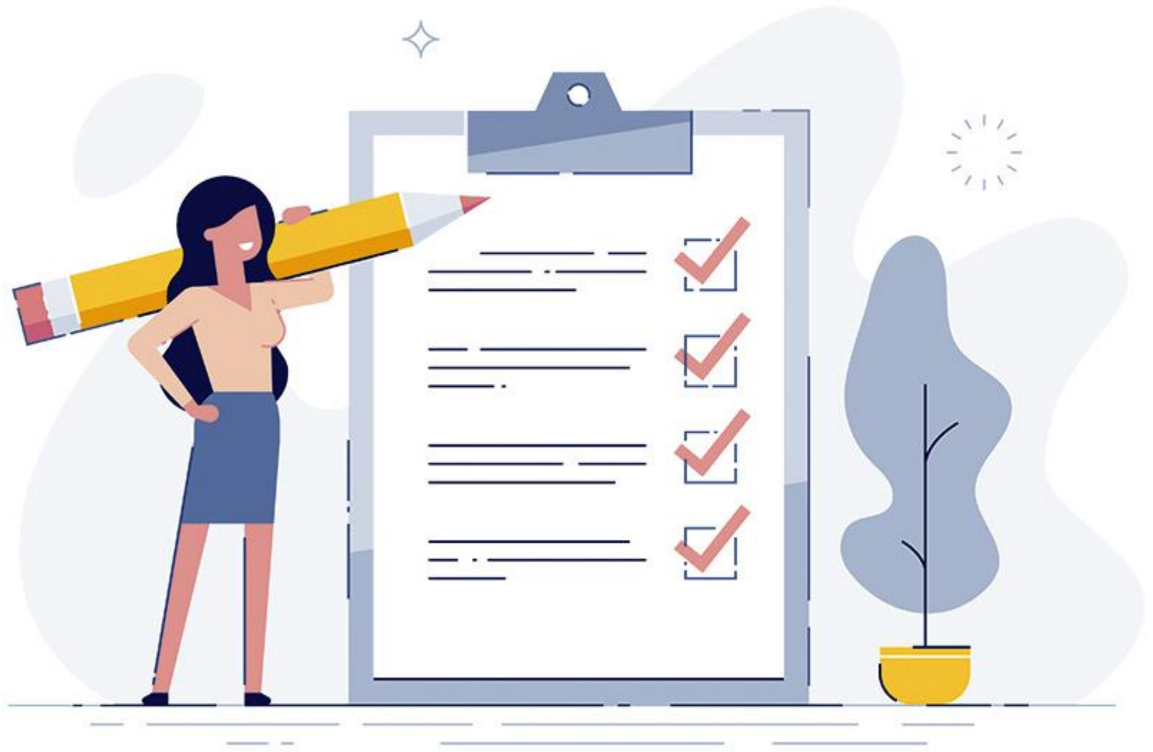
U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Local First Shopping List

LARGE SCALE DISTRIBUTED SYSTEMS

Afonso Baldo: up202004598
Isabel Amaral: up202006677
Milena Gouveia: up202008862
Pedro Gomes: up202006322



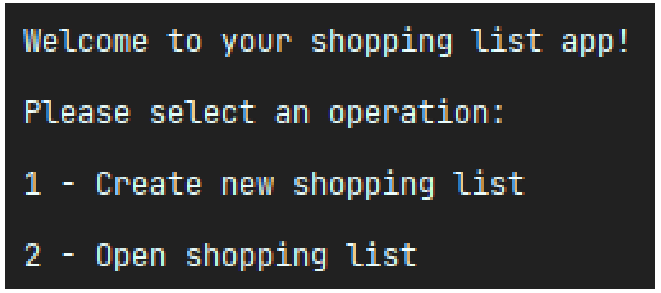
Requirements

The goal of the project is to develop an application that allows users to create shopping lists, add and remove items, including specifying the quantity to purchase.

