Cloud Cart (C2): Planned Design

Fábio Morais
up202008052@edu.fe.up.pt
Faculdade de Engenharia da Universidade do Porto

Francisco Prada
up202004646@edu.fe.up.pt
Faculdade de Engenharia da Universidade do Porto

Guilherme Sequeira
up202004648@edu.fe.up.pt
Faculdade de Engenharia da Universidade do Porto

Pedro Ramalho
up202004715@edu.fe.up.pt
Faculdade de Engenharia da Universidade do Porto

ABSTRACT

This article details the planned design for our shopping list platform, Cloud Cart (C2), developed as a part of the Large Scale Distributed Systems curricular unit. The project aims to explore the creation of a local-first application, which has code that runs in the user device and can persist data locally, and also has a cloud component to share data among users and provide backup storage.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; **Availability**.

KEYWORDS

Distributed Systems, Cloud Computing, Availability, CRDT

1 INTRODUCTION

In this study we aim to detail our planned design for C2, a local-first shopping list platform that emphasizes local storage and processing of data, while concurrently offering synchronization and collaboration capabilities. The platform integrates a cloud component for data sharing and backup storage, catering to a user base of potentially millions. Our architectural design closely follows that of Amazon's Dynamo[1], a renowned distributed system, and incorporates Conflict-free Replicated Data Types (CRDT's)[2] to handle data consistency conflicts.

Our primary focus during the development of this application is to ensure availability, which we prioritize over strict consistency, acknowledging the trade-offs described by the CAP theorem.

The article is structured as follows: Section 2 describes the application's functionalities, section 3 describes the CRDT's implementation and operations, section 4 describes the cloud-side architecture and section 5 concludes the article.

2 APPLICATION

Users should be able to create a new shopping list, generating an identifier which uniquely identifies that list. This unique identifier can then be used as a means for sharing the list with other users, allowing them to edit its contents.

When it comes to editing a shopping list, users may add an item, by specifying its name, description, and quantity. For items already present in the list, users may adjust their quantity, increasing or decreasing it as needed.

The application should also support the upload of a user's locally stored shopping list to the cloud. Users may also download the latest version of their shopping list from the cloud, incorporating any changes made by other users who have access to the same list.

3 CRDT

To address data versioning and consistency, an `ORMap` that maps item names (strings) to a `BCounter` will be implemented, where the counter represents the item's quantity.

The `ORMap`, as a map-type CRDT should be able to support a number of operations:

(1) Read - As a map, it should be capable of returning a list of the keys present in the map as well as their counter's value.
(2) Item addition - As a map, it should be able to instantiate a default value and associate it with a new key.
(3) Item removal - As a map, it should be able to delete a key and, by consequence, its value.
(4) Join - As a CRDT, it should be capable of reconciling branching versions of itself, incorporating changes from both branches and arriving at a state while satisfying causal and sequential consistency.
(5) Item increment - It should be able capable of incrementing a given item's counter state.
(6) Item decrement - It should be able capable of decrementing a given item's counter state.

The `BCounter` should also be able to support a number of operations, as well as ensure some special properties:

(1) Read - It should be able of returning the current state of the counter.
(2) Increment - It should be able of incrementing its value.
(3) Decrement - It should be able of decrementing its value.
(4) Join - It should be able of reconciling branching versions of itself, ensuring the same consistencies as the `ORMap`.
(5) Non-Negativity constraint - The counter's value should never be allowed to fall below zero.

4 CLOUD ARCHITECTURE

The cloud-side architecture is intentionally designed to be decentralized, mitigating single points of failure by distributing data and control across independent servers. This approach also allows for improved horizontal scaling, enhancing the system's capacity to handle increased demands by adding more resources (servers).

4.1 Key-Value Store

Given the simplicity of read operations (fetching a shopping list by its URL), traditional relational database models introduce a large amount of unnecessary overhead and complexity to the system. For

this reason, a simple and fast key-value database is ideal, where a list's URL is the key and the CRDT representation of the shopping list is the value.

