Local-First Shopping List Application

SDLE - G16

Problem description

The goal of the project is to develop an application that allows users to manage their shopping lists. Since the application is local-first, the user must be able to freely store and update their lists locally. Users can also share their lists with others, which is supported by a cloud component.

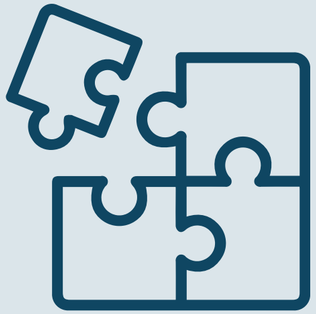
The supported operations for users are:

- Adding an item to the list
- Removing an item from the list
- Updating the desired amount of an item
- Updating the bought amount of an item
- Deleting a list (locally or locally+cloud)

Contents

Front-End

- Bridges the contents with the clients.
- **Local First** and **Testing** Implementations



CRDTs

- Implemented with **PN counters** to track the quantities of products;
- Mixed with **Optimized Observed Remove Set** for key removal support

ZMQ

- Used different **ZMQ** socket types
- Justification of each socket combination found in the project

Cloud Architecture

- Cloud Components
- **Consistent Hashing**
- **Membership Change**
- **Fault Tolerance**
- Limitations

Front-End

Program Flow

The program has 3 interchangeable states:

Login — User types the name of his account in order to enter the program

Start — A client must create or fetch a shopping list, given it's ID, so he can interact with it

Shopping List — The client can make modifications to his list, such as adding, removing, buying products and pushing, pulling the list

List State

Each Shopping List of a client has contents associated with it, such as the **list name**, **crdt** and the **socket** that he uses to comunicate with the cloud

Local First

The states of the clients are being periodically stored locally in a json file and only client made actions such as **fetch**, **push**, **pull** and **list deletion** require cloud connection

Testing

User Testing

A local json file with commands can be used in the program, in order to test certain cases more quickly.

Debug Mode

A **Debug Mode** can be toggled on and off. Once inside the debug mode, the logs of the large scale testing simulations can be analysed one by one, keeping track of the **actions** and **crdt state** along the way.

Large Scale Testing Mode

A **Large Scale Testing Mode** can be deployed, on which several clients are created and they perform actions according to their **taskManager** commands. The **taskManager** provides semi-random tasks for the simulated clients to perform, such as fetching a shopping list, adding and removing items from the shopping list. The results of these actions are logged in txt files, ordered by the time on which they were performed. There are **client logs**, **list logs** and a **general log**. At the end, the program iterates through each list's actions one by one in order and validates the results, indicating the actions that failed

CRDTs

When sharing a list with another user, or simply when making changes locally before pushing them to the cloud, multiple versions of the list will exist. The use of Conflict-free Replicated Data Types ensures that these versions can be merged without causing conflicts with regards to the operations made by different users.

Video Segment 1 (0:00-0:43)

After starting a client, we create a list and add some items to it. Using the link’s unique ID on another client, we can fetch it through the cloud.

CRDTs (cont.)

PNCounters:

To keep track of the amounts for any given item, we use a pair of PNCounters. Each one of these counters has a pair of grow-only counters, one for increments and one for decrements. Each client is assigned their own counter, such that, to read the current amount for an item, we use the sum of that item's counters. The pair of counters track two amounts, "to be bought" and "bought", for a given product..

Video Segment 2 (0:43-1:07)

Using the list from the previous segment, we make concurrent changes on two clients. After synchronizing with the cloud, we can see that the changes from both clients were kept.