The requirements are as follows:

- "local-first" application (users can use it without being connected to the internet).
- It must include cloud functionality to back up lists and enable sharing across different devices.
- Replicas in the cloud must be employed to ensure fault tolerance.
- It must be designed to support millions of clients.
- Conflict resolution between different versions of lists should be addressed using our own implementation of CRDTs.

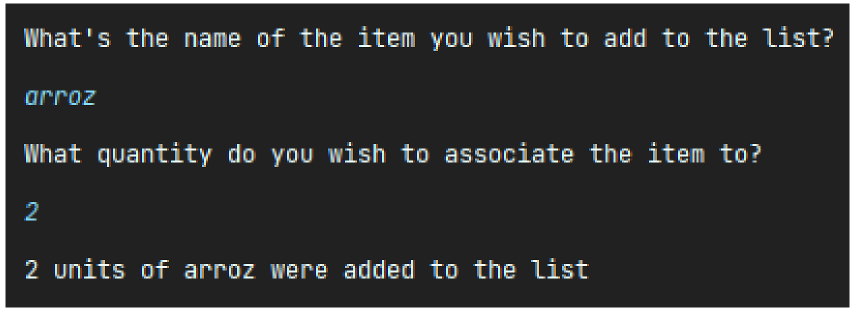
Client



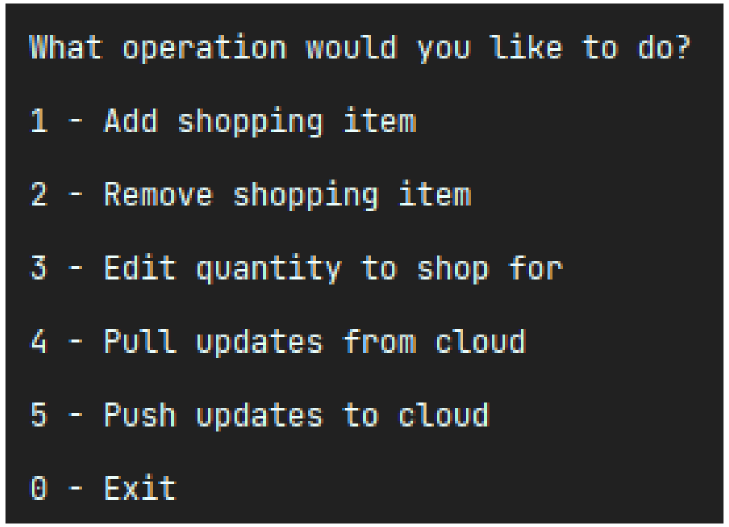
```
Welcome to your shopping list app!

Please select an operation:

1 - Create new shopping list

2 - Open shopping list
```

When running the application, the user has the option to:

- Create a new shopping list and give it a name. The unique link for future access is generated by appending a DateTime string to the user chosen name
- Open an already existing shopping list for editing



```
What's the name of the item you wish to add to the list?

arroz

What quantity do you wish to associate the item to?

2

2 units of arroz were added to the list
```

