



Shopping List Manager

Local First Progressive Web App

SDLE Practical Assignment

Daniel Rodrigues	-	UP202006562
Martim Videira	-	UP202006289
Miguel Silva	-	UP202007972
Abílio Epalanga	-	UP202300492



System Requirements






While developing our project, we made it with the following requirements in mind:

- ❑ **Document Sharing:** every user can open and edit lists created by other users as long as they have the lists' ids.
- ❑ **Offline Work:** users can modify lists **locally**, without being connected to the server. When they finally establish a connection, their changes are merged with the current version stored in the server.
- ❑ **Efficient Storing:** Servers provide high **availability** via replication of data and deal with the process of merging changes swiftly.



System Implementation

Our system can be decomposed into three dependency-inducing parts:

- ❑ **Client** – Main Web App developed using Svelte and Bootstrap  
- ❑ **Intermediary / Node Servers** – Created using node.js library express.js  
- ❑ **Reverse Proxy Server** – Configured with NginX, balances request load between servers 

Every step needed to install the app has been documented and can be found on the project's **README.md** file. Be aware you will be needing `npm` to setup the project.



System Description – Client

There's a client app for both **end users** and **administrators**:

- ❑ Users' client will be used to manage their lists and apply operations on top of them.
- ❑ Administrators' client will be used to manage the servers' status and make any needed changes.

The client gives permission to the browser to write their changes in the chosen directory. Whenever the client receives an update from the server, it also changes their local files that store each list.



System Description – Client

The process of syncing each client's changes with the server is triggered by the client – **it is a manual operation.**

As long as the client has the **link** to a given list – they can access it, change it, certain that when they sync with the server, their changes will be taken into account.

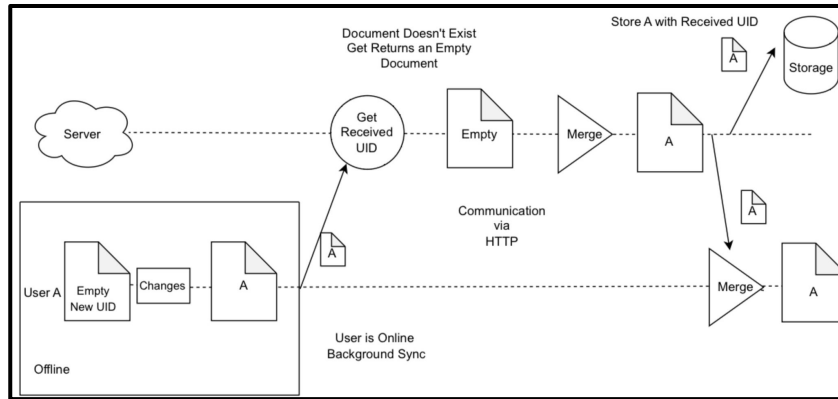
What if it tries to access a list while not connected to the server?

- ❑ A new empty list is created, one that will eventually be merged with the one existing on the server.



System Description – **Syncing**

The focus in the architecture's design remains the same: allow multiple users to work **offline**, in **parallel**, with the certainty that their changes will not result in conflicts when merged.





System Description – Lists

Lists are implemented as **BAWMap** CRDTs (Basic Add-Wins Map). A **BAWMap** is composed of a **Add-Wins Set** of the keys and a Map that links these keys to CRDTs.

In our design:

- ❑ Each Shopping List is a **BAWMap**.
- ❑ Its keys are the names of the items in the list.
- ❑ Its values are **BAWMaps** that match the **desired** and **purchased** quantities to **PNCounters**.



System Description - Cloud

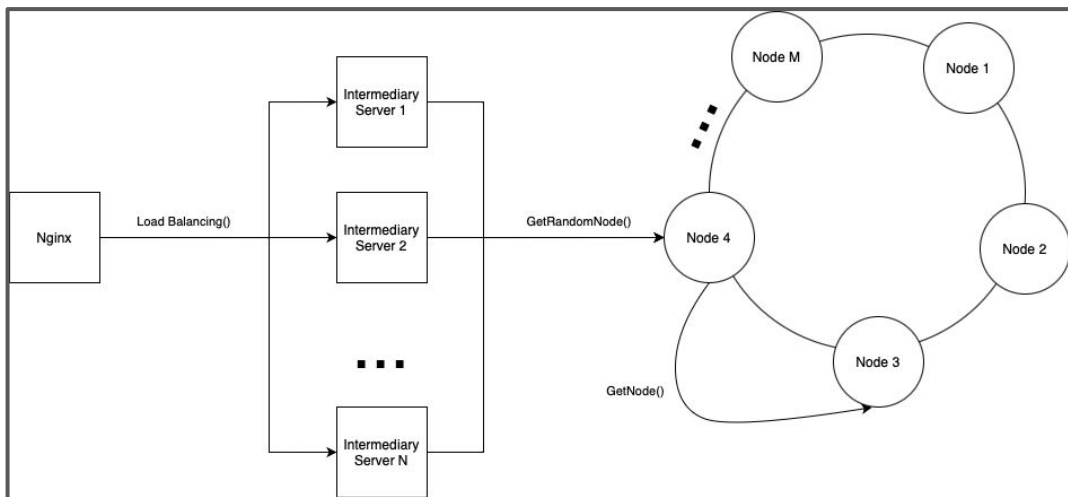
When it comes to our Server-Side implementation, the architecture goes as follow:

- ❑ The request reaches the **Nginx Server**, which is responsible for load managing incoming requests between the replicated intermediary servers.
- ❑ When the request reaches the chosen intermediary server, it selects a random node from the list of available servers within the **Ring** to process the request.
- ❑ Then, the node must either deal with the request or redirect it to the intended instance in case it is not responsible for the request it was given by the random selector.



System Description - Cloud

The following image depicts this flow in more detail:





System Description - Cloud

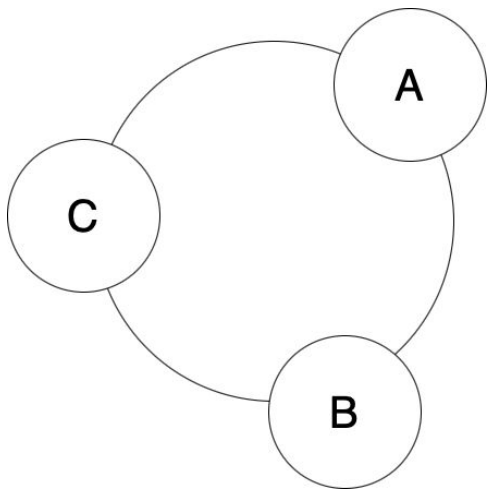
Having an additional Reverse Proxy Server in the Nginx instance improves this architecture in three aspects:

- ❑ Instead of having the client store a list of possible endpoints when replicating intermediary servers, you only need to **expose one port**.
- ❑ No need to update client-side in case of removal / addition of new servers.
- ❑ Removes the need for **load balancing** on client-side. Instead, lets Nginx do it very efficiently.



System Description – Replication

So how do we deal with failures in the end nodes that store the data? By replicating data in a **Ring**-like architecture while implementing **Consistent Hashing**:





System Description – Replication

To ensure a certain degree of replication, we must do the following:

- ❑ Each **node** (ring server) stores lists that are assigned to him based on the hash of their id.
- ❑ The successors of that node must also store a **replica** of those lists, ensuring the wished degree of replication.
- ❑ In case one of these nodes is unavailable, we should store these replicas temporarily in the next available successor, which will **hand them off** to the original node whenever it is available again.

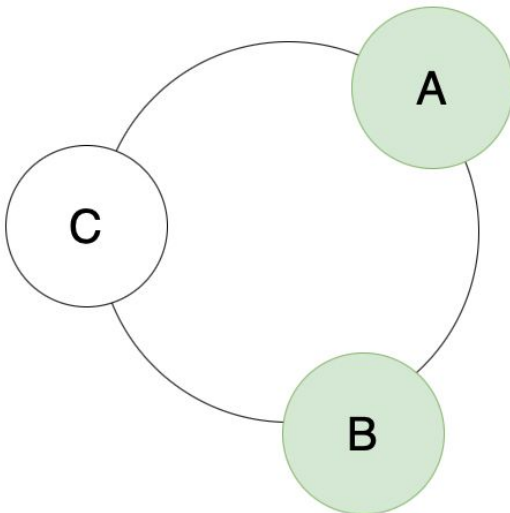


System Description – Replication

In case all nodes are available, with *replicationDegree* = 2:

Trying to store in Node A

 - means stored successfully


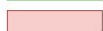


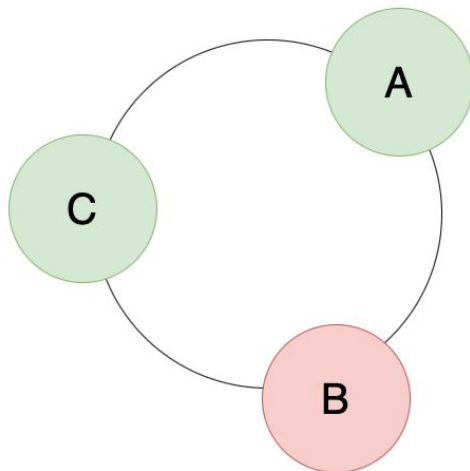


System Description – Replication

What if **B** fails? **C** receives the list knowing it should **hand it off** back to **A** and **B**

