ID2203 Distributed Systems

Preliminary Report

[stormy-haystack]

A Distributed Key-Value Store with Strong Consistency

Johan Mickos

Khaled Jendi

2017-02-19

Johan Mickos
Khaled Jendi
ID2203, KTH
Preliminary Project Report

Problem Statement

The goal of the project is to implement and test simple partitioned, distributed in-memory key-value store with linearizable operation semantics.

Requirements

The project is divided into three sections, each providing an additional layer of complexity and functionality. Based on the project problem statement and project description, the system should fulfill the following requirements:

1. Support partitioned key-space among several nodes
2. Assign nodes to partitions and replication groups
3. Values of portioned key-space should be replicated with a specific replication degree δ
4. Support GET, PUT, and (an optional) CAS operations
5. Provide failure detector to detect and handle incorrect nodes
6. Provide broadcasting abstractions to broadcast messages within and possibly across replication groups
7. Ensure that the system fulfills the linearizability property across all operations and clients
8. Allow reconfiguration to account for dynamic leaving and joining of nodes in the system

Assumptions and System Descriptions

Fail-Noisy Environment

Partial Synchronous [1] assumes that the timing assumptions only hold eventually, without stating when exactly. This means that there is a time after which these assumptions hold forever, but this time is not known so instead of assuming a synchronous system, we assume a system that is eventually synchronous.

The fail-noisy abstraction combines a set of basic abstractions which are crash stop, perfect links and eventually perfect failure detector.

Crash-Stop Model

It is a model which assumes [2] that a process executes its algorithm correctly, but may crash at some time; after a process has crashed, it never recovers. once it has crashed, the process does not ever perform any step again and it is considered faulty node. The correct node is the node that never crashes.

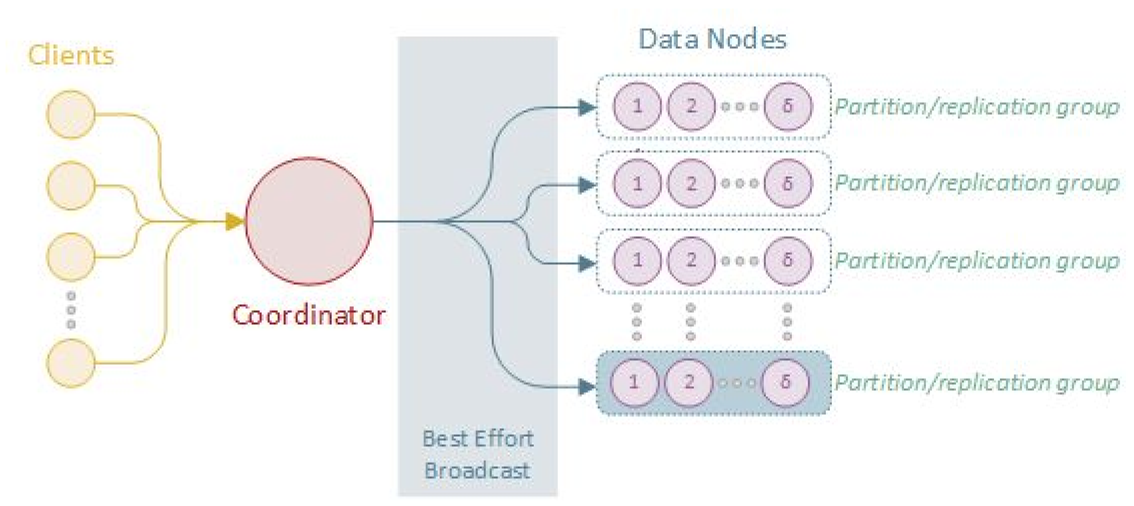
Perfect Links

Perfect links are point to point links that [3] are characterized by three properties. The reliable delivery property together with the no duplication property ensures that every message sent by a correct process is delivered by the receiver exactly once, if the receiver is also correct. The third property, no creation, is the same as in the other link abstractions.