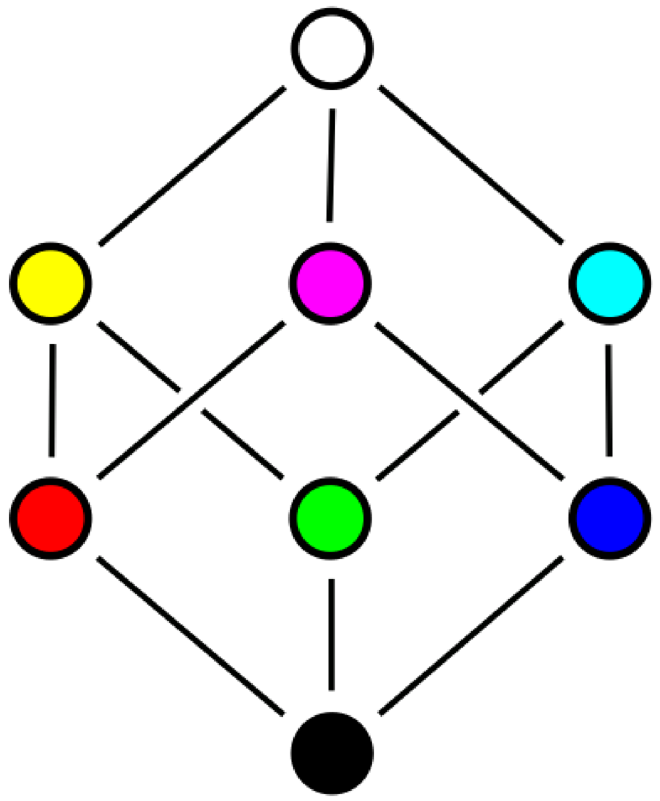


```
What operation would you like to do?

1 - Add shopping item

2 - Remove shopping item

3 - Edit quantity to shop for

4 - Pull updates from cloud

5 - Push updates to cloud

0 - Exit
```

Client

After selecting the shopping list, the following options are available:

- The editing related operations involve specifying an item and respective quantity
- The push and pull operations are used for cloud synchronization at the user's request



Shopping List CRDT

- For dealing with the conflicts that arise from more than one user editing the same shopping list, we implemented a shopping list level CRDT
- The shopping list itself is a String, Integer Map that associates to each item its quantity
- Internally the CRDT is implemented using:

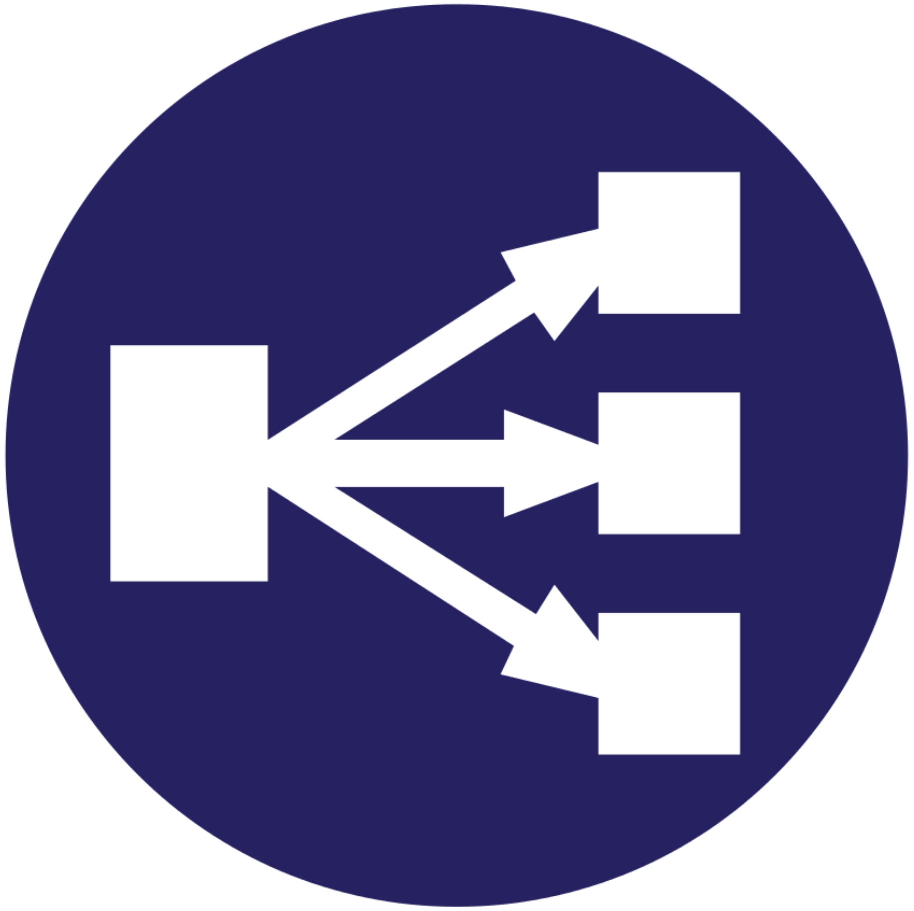
    - a String **AWORSet**, which keeps track of the items added and removed by different users to the shopping list

