Assignment 2

Ali Jawed, Rutvik Shah
{h20190042,h20190179}@pilani.bits-pilani.ac.in

1 Introduction

The requirement is to build a distributed key-value store of user shopping cart data for an e-commerce platform. The user of the platform can access/modify their shopping carts from multiple location. The service also provides features for the administrator to do some analysis on the user cart data. The clients access the service through a web-interface hosted on gateway server. There could be multiple instances of the gateway server running. The gateway in turn translates the user request into request amenable to REST API exposed by the key-value store.

The web-interface displays the product details available on the platform which the users can add/remove/list in the cart. The distributed key-value store resides on a cluster of nodes. Since the nodes may crash or become live again, so a map of the node identifiers and its IPv4 address is maintained using ZooKeeper service. The gateway is also responisble for ditributing and replicating the key-value data across the nodes in cluster. CRUSH service is used for key distribution on the nodes. This also takes care of balancing the nodes on node failure or new node addition.

2 Design Choices

In this section, we describe the concrete choices made for the distributed key-value storage system.

2.1 Store Consistency

The CRUSH service in coordination with ZooKeeper service chooses the set of nodes where a particular key gets replicated. The gateway must be responsive to the users update request so it cant wait for responses from all the replicas. So, we opt for *high write availability* where only acknowledgement from node suffices for the gateway to respond back to the user with a successful update message.

Quorum is used to achieve high write availability and ensuring that atleast some nodes do receive the updates made by the user. In our case, we choose the quorum as $N = 3$. For successful write we choose, as previously stated, $W = 1$. Then, eventual consistency criteria $R + W > N$ requires that $R = 3$. This means, when a user request its cart, then the gateway must receive a response from all the replicas before it sends the cart back to the user. Due to various reasons like packet loss, network partitioning

each of the replicas may not receive the user requests for updating the cartOver time, this leads the user cart values stored on the replicas to diverge.

Object versioning along with ***read repair*** can be used to achieve eventual consistency of user carts. So we opt to maintain a **version vector** associated with each cart. Version vector captures the number of modifications made to the cart in the replicas. It is a map of device identifier and a counter. The counter increments for that particular node on which the request for addition or removal of an item from the cart is made. Initially each version vector associated with a cart just contains the device identifier with counter set to zero. On receiving an update request, counters are incremented on nodes entertaining the request.

The read cart operation given by the user initiates the read repair process. The gateway sends a read request to all the replica nodes storing user cart. The nodes respond with the user cart value along with its version vector. Since the cart values on account of some failure may have diverged. The divergence due to concurrent writes are detected using the version vector. Due to partial order, the comparable version vectors get subsumed into the greatest version vector. So what remains are the divergent cart values that need to be reconciled. Only the most updated cart value should be sent to the user.

A simple merge of carts may lead to re-appearance of deleted items. So we opt to use **Conflict-free Replicated Data Type (CRDT)** as the data structure for user cart. An ORSet (Observer-Removed Set) is a CRDT which keeps track of added and removed items along with their unique tags. A merge operation on ORSet makes sure that the user get to only see the item which they intend to keep in the cart.

2.2 Secondary Indices and Admin Operations

As stated in the introduction, to facilitate some analysis by the administrator we need to maintain secondary indices on the product attributes added to the cart. These secondary indices ensure fast lookup of the key-value containing the items with the attributes mentioned in the admin operations. So we opt to mainitain secondary indices co-located along with the key-value pairs on the nodes. When a user issues an update command the key-value store also updates the secondary index pertaining to a product attribute. Secondary index is a dictionary which maps product attribute values to the user cart holding products with that attribute value. When an administrator sends a listall command for a particular attribute value then gateway initiates a read operation on all nodes and finally returns all the appropriate carts.

3 Implementation

The language of implementation was chosen to be Python. We use the specification language of Docker and Docker-Compose to create container images and a netwoked cluster of containers respectively. The `scale` feature of `docker-compose` allows to create multiple copies of containers from a single image. So we chose to scale the key-value store image to 5 containers and the gateway image to 2 containers. All the containers are put on the same network.

