JBstore: A Fault Tolerant Distributed KV Store

Tobias Johansson, August Bonds

March 7, 2016

Contents

1 Introduction

A key-value store was implemented in Kompics [1], a programming framework for distributed systems that implements protocols as event-driven components connected by channels. The goal of the project was to implement and test and simple partitioned, distributed in-memory key-value store with linearizable operation semantics. Decisions regarding the implementation of the system and the results of those decisions are presented in this report.

You can find a service specification for the key-value store and an overview of the needed infrastructure and its considerations. At the end verification scenario simulations and tests are presented.

2 Service Specification

The JBstore is a linearizable single-writer, replicated key-value store. It can be configured for degrees of replication δ and number of participating nodes. It supports two operations: $PUT \mid key, value$ and $GET \mid key$. A formal definition follows.

- Module
  - Name: JBStore, instance jb
- Events
  - Request: $< jb, PUT \mid key, value, id >$: Requests a value to be stored.
  - Indication: $< jb, OK \mid key, id >$: Confirms that the value has been stored.
  - Request: $< jb, GET \mid key, id >$: Request a value from the store.
  - Indication: $< jb, VALUE \mid key, value, id >$: Returns the stored copy associated with key key.
  - Indication: $< jb, NOTFOUND \mid id >$: Triggered by failed GET.
- Correctness
  - JB1: JBStore has linearizable shared memory semantics.
  - JB2: JBStore will perform every request exactly once.
- Model
  - The implementation assumed the fail-silent model but with the introduction of an EPFD we planned to move to the fail-noisy model and introduce reconfiguration. This is discussed in a later chapter.
  - At replication degree δ JBstore requires $\lceil \delta/2 \rceil$ nodes in each replication group to be correct.
  - To achieve linearizability in the fail-silent model JBstore uses an ONAR, (1,N) Atomic Register, implemented by Read-Impose Write-Majority algorithm.
- Implementation
  - The service is composed of a group membership (GMS) that handles initial view construction and a key-value store service that stores key-value pairs and handles replication. To use the distributed store, a client/front-end is provided that handles user requests.
  - The service assumes perfect links.
  - Planned is an Eventually Perfect Failure Detector to be connected to the GMS and provide suspect/restore for reconfiguration purposes.

3 Infrastructure

The program is divided into three components. The KVStore (Application), the Group Membership Service (Middleware) and the Network component.

Furthermore we developed a closely coupled client component that could easily be converted into a Front-End for better abstraction.

The Network component receives and sends messages over the network. It directs incoming messages to the right component. Write/Read operations are sent to the application, View/Join/View Request messages are sent to the GMS.

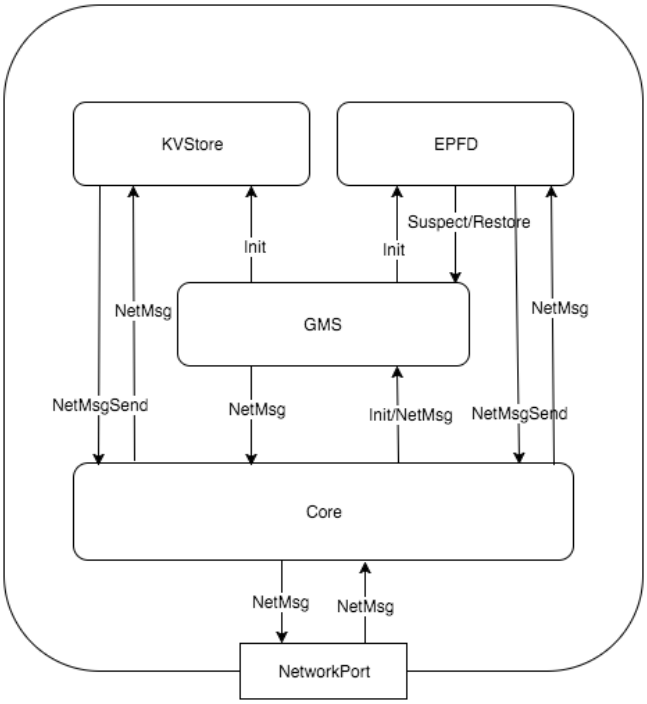

Figure 1: Node architecture

More than simply constructing or receiving the view, the GMS also indicates directly to the KVStore the current view. The store uses this information to determine its replication group.

The store is also in charge of the Read-Impose Write-Majority algorithm. The client/front end simply directs get and put requests to the appropriate node and expects a result.

These are some other aspects of our system:

- **Networking Protocol:** Kompics TCP Assumed to be a perfect link.
- **Bootstrapping:** An initial node is instantiated in "leader" mode waiting for $N-1$ nodes to join. Remaining nodes request to join the leader, and when the leader has seen $N-1$ requests the view is broadcast to all of the nodes. Each node now identifies its replication group.
- **Group Membership:** View-synchronous. Every node knows every other node. Every node only communicates with the client, and its replication group.
- **Failure Detector:** None. An EPFD is later considered to enable reconfiguration.
- **Routing Protocol:** The complete view is known to every node in the group. Each node uses the same algorithm to decide its replication group.
- **Replication Algorithm:** ONAR - (1,N) Atomic Register

3.1 Group Membership Service

A group membership component instance resided on every node. Depending on node role (leader/follower) the GMS would either wait for nodes to join or send a join request to the leader. When all followers had joined the leader broadcast the view.