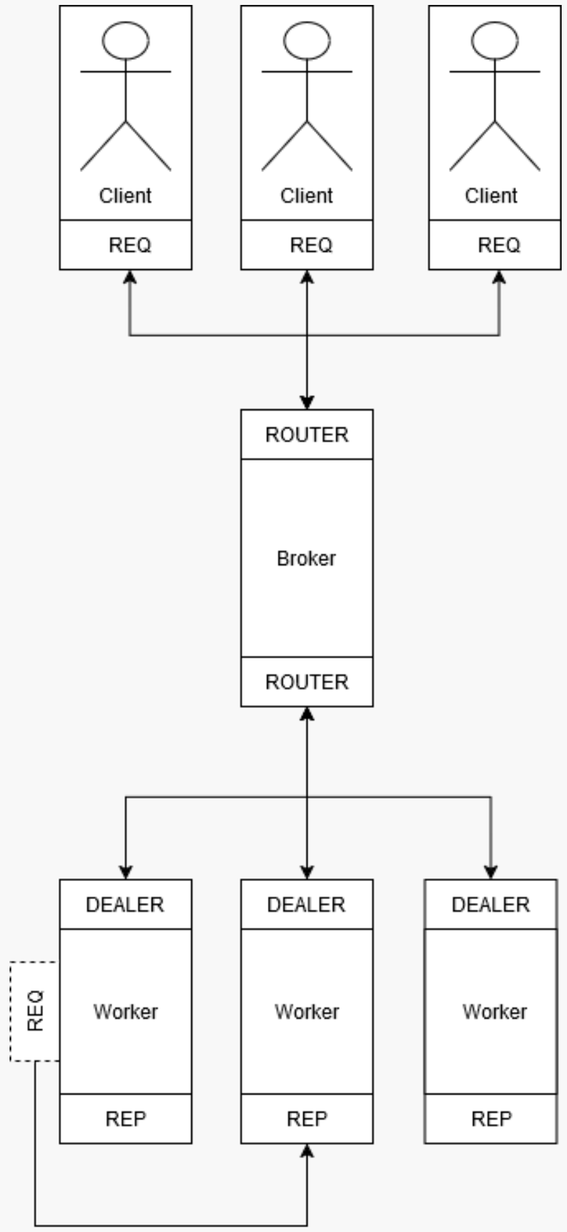
ORSet:

The ORSet is used as an auxiliary structure in order to keep track of what items exist on the list. For all intents and purposes, the combination of this set and the counters forms a map. It isn't explicitly programmed that way due to the set being added after the counters. Using it as a "secondary" structure allowed us to fully use its capabilities without changing the use of the counters.

ZMQ

- As per the project's requirements, we used **ZMQ** for messaging. In our case, we used the **Node.js** bindings. Since these bindings include type declarations, we were able to develop the project in **TypeScript**.
- Taking into account our needs for the project, we used multiple socket combinations:
  - **REQ-REP:** The simplest combination, which allows us to easily send message from point A to point B.
  - **REQ-ROUTER:** The purpose of this combination is to allow the message to be forwarded to one of multiple services, while ensuring that the reply can be routed back to the correct **REQ** socket.
  - **ROUTER-DEALER:** This combination has little restrictions, and allows us to send messages asynchronously without waiting for the send/receive pattern we find when using **REQ** and/or **REP** sockets.

Cloud



Client:

- Sends its requests to the broker through a **REQ** socket.
- The socket is configured with a timeout, to avoid indefinite blocking in case the service is down or a message is lost.

Broker:

- To implement sharding, we must be able to forward our requests to specific workers. Using a **ROUTER** socket in the “backend” of the broker allows us to do this.

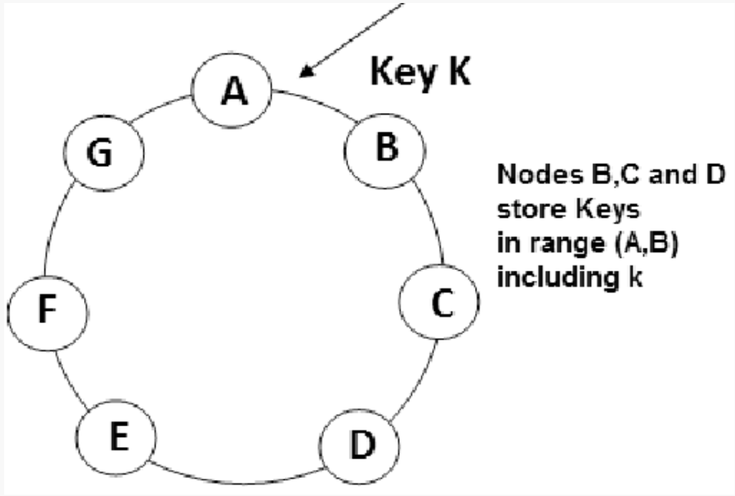
Worker:

- Although we use a **DEALER** socket to communicate with the broker, the way we use it is very similar to what we’d do with a **REQ** socket.
- An ephemeral **REQ** socket and a persistent **REP** socket is used for communication between workers.

