic Broadcast protocol that is heavily inspired by Zab [2] which ZooKeeper [3] relies on.

2.2.1 Assumptions

We are assuming that all our network based components have access to a perfect-link abstraction.

3 Replication Protocol

3.1 Atomic Broadcast

Zab, which is the algorithm behind ZooKeeper, is a crash-recovery atomic broadcast algorithm. A primary server handles client requests and broadcast state changes to the rest of the backups. The delivery order of state updates is crucial. This order is often referred as Primary Order [2]. Zab consists of 3 phases. However, as we are not implementing our kv-store in a crash-recovery model, we skip the synchronising phase and only implement Leader Activation and Active Messaging [4].

In this report, we will put more focus on describing how the broadcast phase works and not so much on Leader Activation. Zab assumes all communication channels are FIFO (TCP), by utilising FIFO channels, preserving the ordering guarantees becomes very easy [5]. This was not an issue for us as the channels in Kompics provide FIFO semantics. The Active Messaging in Zab is very similar to a classic two-phase commit. Once a primary has received a client request, it broadcasts the proposal that includes its monotonically unique proposal id. Replicas that receive proposals return with an acknowledgement (ACK). As soon as the primary has received a majority of ACKs for a given proposal, it will issue a commit to itself and its backups. This whole process can be seen in figure 2 down below.

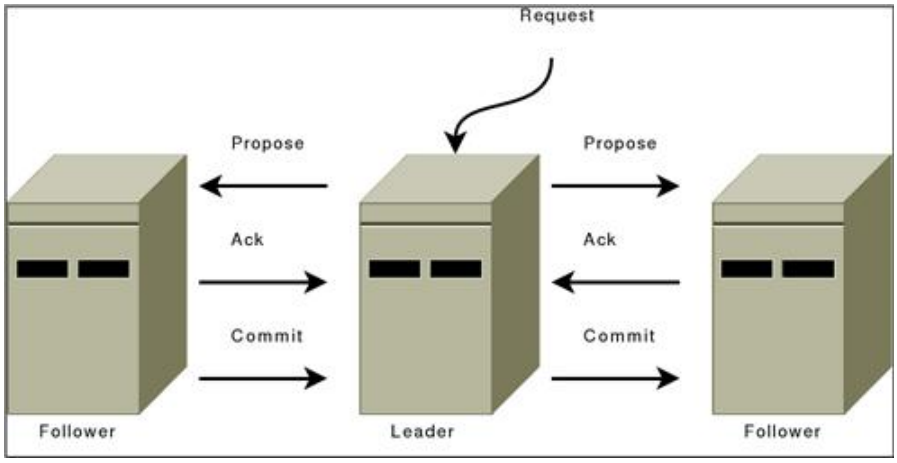

Figure 2: Zab messaging [4]

3.2 Failure Detection

As we are in the fail-silent model, we do not rely on a perfect failure detector or an eventually perfect failure detector. We utilise the Zab approach, where the primary periodically checks if it has a majority of backups connected to it. This means that backups frequently send heartbeats to the primary. If a primary notices that it no longer has a majority of followers connected, it enters an election phase. What if backups are isolated and cannot connect with anyone? The solution to this is that backups also expects heartbeats from the primary. If a backup stops hearing from the primary, it will enter an election phase and attempt to elect a new primary.

3.3 Leader Election

As mentioned in [4], the only requirements that we have to meet is that the leader has seen the highest (epoch, commitNumber) and that a majority of replicas has committed to following it. Our leader election is not an optimal approach, but it satisfies the previous mentioned properties. Once a replica enters election phase, it broadcasts a NewLeaderProposal event. Replicas receiving the proposal compare the commitNumber and epoch to its own, if it is safe, then we reply with a NewLeaderAck. After acquiring a majority, a NewLeaderCommit is sent out and the replica enters the Primary state and is now able to handle client requests once again.