3.1 Architecture

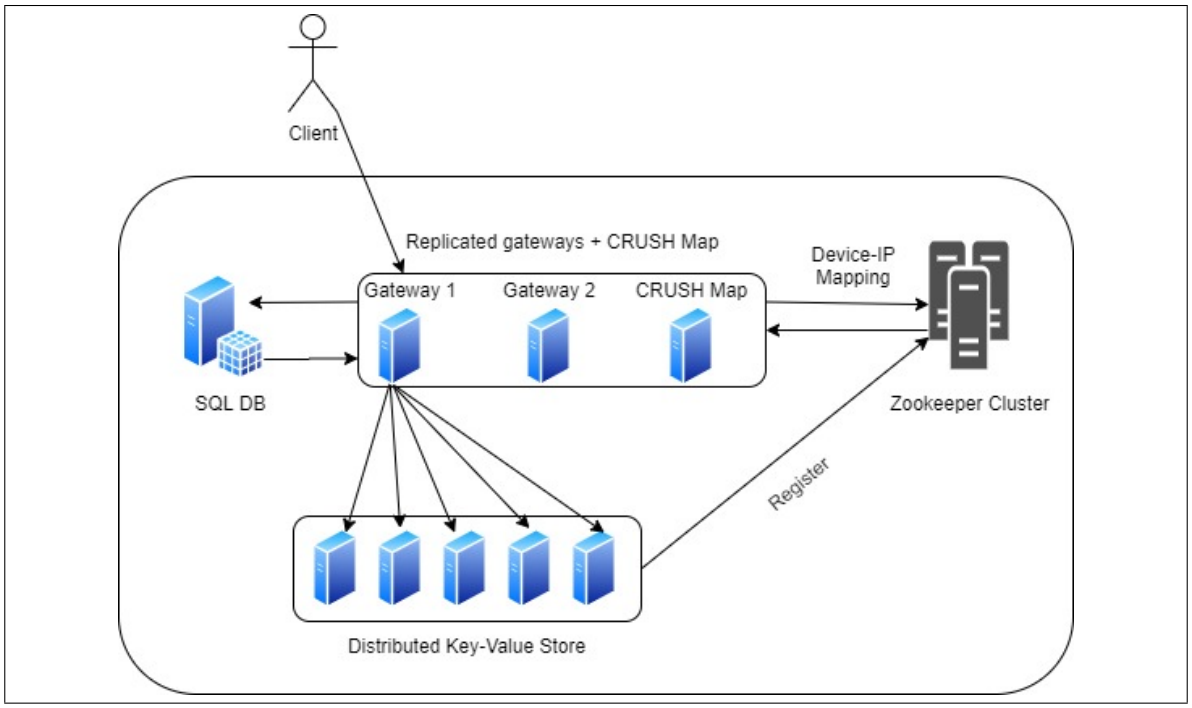


Figure 1: Architecture

The gateway hosts the web-interface which allows the user to choose the products and make changes to their carts. The gateway fetches the product details from a SQL database. The database is named `store` and contains two tables `product` and `customer`. Schema of these tables is described in the following subsection.

3.2 Table Schema

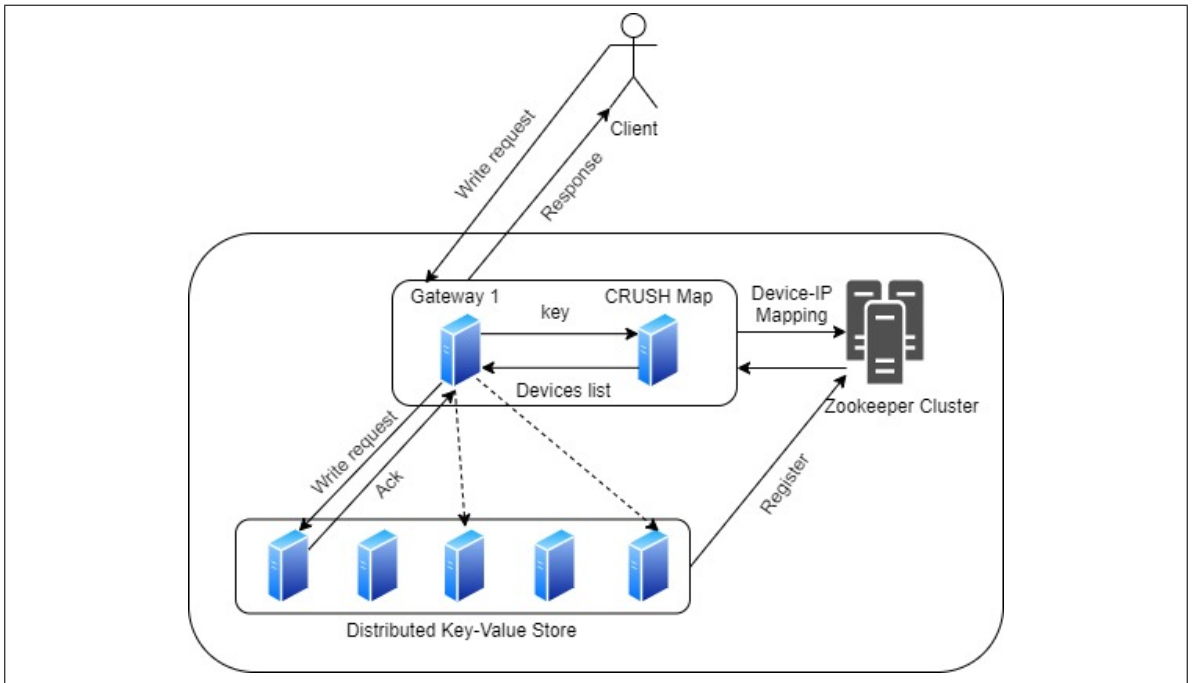
The schema of the product and customer tables are as follows:

3.2.1 Table Name: Product

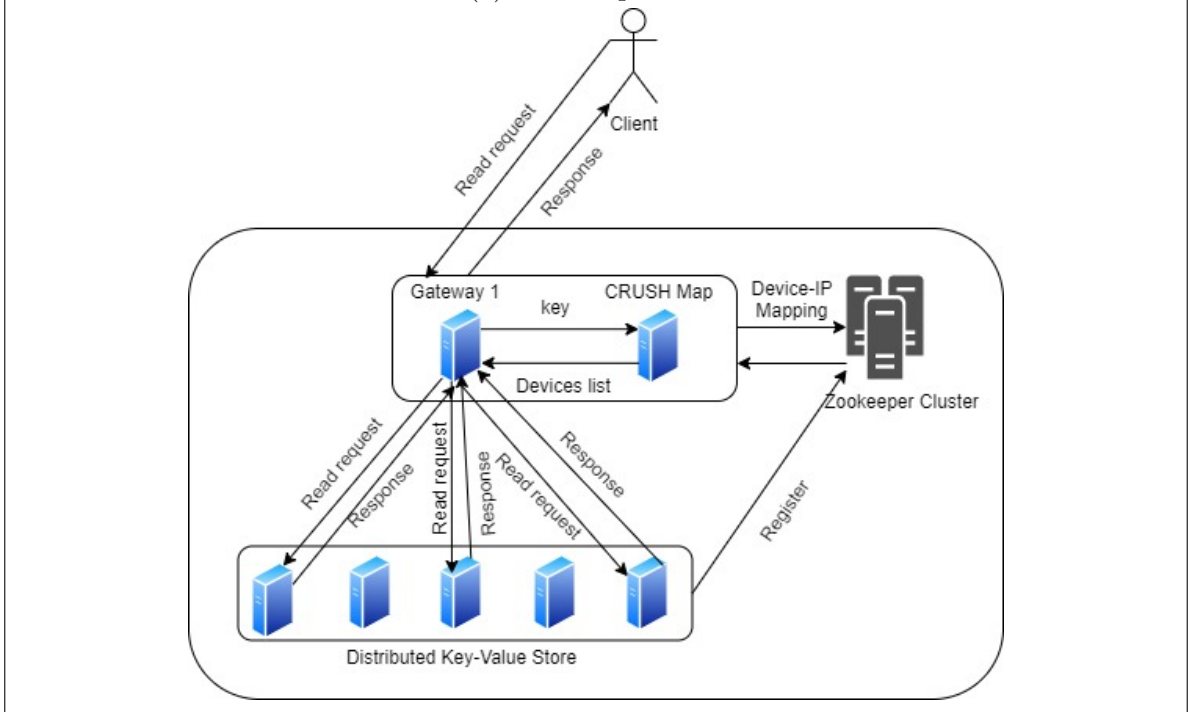
Column Name	Data Type	Description
pid	int	Product id (Primary Key)
pname	varchar	Product name
price	int	Price of the product
qty	int	Number available in inventory

3.2.2 Table Name: Customer

Column Name	Data Type	Description
cid	int	Customer id
cname	varchar	Customer name



(a) Write Operation



(b) Read Repair

Figure 2: Read (List) and Write (Add, Delete) Operations on the Key-Value Store