    - several **CCounters**, which keep track of the quantity of each item in the AWORSet

- Each of these have their own merge operation
- The merge of the upper level CRDT consists of, first merging the AWORSets and then, for each of the items in the merged set, merging their CCounters

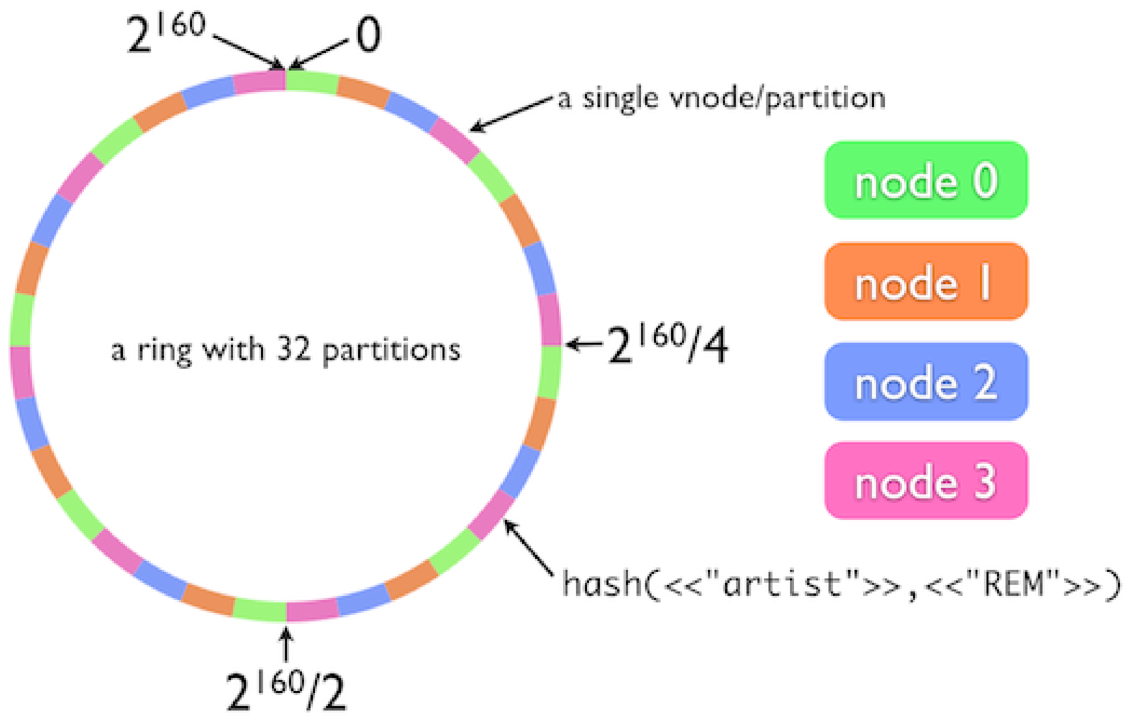
DNS Mock

- Load Balancers are connected to a DNS Mock
- DNS Mock is located at a fixed IP and port
- Clients interact with the DNS Mock for the load balancer endpoint
- Simulates a real-life DNS infrastructure
- Avoids single point of failure in the load balancer by replacing it with a single point of failure in the DNS mock. However, in real-life the DNS system is distributed
- Load balancer selection strategy: Round Robin (a more optimal choice would be based on geographic location)



Load Balancer

- Multiple Load Balancers
- Each knows about the token ring topology
- Performs consistent hashing given the shopping list ID and chooses one of the replicas randomly
- Receives the reply and forwards it to the client



