U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Shopping Lists on the Cloud

A Scalable, Highly Available Distributed System

Emanuel Gestosa, José Gaspar, Lucas Sousa
SDLE, MEIC - FEUP
December 9, 2023

Outline

Introduction

Introduction

We want to build a system that can manage shopping lists from different users. Our system should follow the following guidelines:

- Local-first application.
  - Store data locally first and on the cloud.
- Share lists between clients.
- Version conflicts solved with CRDTs.
- High availability.

Architeture

Architecture

- **Local-first application**: Provides an interface between the client and the cloud, but can also be accessed while offline.
- **Proxy**: Gateway between the local application and the rest of the cloud infrastructure.
- **Node**: Deals with requests to read or update shopping lists, as well as data replication.
- **Coordinator**: A special kind of Node, that receives requests directly from the client, and may use other Nodes to store/retrieve replicas.

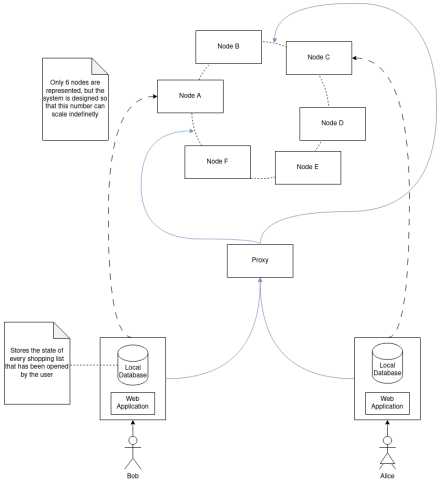

Figure: High-level overview of the design

Implementation

Implementation

Communication

- Communication is done via HTTP and with ZeroMQ.
- We have communication between the following entities:
  - Proxy - Local application: to inform the local application about the Coordinator of the desired list.
  - Proxy - Node: to inform the Node about its preference list and for a Node to join/leave the circle.
  - Node - Node: to manage replicas.
  - Node - Local application: to retrieve/update lists from the cloud.

Implementation

Node preference lists

- A Node joins the circle by sending an HTTP POST request to the Proxy.
- The Proxy updates the Node circle, and the preference lists of the Nodes.
- The Proxy publishes the updated preference lists.
- Every Node subscribes to updates to their preference lists, and checks for updates every time they need to act as Coordinator.
- A Coordinator will use its preference list to know which Nodes it should use for consulting or updating replicas.

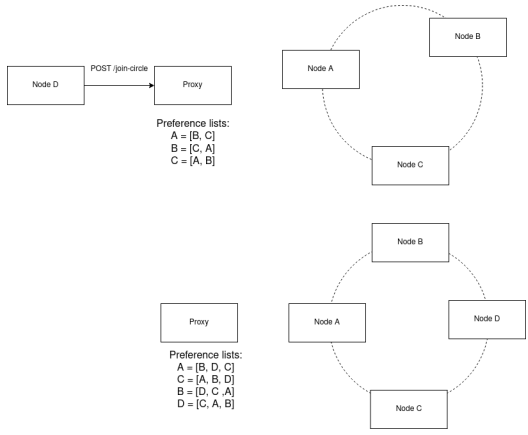

Figure: New Node joining circle

Implementation

Replication

Quorum strategy

- Coordinator has it's own preference list.
- The Coordinator waits for a request from the client.
- if PUT:
  - Coordinator stores information in its database.
  - Sends request for all nodes in its preference list for replica creation.
  - Waits for responses.
  - PUT always succeeds in ensuring high availability.
- if GET:
  - Coordinator sends requests for all nodes in its preference list for getting existent replicas.
  - Waits for responses.
  - GET only succeeds if at least one replica is received.
  - Merge replicas.
  - Return merged replicas to the Client.

Implementation

Resolution of version conflicts - Conflict-free Replicated Data Types (CRDTs)

Operation-Based CRDTs were used:

- All operations are saved as an operation entry, with an id hash, item name, type field ("ADD", "REMOVE"), and a count field.
- Both the add and remove operations are treated the same way, the only difference is how they are displayed in the GUI.
- Store operations over the list.
  - Add N element to a list.
  - Remove N element from a list.
- Merge versions function:
  - Used GSet strategy.
    - Node merging by adding all operations to a set, thereby keeping one entry of every operation.
- Inspired by git commits.

Implementation

Local

- Django web app, assumes that the Django Backend and Frontend are running on the same machine, emulating a local application.
- On page load, queries the proxy for the IP address and port of the appropriate node.
- Subsequently queries the node, sending its locally saved items and retrieving the merge of all items.
- Sends a PUT request when making changes and a GET request when retrieving from the server.
- At 5-second intervals polls the nodes for changes.
- Saves all changes locally, regardless of node connection.

Implementation

Local

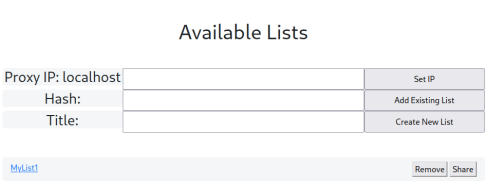

Figure: Main Page with Lists

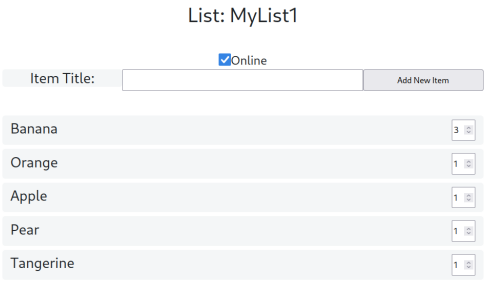

Figure: List Page with Items

Implementation

What if something goes wrong?

- What if a node is not responsive?
  - The Coordinator knows it upon GET or PUT requests.
  - Informs proxy about it and proxy disconnects it from the circle.
  - The system works fine because of the replicas on other nodes from the coordinator's preference list.
- Failure points:
  - Proxy failure.
  - All nodes from a preference list are not responsive.

Discussion

Discussion

CAP Theorem

- On our project, we wanted to have a highly avaliable system.
- That is why we allow the PUT requests to always succeed, and GET requests succeed with just one replica.
- To achieve this we cannot guarantee that replicas are consistent all the time, but on the other hand, if a node goes down, its information should still be saved somewhere else, and thus always available.
- However, the synchronization of replicas is done upon GET requests.

Conclusion

Conclusion

- We fulfilled our goal of achieving a highly available distributed system.
- Users can create and share lists, and work on them at the same time.
- Conflicts are always resolved, and data loss is highly unlikely.
- Possible improvements:
  - Addition of backup proxies to avoid the system being down on the chance the Proxy fails.
  - Local application currently uses polling, an improvement would be to use a pub/sub approach.