Consistent Hashing

Data structure:
To assign lists to workers, we use a consistent hashing ring. We assign 5 virtual nodes to each worker in order to better split the load between them.

Replication:
To increase availability and improve fault tolerance, each list is replicated across three nodes. After a write, the changes are propagated across replicas.



Ring sharing:
Both the broker and the workers have knowledge of the ring's state. With each request sent from broker to worker, a list of the current worker's identities is sent.

Video Segment 2
(1:07-1:34)

When creating a new list in the client used previously, after pushing said list to the cloud, we can see that the request was routed to a different server than previous requests (first UUID in the server's logs)

Fault Tolerance

Our system is resilient with regards to multiple fault scenarios:

- If a client's request isn't properly delivered to the broker, or if for some reason it doesn't receive a reply, the timeout ensures that the client can recover and try again, if the user wants to.
- If a worker fails, the next request sent to that worker will result in a timeout. Until that worker recovers, no further requests will be sent to it. If a client retries the request, it will be forwarded to a different worker.
- To ensure that a worker doesn't wait indefinitely while the broker considers them to be "busy", it periodically sends a message to the broker while waiting, signaling its status (hence the need for a **DEALER** socket).
- Issues with the **REQ** sockets used for worker-to-worker communication will not persist, since the sockets are ephemeral and used only for one request.

Fault Tolerance (cont.)

- The broker and each worker periodically save their state to disk in a JSON file. This means that individual workers can recover in case of failure, and that, if the broker fails, the entire system (broker and workers) can be reinitialized in its previous state.

Video Segment 3
(1:34-2:12)

For the video, we have configured the workers in order to induce a failure after 15 requests. First, we send some requests to reach that amount. As we can see, at a certain point, we get a timeout on the request. In subsequent requests, we can see in the server logs that the request is routed to a different server. A few seconds later, when the server recovers, the requests start being routed to it again.

Membership Change

We implemented two processes that allow us to dynamically change the nodes present in the ring, through addition or removal of a worker. These requests are sent through the broker.

- In the case of removal, the worker in question redistributes its lists, one by one, to their new owners.
- In the case of addition, the new node goes through the existing nodes, announcing its existence. Those nodes gather the lists that should be redistributed to the new node, and send them.

Video Segment 4
(2:12-2:38)

In this case, we trigger a removal of the node that was receiving the client's requests. After the process is complete, the requests are routed to a different server, but this time they don't return to the initial server

Video Segment 5
(2:12-3:14)

Here, we trigger multiple node additions, until a new node is placed in such a way that the client's requests change destination. Once again, this can be shown through the first UUID shown in each line of the server's logs.

Limitations

We are aware of some limitations in our project:

- The broker is a single point of failure. If it shuts down, the whole system must be restarted. However, since the state is preserved on disk, it can easily be restored to its previous state.
- Sharing of the ring's participants could be a bit more efficient, for example by sending only the alterations to the ring instead of all node identities.
- The membership change operations are complex when compared to client requests, and are particularly prone to failure and to disrupt the system if something happens to the changed node during the operation.

Video Segment 6
(3:14-3:52)

In the last segment, we show how the system can be recovered after a critical failure (the broker being shut down). When restarting the broker, it will load its previous state from the file it was previously writing to, and it proceeds to restart the workers as well, which read their state from their own files.

References

- Local-first: https://www.inkandswitch.com/local-first/
- CRDTs: https://crdt.tech/papers.html
- Dynamo: https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
- Node.js ZMQ Documentation: http://zeromq.github.io/zeromq.js/
- Hashring package: https://www.npmjs.com/package/hashring

Thank you