Cloud - Overview

- Consistent hashing to distribute the load across storage hosts
- Virtual nodes to avoid load imbalance
- Given a key, the corresponding object is eventually stored in the following 3 (real) nodes.
- One of the first 3 nodes is chosen to immediately store the object
- Upon receiving a key, a node verifies if the key ranges that it is responsible for include that key's hash. If they don't, the node performs consistent hashing and forwards the request
- A synchronization algorithm runs periodically to synchronize common hash spaces

Cloud - Anti Entropy

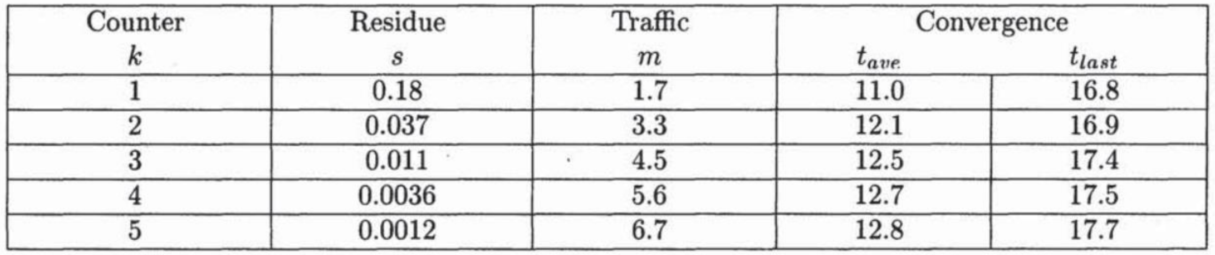
- Every 5 seconds, each node sends all the neighbors that it shares a key range with all the keys and corresponding shopping list hash in that range
- Upon receiving this message, the other node replies with the shopping lists it is storing whose hash are different (or non-existent) and its shopping lists that it did not receive
- The initial node receives the new shopping lists and stores them, merges the shopping lists which were not synchronized and sends both the different shopping lists and the ones the other node didn't have

Node Discovery

- Each node specifies a list of endpoints to which it connects and spreads the rumor that it has just joined
- Nodes whose endpoints are specified in the list receive the rumor and spread it to the rest of the nodes in the network
- Upon receiving the rumor, the other nodes connect to the new node
- An epidemic algorithm is used for spreading the rumor, with a periodic network topology synchronization between random nodes to deal with the probability of failure in the epidemic mechanism

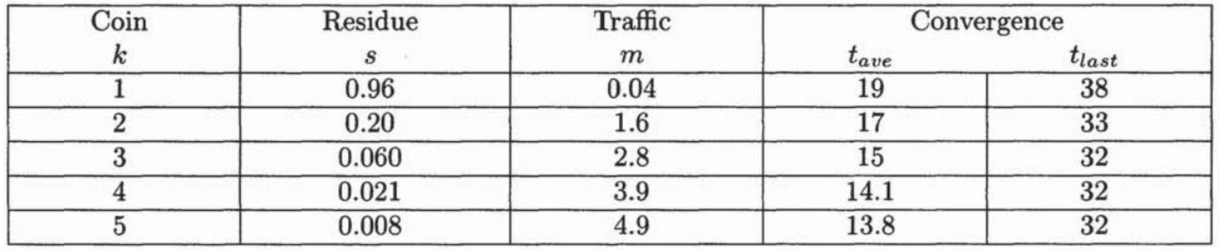
Epidemic Algorithm – Our choices

Table 1. Performance of an epidemic on 1000 sites using feedback and counters.