4 Kompics Components

4.1 Overview

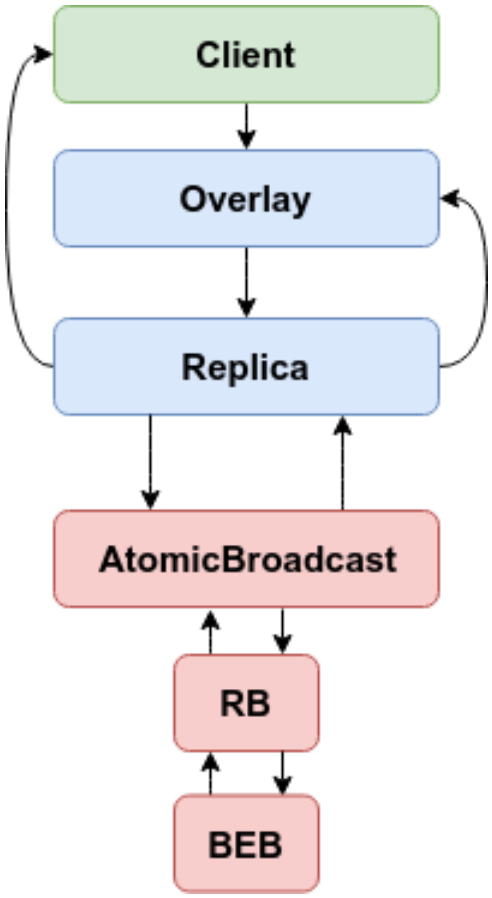

Figure 3: Component Structure

In figure 3, we show the relationships between the most important Kompics components in our system. The client initially connects with a bootstrap server, it then forwards the request to an overlay which checks which partition should be handling the request. If the current replica is in the group, the overlay forwards the request to the Replica component. If the current replica is not involved with the partition, it routes the request to any of the replicas in the replication group responsible for the partition.

4.2 Replica

As most of the logic happens in the Replica component, it will be the only component we look further into.

The Replica starts of with an InActive state, later when it is has been fully activated, it can take the role of either Primary, Backup or Election. To enable Replicas to send heartbeats, we utilise Timers. The Primary and backups have separate timer id's. If a backup changes state to Primary, it then cancels the backup timers and enables primaries. Same goes for the other way around. We earlier discussed Atomic Broadcast and Leader Election in section 3.1 and 3.3, now lets look at how it relates to the KompicsEvent's that each Replica can send and receive.

Atomic Broadcast KompicsEvents:

```
case class AtomicBroadcastProposal(clientSrc: NetAddress, epoch:
    Int, proposalId: Int, event: KompicsEvent, addresses:
    List[NetAddress]) extends KompicsEvent with Serializable {
  val uuid: UUID = UUID.randomUUID()
}
case class AtomicBroadcastAck(src: NetAddress, dest: NetAddress,
    event: KompicsEvent) extends KompicsEvent with Serializable
case class AtomicBroadcastCommit(payload: KompicsEvent) extends
    KompicsEvent with Serializable
```

Leader Election KompicsEvents:

```
case class NewLeaderProposal(src: NetAddress, epoch: Int,
    lastCommit: Int, group: Set[NetAddress]) extends KompicsEvent
    with Serializable

case class NewLeaderAck(src: NetAddress, dest: NetAddress, event:
    KompicsEvent) extends KompicsEvent with Serializable
case class NewLeaderCommit(payload: KompicsEvent) extends
    KompicsEvent with Serializable
```

5 Testing

Here, the Kompics Simulation framework has been used for testing and verifying various properties by simulating a scenario related to their specifications. A test package consists of a test scenario and a respective client to test the operations on the server. These scenarios are a good indication of whether the implementation is correct or not.

5.1 Operations

The capability of the store to support the fundamental operations such as GET, PUT and CAS is tested here with a scenario which is designed to start a cluster of servers as well as a test client. The client issues a set of PUT operations which are followed by a set of GET operations to read the values stored at the corresponding keys after the PUT operation. And subsequently, a set of CAS operations passing the parameters key, value and reference value is executed. During the CAS (Compare and swap) operation, the previously read value is compared with the current value and if there is a match, the swap occurs with the new value. Upon the termination of this test, we could verify that all operations were successfully executed and expected responses were received.

