Design of Data replication model for IOT infrastructure

TOUNEKTI Dhiaeddine
University of Passau
tounek01@uni-passau.de

ABSTRACT

In this paper we are going to compare different replication systems available and decide which one is the most suitable for an IOT infrastructure of a smart garden.

1 INTRODUCTION

IOT infrastructure relies on distributed systems. A typically IOT infrastructure would consist of device layer, edge layer and a cloud layer. To make these layers resilient to failure a replication technique should be adopted. Many replication techniques where developed, with each one of them conceived to solve specific problem. Some of them were created to handle massive amount of data, order of petabyte, distributed over a large space, intercontinental infrastructure, commonly named data grid [DS15] these replication systems were created to insure fast and efficient access to a massive amount of data rather than answer the problem of consistency and availability commonly faced in highly available systems. In this paper we are going to investigate the characteristics of database replication technologies used by most used database systems namely mongodb [3] and Cassandra [4]. We are going to compare the strategies used in the replication methods in order to decide which replication method is the most suited for our use case.

2 BACKGROUND

2.1 Use Case

We are constructing an IOT infrastructure in a smart garden to automate the irrigation process. The infrastructure consists of three layers:
Device Layer : consists of sensors that are responsible of capturing the weather data (temperature, humidity,...)
Edge layer : consists of edge devices that are responsible of doing computational work, partially storing data, providing instant feedback to the user and controlling the device layer.
Cloud layer : consists of large servers providing some services to other layers like big amount of data storage and high intensive computational tasks that can not be done in an edge device.

2.2 Replication types

There are two main replication mechanism currently used, Primary - replica and Multi-Primary - replicas. Both of these designs provide Network partitioning tolerance and fault tolerance but depending on the configuration they might provide different consistency and availability levels. They can also with the right configuration provide atomicity of transactions. We will be discussing in the next sections MongoDB as an example of Primary-replica replication model and Cassandra as an example of Multi-Primary replication model.

3 PRIMARY-REPLICA : MONGODB

MongoDB[mon] is a well known NoSQL document storage database. To make its data highly available and resilient to network Partitioning and server failures it implement master-replicas mechanism

3.1 Component

The database is composed of replication clusters.
Cluster is a set of nodes (servers). Unless a network partition has happened there is only one primary node and the others are replicas. **Primary** is the server that accepts read and write request and it is the responsible of replicating these requests to secondary nodes.
Replicas are nodes that replicate the primary transactions. They do this asynchronously. They cannot accept write request and forward read requests to the primary whenever they receive one.[1].

3.2 Primary election

During this process the cluster nodes try to select the primary node. During this time no write process are accepted but read processes can be done if configured to be handled by replicas. In the case where there aren't enough replicas (less than two) for the election process a node called "arbeiter" can be added to the cluster. This node does not replicate data but just contribute to the election process.
In the case of network partitioning it might happen that the cluster ends up with two primaries, one from the old cluster and another one recently elected after the network partition. To solve this issue the old Primary will downgrade to a replica once he detects that he cannot connect to the majority of nodes.

3.3 Failure detection

nodes would send to each other heartbeat messages each two seconds if no response comes within 10 seconds the node is considered as unaccessible

3.4 CAP and atomicity

MongoDB provides tolerance for network partitioning but depending on the read and write conditions it might provide different levels of availability and consistency.
From now on we are going to consider a system where the

[1] It is to note that the commercial version of MongoDB allow for some degree of flexibility and allow client to issue write and read requests to replicas but this would violate the one-master-replicas design, for this reason it was not considered here

majority of nodes exist and less than the half of nodes are faulty.
If the write request is set to majority, which means that a write request is only acknowledged if the majority of replicas have replicated the write request, MongoDB provide strong Consistency, eventual availability and atomicity of transactions.
If the write request is set to less than the majority of nodes then MongoDB would provide read-your-write consistency, eventual availability but no atomicity is granted. This is caused by the coexistence of two primaries in the case of network partitioning. In that case the old primary would accept some write request and read requests that would be rolled back after it steps down from being a primary.

4 MULTI-PRIMARY : CASSANDRA

Cassandra[LM10] is a decentralized Column database. To provide fault tolerance Cassandra uses a Multi-Primary strategy.

4.1 Components

Cassandra consist of a cluster of nodes. All nodes are considered primary nodes. However there are two types of nodes seed nodes and non-seed nodes. As there names might suggest they are used as the cluster creation trigger. In the bootstrapping process they do not have to communicate with any other seed node.

4.2 Data replication

The server cluster is constituted of multiple Primary nodes. Cassandra uses consistent hashing [KLL+97] to assign data to nodes. In consistent hashing the hash space (ring) is divided on the available number of nodes and each node would receive and interval of the hash values. This node, also called coordinator (for data whose hash is in the interval), is in charge of read and write requests of the data whose hash falls in its interval. It is also responsible of replicating the data on the next n servers. It is to be noted that for performance reasons we might assign multiple nodes in a ring to one server in order to balance the load on different machines see the figure 1.

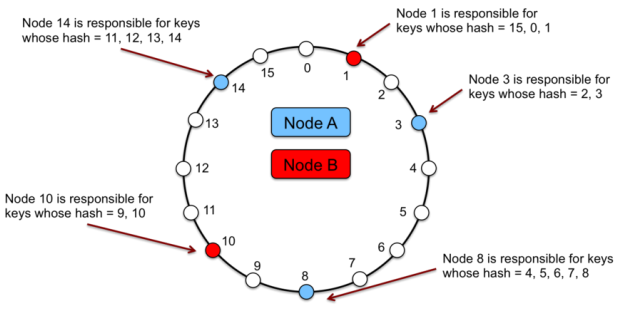

Figure 1: consistent hashing example [Nai18]

4.3 Membership

To detect other nodes, cluster server would exchange gossip messages periodically. this part makes use of two kind of clocks, one logical clock called version and another timestamp called generation. the gossip protocol runs this way :

(1) the node update its local state (version)
(2) picks another random node to exchange messages with it
(3) probabilistically try to communicate with any unreachable node
(4) Gossip with a seed [4.1] node if it couldn't reach any other node

gossip messages consist of a vector clock (generate, version) and ϕ-Accrual failure detector [HDYK04] probability alongside other network information. The nodes do not send a list of "UP" and "DOWN" of peer nodes but rather send probability of how sure a node is, that another node would not respond if a gossip message is sent to it. Then the decision to mark the node "UP" or "DOWN" is made internally. If marked "DOWN" the node will never root requests to that peer node. The ϕ value used in the original cassandra paper was set to 5 which allowed to detect a failed node in 15 seconds.

4.4 Write Conflicts

4.5 CAP and atomicity

4.6 Bootstrap

5 COMPARISON OF DIFFERENT REPLICATION SYSTEMS

6 RELATED WORK

7 CONCLUSION

8 MODIFICATIONS

REFERENCES

[DS15] Naveen Dogra and Sarbjeet Singh. A survey of dynamic replication strategies in distributed systems. *International Journal of Computer Applications*, 110:1–4, 2015.

[HDYK04] N. Hayashibara, X. Defago, R. Yared, and T. Katayama. The /spl phi/ accrual failure detector. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*, pages 66–78, 2004.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC 97, page 654663, New York, NY, USA, 1997. Association for Computing Machinery.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):3540, April 2010.

[mon] Mongodb documentation. https://docs.mongodb.com/manual/replication/.

[Nai18] Sujith Jay Nair. Consitent hashing. https://sujithjay.com/data-systems/dynamo-cassandra/, 2018.