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 streaming system replication technology using Kafka [5] as an example and database replication technologies used by 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 and Kafka as an example of Primary-replica replication model and Cassandra as an example of Multi-Primary replication model.

3 PRIMARY-REPLICAS : 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 ACID

MongoDB provides tolerance for network partitioning but depending on the read and write conditions it might provide different levels of availability and consistency.

[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

From now on we are going to consider a system where the 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 ACID transactions for single document.
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.

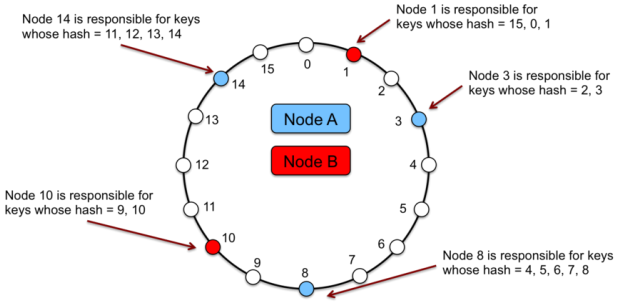

Figure 1: consistent hashing example [Nai18]

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

Since data is versioned whether using a client clock or a the coordinator clock, Cassandra follows the rule of last write win to solve conflicting data modifications. This means that the data having the latest timestamp would be preserved. To be applied this need the nodes to be synced in time and the system should consider the fact that it is impossible to reach perfect synchronization.

4.5 CAP and ACID

As we saw in the section [5.3] Cassandra's node clocks need to be in perfect synchronization to be able to preserve update order which is impossible to realize. For this reason multi-primary systems are not suitable for ACID transactions.
Cassandra, using multi-primary approach provide eventual consistency and strong availability. Let R be the number of nodes that need to acknowledge a read request, W the number of nodes that need to acknowledge write requests before responding to the client and f the number of nodes. Cassandra can tolerate t-R faulty nodes when serving reading requests and t-W faulty nodes when receiving write requests.

4.6 Bootstrap

When a node is created or recover from failure we saw that it should communicate with a seed but when initially created there are no seeds in the cluster because there is no cluster obviously. For this reason when creating a cluster for the first time the user picks up nodes that will be considered as seeds and these seeds do not have to communicate with other seeds to join the cluster.

5 PRIMARY-REPLICAS : KAFKA

Kafka [Kre11] is a streaming system to process huge volume of logs developed by LinkedIn. It is generally thought of a messaging system but it is capable of much more. As we will se we can rely entirely on apache kafka in our use Case

5.1 Data Model

Kafka splits data into topics. Each topic is split into partitions. partitions are the replication unit which means that each topic partition should fit in a node [5.2] storage capacity. each partition consists of multiple records that are ordered using id numbers assigned by the node responsible for the partition.

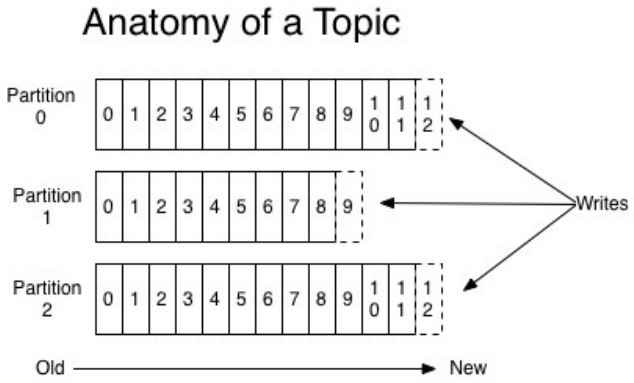

Figure 2: Topic data design[Fou17]

5.2 Components

Apache kafka uses primary-backup architecture to replicate the data. It is composed of clusters where each cluster is itself composed of nodes. nodes are also named sometimes as servers or brokers. each node can be weather a primary for certain topic partition or a backup. Multiple partitions can be assigned to a single node.

Primary is responsible for responding to all read and write requests. It persists these changes inside a log. **Replicas** replicate the primary log in a **pull-based** manner. Which means that it is not the job of the primary to replicate the data in each replica but rather each replica have to request new data records from the primary.

There are also **Consumer** groups which are thought of as broadcast domain. In other words each partition should be sent to each consumer group. In each consumer group only one consumer can subscribe to a given partition, which means that each record appended to the partition log should be sent to one particular consumer within a consumer group. **Producers** are those who publish records. They choose which topics partition a certain record should be published in. It is to note that these are roles and one node can play all the roles at the same time for different topic partitions.

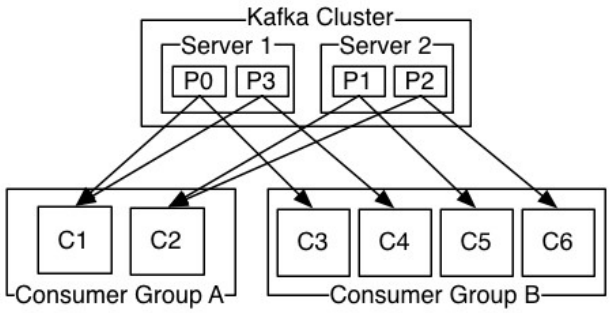

Figure 3: Consumer groups[Fou17]

5.3 Membership

kafka uses zookeeper server under the hood to persist configuration data. It assigns for each node a static id that will persist even if the node fails and the same id will be assigned to the same the node when it recovers. Add to that zookeeper keeps information about in-sync replicas (ISR) for each topic partition. a node is considered in-sync if:

- it can keep a session with the zookeeper through heartbeat
- it does not fall far-behind when replicating the primary logs. "far-behind" is defined through the configuration of the kafka cluster.

5.4 Failover

let a replicas set for each data partition have n nodes, one primary and the others are replicas. In this case the replicas set should be able to tolerate up to n-1 failed nodes without loosing any data stored.

For write request acknowledgement, kafka uses the number of nodes in the ISR set, if it reaches a preset minimum than the write request is acknowledged to the producer else it will wait until enough nodes are present in the ISR set.

In case of Primary node failure, one of the ISR set nodes will get automatically elected to be the new primary this way kafka can tolerate up to n-1 failures given that one of the ISR nodes remains in-sync (*alive*) rather than $\frac{n-1}{2}$ in the standard quorum based systems.

After a node recovers from failure it has to resync all its data before it can rejoin the ISR set.

In the case where all nodes fail together two approaches are possible

- the first node from the last ISR set recovers from failure will get elected.
- the first node that recovers from failure (not essentially part of the latest ISR) will get elected.

5.5 CAP and ACID

kafka can provide ACID transactions depending on its read configuration ($read\_committed, read\_uncommitted$) to control isolation, message delivery policy ($at\_least\_once, at\_most\_once, only\_on$ to control atomicity and minimum number of ISR nodes to

control Durability and consistency. Choosing a high number of ISR would result in a more durable and consistent data but less available service while lower number would make the service less consistent, less durable and more available.

6 COMPARISON OF DIFFERENT REPLICATION SYSTEMS

7 RELATED WORK

8 CONCLUSION

9 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.

[Fou17] Apache Software Foundation. Kafka documentation. http://kafka.apache.org/intro, 2017.

[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.

[Kre11] Jay Kreps. Kafka : a distributed messaging system for log processing. 2011.

[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.