Counter k	Residue s	Traffic m	Convergence t_{ave}	t_{last}
1	0.18	1.7	11.0	16.8
2	0.037	3.3	12.1	16.9
3	0.011	4.5	12.5	17.4
4	0.0036	5.6	12.7	17.5
5	0.0012	6.7	12.8	17.7

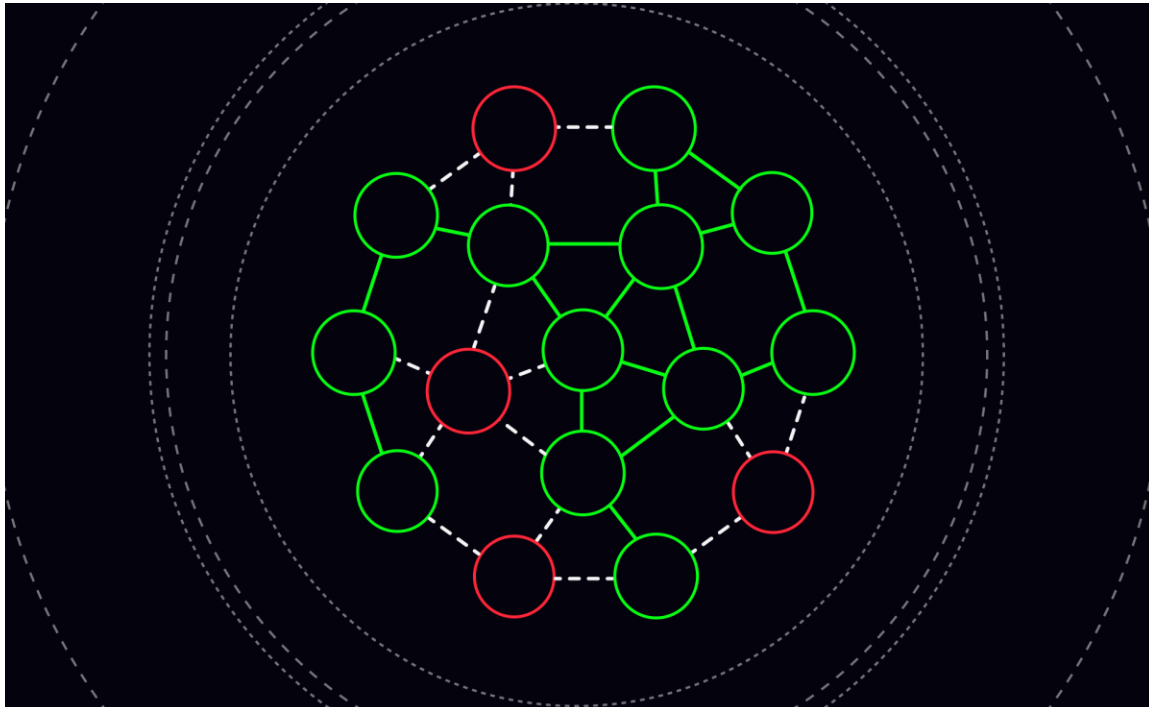
Table 2. Performance of an epidemic on 1000 sites using blind and coin.



Coin k	Residue s	Traffic m	Convergence t_{ave}	t_{last}
1	0.96	0.04	19	38
2	0.20	1.6	17	33
3	0.060	2.8	15	32
4	0.021	3.9	14.1	32
5	0.008	4.9	13.8	32

Reference: "Epidemic Algorithms for Replicated Database Maintenance", Palo Alto Research Center

- Rumor sent to two random neighbors every second
- Neighbor replies with an ACK, mentioning if it already knew about the rumor
- Node loses interest in rumor after 3 unnecessary contacts



Fault tolerance

- To detect a node failure, we catch exceptions thrown by the socket; if the socket is closed, an exception is raised.
- Upon detecting a node failure, we treat it as a temporary failure, marking it as inactive. We skip that node and when redirecting message to the correct ring node, and we don’t synchronize with that node during this period. At this time, we are operating with -1 replica for some key ranges.
- When a node with the same ID connects, we mark it as active and synchronize the lists using the anti-entropy algorithm.