We chose to divide the communication load between the GMS and the Network layer since the wrapping/unwrapping of network messages became quite messy and would have been hard to manage combined with the group communication aspects. The alternative would have been to have the GMS responsible for all communication.

Hence the only purpose of the GMS was to maintain a view for any connecting clients and the application layer to use. The actual group communication was handled by the application and network layers together.

3.2 Failure Detection

Our implementation assumed the fail-silent model, ergo, no failure detection. That being said; a solution for an Increasing Time eventually perfect failure detector (Increasing Time-EPFD) was still implemented.

The failure detection component sent periodic heartbeats to every node. In case a response timed out the EPFD indicated a "suspect" event to the GMS that then notified the application of the view change. If the node later turned out to be alive, a "restore" event was triggered.

4 KV-Store

A KV-Store component resided on every node and was in charge of keys according to the distribution function. But for $\delta > 1$ each KV-store also handled keys for $\delta - 1$ other nodes. We said that a KV-store belonged to minimum one replication group and had a built-in (1,N) Atomic Register for each key it handled. The store also knew about the other nodes who handled its keys, so on GET/PUT, corresponding reads/writes were sent to every node in the replication group.

No data was stored persistently.

4.1 Key Space

Since the cluster size was known during initialization and did not change during runs (no reconfiguration) we decided to divide the key space evenly between the nodes. With a key space $k = [0 - 2^{128}[$ and N nodes we had $2^{128}/N$ keys in each partition.

To make sure each node had the same understanding of the key distribution, we used the same function over the same commonly known view to determine range ownership.

4.2 Replication

The replication groups were calculated during initialization when the first view message reached the application layer. During execution, every GET/PUT

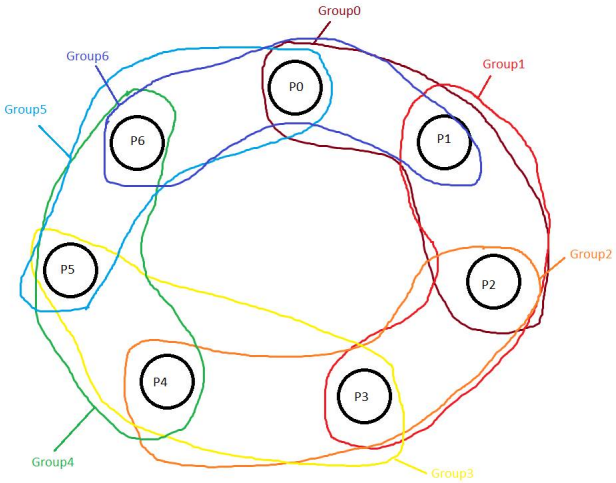

Figure 2: Replication groups $N = 7, \delta = 3$

request key was hashed at the front-end/client and sent to a random node in the replication group. That node then broadcast the request to its replication group. The replication was therefore a kind of passive replication where each node could be the "leader" for that request.

Nodes to replicate to were simply the $\delta - 1$ next nodes in the sorted view.

5 Reconfiguration

With a static number of nodes and a replication degree $\delta > 1$ in the crash-stop model: if one node crashes the cluster will still function. To restore that node we would have to spawn a new node within the same position in the view, and it would have to import all key-value pairs from another node in the replication group. If the system stood still while the node was down this would be OK. Unfortunately in a real system there could be PUT-requests being made for keys in that replication group these would have to be remembered and later re-sent to the recovered node. To implement reconfiguration, the question of "during-downtime-requests" needs to be dealt with.

Reconfiguration could also be extended to allow for a dynamically sized cluster. The current solution of evenly divided keyspace would then become useless since the whole key space would have to be recalculated on join/leave. A randomized node ID and key space divided in a "ring" (see lookup table [2]) should be used instead.

A shorter step towards reconfigurability would be using the existing EPFD to implement a Monarchical Eventual Leader Detection [3].

6 Testing and Verification

Informal definition of Linearizability: "only allow executions whose results appear as if there is a single system image and 'global time' is obeyed." A linearizable shared-memory architecture doesn't allow executions that do not make sense sequentially.

We transformed the formal definition somewhat to make it more understandable. Any value read, must be the value of the latest completed write, or any concurrent write still not complete. Any two reads need to make sequential sense as well.

To test this we executed a sequence of reads and writes and checked whether the result "made sense" (sequentially). We found two main scenarios to test:

1. Multiple Write Single Read
2. Multiple Write Multiple Read (After two or more consecutive writes perform multiple reads)

While analyzing the event logs we looked for violations of the constraints mentioned before. We did not figure out a good way to automate this testing. There has been work done in this area which is still to be investigated [4].

7 Conclusion

Reconfiguration seemed to be the trickiest aspect of distributed systems. You could always implement a static cluster with built in fault tolerance, but making it re-configurable turned out to be quite hard.

Testing and verification of distributed systems is big and complicated enough to be a course on its own. This was an interesting starting point, but more work is needed.

The task of constructing a distributed key-value store felt daunting at first, but after learning about abstractions and how to combine them, both choosing what to implement and the implementation itself became easier.

References

[1] SICS Swedish ICT and KTH Royal Institute of Technology. Kompics's documentation. `https://kompics.sics.se/`, 2015 (Accessed: 2015-01-22).

[2] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[3] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming.* Springer Science & Business Media, 2011.

[4] Gavin Lowe. Testing for linearizability. `http://www.cs.ox.ac.uk/people/gavin.lowe/LinearizabiltyTesting/paper.pdf`, 2015.