Eventually Perfect Failure Detectors

It is an abstraction that allows to detect a faulty (crashed) node eventually which means the failure detector can perform wrong in the beginning but eventually the failure detector will be accurate. The important assumption of eventually perfect failure detector is that the time until it becomes accurate is unknown. So [4] an eventually perfect failure-detector abstraction detects crashes accurately after some a priori unknown point in time, but may make mistakes before that time. This captures the intuition that, most of the time, timeout delays can be adjusted so they can lead to accurately detecting crashes

Architecture and Design Choices


Clients
Coordinator
Best Effort Broadcast
Data Nodes
1
2
δ
Partition/replication group
Partition/replication group
Partition/replication group
Partition/replication group

Coordinator Node

The coordinator node assumes the responsibilities of **bootstrapping** the system, **routing** client operations to their respective partition groups, and **partition management**.

Bootstrapping

The bootstrapping process takes inspiration from the template code provided by Lars. The coordinator node will await incoming connections from the server nodes until the system is supported by enough nodes to support the configured replication factor across the key-space. During this process, the coordinator assigns nodes to partition groups. When the bootstrapping process has completed, the coordinator broadcasts the finalized partition groups to the server nodes and the system enters the “ready” state, indicating that it is available to process client operations.

Operation Routing

As the coordinator is accepting server nodes during the bootstrap phase, its overlay component builds up a routing table based on the partition groups across the key-space. Each client operation (GET(key), PUT(key, value), CAS(key, value, newValue)) will request a specific key. This key will be hashed and assigned to a replication group for processing. The forwarded operation will be distributed to the entire replication using **best-effort broadcast** across a **perfect link**, and the operation’s response will be returned to the client once the replication group has completed processing. All messages contain the addresses of both source and destination nodes, which will be updated along the way to specify the client as the final recipient.

Partition Management

As hinted at above, the connecting server nodes will be assigned to partition and replication groups to split up the key-space and hopefully aid in increasing availability in the case of network partitions. The joining server nodes will be placed into replication group buckets according to the current system replication state. During the bootstrapping phase, replication groups will simply be assigned server nodes in a sequential fashion. If the system becomes under-replicated, then incoming server nodes will join the under-replicated groups.

Group Membership

In order to support reconfiguration and dynamic leaving and joining of nodes, the coordinator node will need to make use of an **eventually perfect failure detector** to detect failed server nodes and in response update the system state to an under-replicated one. The details of how the group membership component will operate are undecided, but the essence of it is that

1. Correct server nodes are members of active replication groups
2. When servers are detected to fail, the coordinator updates the correct nodes with the new replication group view
3. When servers rejoin the system, the coordinator node will assign them to a group and allow the group to gossip amongst themselves to bring the new node up to speed with the current data state

Because of the requirement of **linearizability**, we will have to make sacrifices in data availability in the presence of network partitions. What this means is that when the system is **under-replicated**, the clients will experience reduced availability until it is stable once more. An example of this is that clients may be forbidden to write to certain key partitions during this time and may only read the data from the remaining correct nodes.

Data Nodes

The data nodes (previously referred to as *server nodes*) are responsible for **providing the key-value store functionalit**y and **maintaining the desired replication factor**. They will receive a list of neighbors within their replication group from the coordinating node during the bootstrapping phase or reconfiguration phase.

In order to maintain linearizability/strong consistency across the data and clients, the server nodes will maintain an **active replicated state machine** (RSM) across their partition groups. The reason for utilizing active replication rather than passive replication is to allow for reconfiguration and group membership (which are the final components of this project). An active replication scheme will make implementation of group membership and view synchrony a bit more intuitive and simple than a passive one.

The implications of using an active RSM are that the partition/replication groups will need to support **total-order uniform broadcast** and a **sequential consensus algorithm** (such as multi-Paxos) when they receive client commands. Only when consensus has been reached across a replication group will the client be informed of the operation response.

References

[1]: Introduction to Reliable and Secure Distributed Programming, Second edition, Page 47

[2]: Introduction to Reliable and Secure Distributed Programming, Second edition, Page 25

[3]: Introduction to Reliable and Secure Distributed Programming, Second edition, Page 37

[4]: Introduction to Reliable and Secure Distributed Programming, Second edition, Page 53