5.2 Linearizability

Linearizability is a correctness condition for concurrent objects that exploits the semantics of abstract data-types. In a linearizable system, every operation seems to execute atomically and instantaneously at some point between its invocation and response. The test for linearizability can be implemented as follows: consider a scenario where a set of servers is started as a cluster along with their corresponding sequential client. The test scenario basically takes as an input a sequential specification and a concurrent history of the operations which are written into a queue. A decision procedure known as the Wing and Gong algorithm can then be used to check for linearizability. The algorithm is shown below:

```
/**
*  Inputs: A complete history h and a sequential specification S
*  Returns: true if h is linearizable
*/
def isLinearizable(h, S) : Boolean = {
     if (h is empty) return true
     else {
        for each operation op {
        // try linearizing op first
        let res be the result of op in h
        run op on S
        if(S gives result res && isLinearizable(h  op, S))
            return true
        }
      return false
      }
}
```

6 Leader Lease

When a replica enters the Primary state and the lease option is on, a LeaseRequest is broadcasted to all replicas. Once the primary has gotten a majority of LeasePromise's, it enables the so called fast-read mode. As long as $C(T) - tL < 10 * (1 - p)$ is satisfied, it can serve GET requests without contacting a majority. Important thing to note is that we do not use a real clock drift rate, we simulate it with a randomised rate between 0.1-0.2.

We did not write test scenarios that interleave reads with writes during a partition. The reason for this is that our implementation is different than Sequence-Paxos/Multi-Paxos in the sense that when a primary is isolated from the rest of the cluster, it will no longer be in contact with a majority of replicas. Hence, GET requests wouldn't be allowed as the replica has entered an election phase. The benchmark was performed by inspecting the real-time of the simulation.

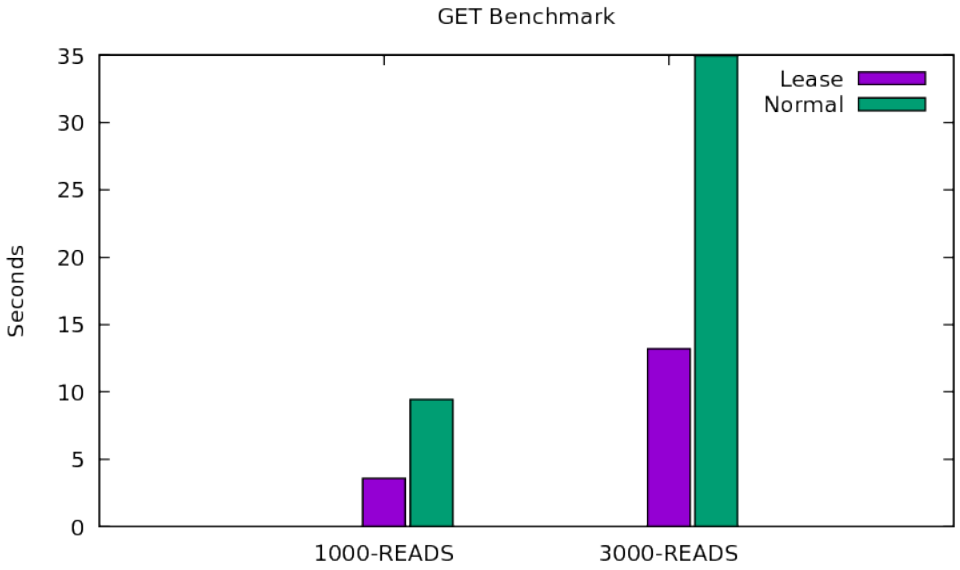

Figure 4: Leader Lease enhancement

7 Contributions

- Max Meldrum

  Replication protocol, Infrastructure, Broadcast and Lease.

- Vaikunth Srinivasan A

  Broadcast and Testing

References

[1] Cosmin Arad, Jim Dowling, and Seif Haridi. Building and evaluating P2P systems using the kompics component framework. In *Peer-to-Peer Computing*, pages 93–94. IEEE, 2009.

[2] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of*

the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks, DSN '11, pages 245–256, Washington, DC, USA, 2011. IEEE Computer Society.

[3] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[4] Apache Foundation. Zookeeper internals. `https://zookeeper.apache.org/doc/r3.5.2-alpha/zookeeperInternals.html`, 2016. Accessed: 2018-03-09.

[5] Benjamin Reed and Flavio P. Junqueira. A simple totally ordered broadcast protocol. In *Proceedings of the 2Nd Workshop on Large-Scale Distributed Systems and Middleware*, LADIS '08, pages 2:1–2:6, New York, NY, USA, 2008. ACM.