Trying to store in Node A

-  - means stored successfully
-  - means failed to store here





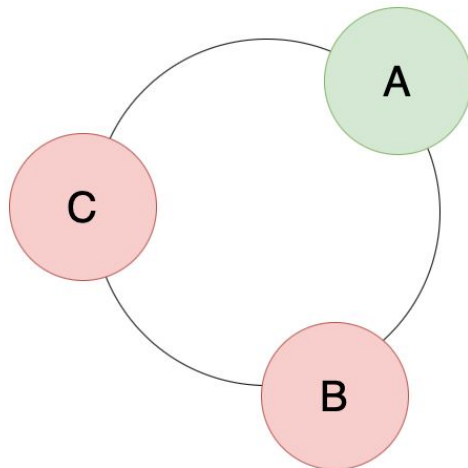


System Description – Replication

What if **B** and **C** fails? It only wrote the list on **A** but... when **B** becomes available again... **Mismatch!**

Trying to store in Node A

-  - means stored successfully
-  - means failed to store here





System Description – Replication

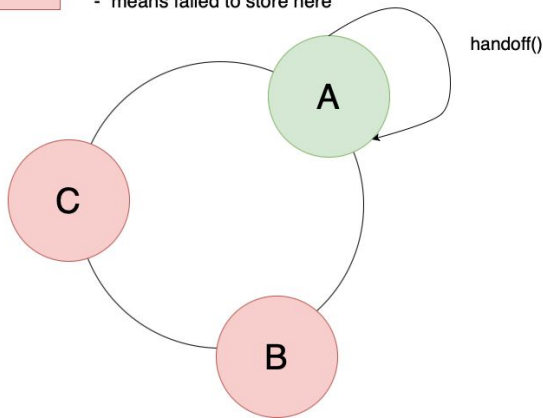
Solution! A writes in **its own handoff folder too**, so that it knows that data must be handed off to the successors once they become available again!

Trying to store in Node A



- means stored successfully

- means failed to store here



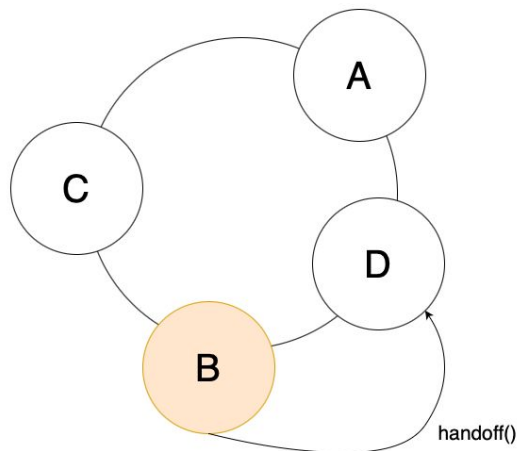


System Description – Replication

What if **D** is added to the system? **B** contains replicas of data from **A** that need to be moved to **D**, now that it is the immediate successor of **A**!

Adding Node D to the Ring

 - needs to hand off data





System Description – Gossip

But how does this communication happen? How does **B** get to know that **D** was added to the ring? To ensure all nodes **eventually** have the same knowledge on the ring, we implemented **Gossip Protocol**:

- ❑ Whenever a node is added / removed from the ring, this will trigger a series of messages to **N** random nodes, letting them know of the change.
- ❑ This message consists in an **id** (unique per action triggered), an **action** (type of the operation, removal vs addition of a node) and a target (fellow node that is trying to communicate with).
- ❑ Using this unique id, the node knows how many times it has seen the update, helping it knowing when to stop propagating the same message.



System Description – Admin

Via the Client, an administrator can do one of three actions:

- ❑ Add a **new** node to the ring
- ❑ Remove node from the ring
- ❑ **Pause** node activities for **N** seconds

Note: Pausing nodes can be useful after introducing some changes in the system, to make sure some mechanisms are triggered and the system answers accordingly.



System Testing

We added some unit tests using Mocha, which helped test the application's server separately and find some bugs in our endpoints.



CRDTs were also tested extensively with the help of a **application developed in parallel** to the project that aims to test these data structures. Unit tests were also created.



Future Improvements

If there were **no time constraints**:

- ❑ Implement DNS Server
- ❑ Dockerize this project and host it in FEUP's Server.

Thank you!