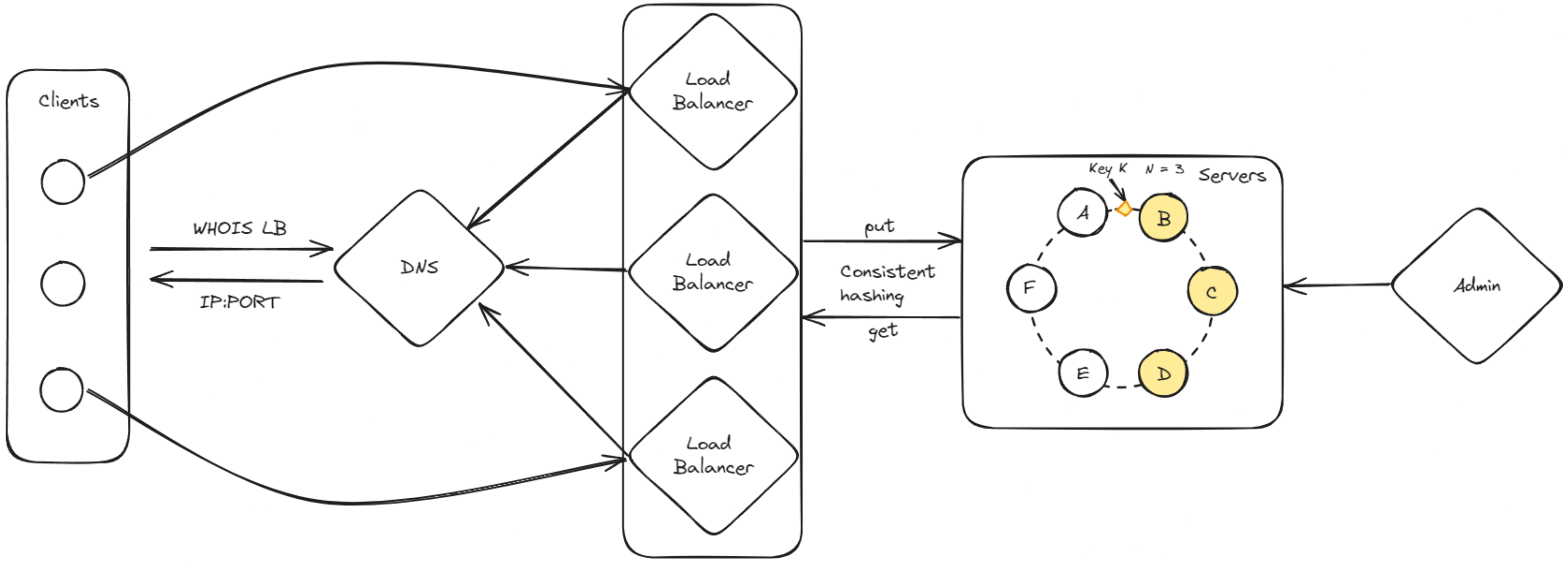
Adding a Ring Node

- Upon receiving a rumor about a new ring node, the recipient adds that node to its representation of the token ring
- Moving forward, the new ring node's virtual nodes' hashes are used when performing consistent hashing and replica synchronization
- Keys in intervals which the recipient is no longer responsible for are deleted locally

Removing a Ring Node - Admin

- Similarly to the load balancer, the admin knows about the token ring topology
- Upon the need to manually remove a node from the ring, the admin sends a remove message to all nodes
- From this point forward, keys will be redistributed to the correct nodes in the new ring topology due to the anti-entropy algorithm and the removed node's virtual nodes' hashes will no longer be considered when performing consistent hashing

Architecture overview



Architecture Limitations

- Every node in the cloud side of the system needs to know about all the other nodes, which is not very scalable. Utilizing a DHT algorithm like Chord would minimize state and enhance scalability, eliminating the need for nodes to be aware of every other node.

- The current system, where all nodes uphold persistent TCP sockets with each other, faces inherent limitations in scalability due to the finite number of available ports. Transitioning to a UDP-based approach would provide a more scalable solution by alleviating the constraints imposed by port limitations.