4.2 Partitioning

As previously mentioned, each data item is associated with a unique key. To distribute data evenly we follow the approach used by Amazon's Dynamo, a variant of consistent hashing. Each key is mapped to a continuous space of hash values where the highest wraps around the lowest (forming a ring whose hash values grow clockwise). Each server in our application is assigned to multiple randomly assigned points within the ring (tokens). The number of tokens attributed to a server scales linearly with the server's capacity, to account for differences in processing capacity between servers. To determine which server is responsible for a given shopping list, we find the first token that is higher than the key's hash value.

4.3 Replication

To ensure data persistence even in the presence of server failures, the data is replicated in multiple servers. The responsible server for the data, described in the previous section, is also charged with the task of replicating the data. To this effect, it iterates through the existing tokens to find the $N - 1$ successor tokens (in a clockwise fashion) that belong to distinct servers (including itself). Then, it sends a write request to the identified servers. Further logic is necessary to account for server failure, but it is omitted here for the sake of brevity.

4.4 Supported Operations

The architecture should support two operations:

- `read(key)` - A client may request to download the state of a list present in the cloud. Upon the request, the load balancer will forward the request to a server at random. The selected server will then issue a `read` request to the servers within the scope of data replication and merge the results, if needed, via the CRTDs' conflict resolution capabilities. The logic needed to account for server failure is described in the next section. Finally, the load balancer will be notified about the operation's success and redirect the read value to the client.
- `write(key, value)` - A client may upload its local state of a list to the cloud. Upon the request, a similar procedure is employed by the load balancer, however, if the chosen server is not within the scope of the data replication for that key, it will forward the request to one of the servers within the scope, chosen randomly between them. After a suitable server receives the request, it merges the received data with its own internal representation of the list and issues a `write` request to the other servers as described in the data replication section, as well as sending a `write confirmation` back to the load balancer. Therefore, as long as the request is able to successfully reach a suitable server, the write is deemed a *success*. This means that, in Dynamo's terms, W is always equal to 1.

4.5 Failure Detection/Handling

When a server performs a `read` request, it awaits for confirmation from the nodes for a given time, upon which it counts the number of confirmations received. The `read` operation may fail depending on this number, as specified above.

As for `write` requests, these will only fail when the load balancer awaits a confirmation for a given time but fails to receive it.

Note that although servers expect a confirmation when handling `read` and `write` requests, nothing is done about the servers that do not respond, as permanent failures tend to be rare. After a permanent failure of a server, it is expected to be removed manually by a system administrator. Details are specified in section 4.7.

4.6 Data Versioning

All data versioning pertaining to the shopping lists is handled by CRDT's. As for the tracking of available nodes in the network, the data shared between the nodes through the gossip-like system will implement a last-writer-wins policy.

4.7 Adding/Removing a Server

The addition and removal of a server is done manually by a system administrator by a separate service.

When adding a server, this external service will connect to one of a statically configured set of `seed` servers that are responsible for holding a correct view of server membership and request its current view. It will then calculate new tokens to be added and send the updated membership view to all of the `seed` servers. The added server will then request the membership view from a random `seed` server and calculate which tokens it will need to *steal* from other servers. It will hereupon contact the aforementioned servers and request the values of the keys it is now responsible for. The contacted servers will have their membership view updated at this time, while the other servers will have their view updated eventually via the gossip-like system.

When removing a server, the external server will connect to a random `seed` server and request the current membership view. After verifying that the server to be removed is indeed present, it updates the membership view and then updates all `seed` servers. The rest of the servers will eventually be updated via the gossip-like system.

Note that as per Dynamo's design, this system is ideal for systems with hundreds of servers. For systems with thousands or tens of thousands of servers the membership view would take a non trivial amount of time to be transmitted between the servers through the gossip-like system, which is expected to run in conjunction with `read` and `write` requests, that is, these requests also carry with them the current membership view of the server sending the request.

REFERENCES

[1] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. *Amazon.com* (2007).

[2] Nuno Preguiça, Carlos Baquero, and Marc Shapiro. 2018. Conflict-free Replicated Data Types. *CRDT.tech* (2018).