

Shopping Lists on the Cloud

Anu Atolagbe 202400090
Catarina Canelas 202103628
Dany Ferreira 202108799
Margarida Pinho 201704599





Project Description: The main goal of the project is to develop a local-first shopping list system which allow users to create and manage shopping lists. The system contains local data from clients and a cloud component for data sharing and backup storage.

System Overview: The system consists of three parts:

Local: Users can locally manage their shopping lists

Cloud Component: Backend infrastructure for data synchronization

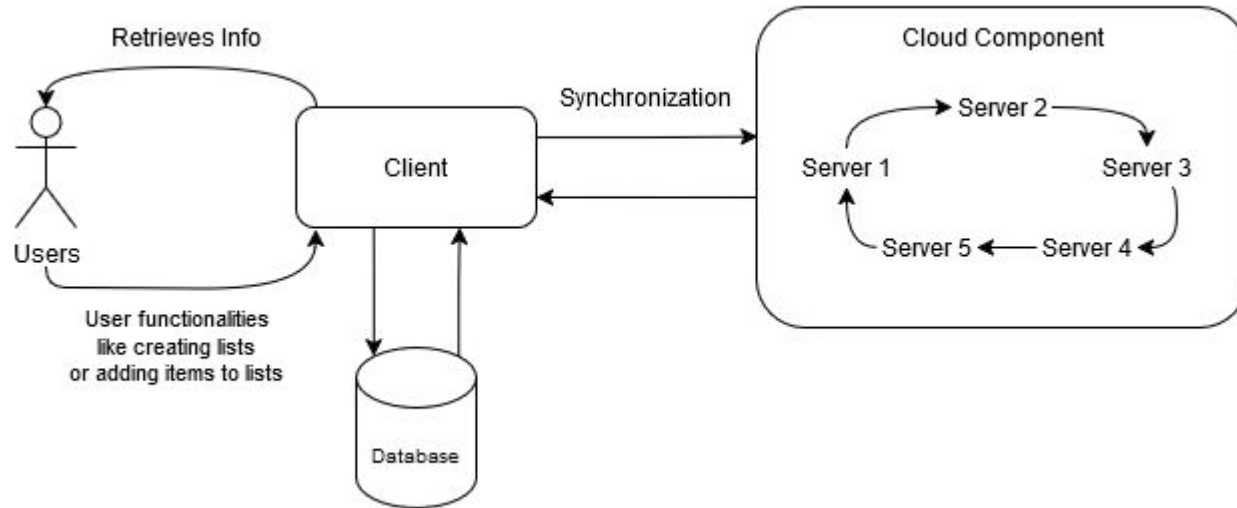
Data Storage: Each shopping list has an unique ID (URL) and both shopping lists and their items are storage in databases

Data Replication and Consistency:

Technology: The system was developed in Python, zmq was used for asynchronous messaging and UUID was used to generate unique identifiers across the system, SQLite was used for database management, threading was used for running background synchronization tasks and JSON was used for structured communication between the client and server.



Architecture





The system is a distributed client-server model that uses ZeroMQ DEALER-ROUTER messaging pattern for communication between the client, proxy, and worker servers.

Users: interact with the system through the Client to perform actions like:

- Creating shopping lists.
- Adding items to lists.
- Retrieving and viewing shopping lists/items.

Client: acts as the interface between the user and the system. It handles:

- User inputs and requests.
- Local database operations (offline support).
- Synchronization with the cloud servers for consistency.

Database: SQLite database managed by ShoppingListManager object. Maintains unsynchronized and synchronized copies of shopping lists and items.

Cloud Component: worker servers, each server processes client requests and synchronizes data. Proxy distributes client requests to workers using consistent hashing (via hashring)



Client

The client acts as a user-facing interface for interacting with the shopping list management system.

Each client gets an ID and a local database. This database contains all local shopping lists the user has created.

To manage the database we used a `ShoppingListManager` object that stores databases and performs all the tasks the client needs, like view lists, create a new list, create a new item, among others.

It allows offline functionalities by maintaining local copies of lists and their items.

```
manager = ShoppingListManager(db_path)
```



For each client an unique identity is generated using hash.

Users can interact with the system by selecting options from the menu to perform actions like viewing local or server-side lists, creating lists, or adding items.

Since this is a local-first system, when creating a new list, through the ShoppingListManager, its shopping list is locally storage.

```
name = input("Enter the name of the new list: ")
creator = input("Enter the creator's name of the new list: ")
try:
    new_list = manager.create_list(name, creator, client_id)
    print(f"\nNew list created locally: {new_list}")
```

```
def create_list(self, name, creator, client_id):
    url = str(uuid.uuid4())
    with self.lock:
        self.db.execute("INSERT INTO lists (url, name, creator, client_id) VALUES (?, ?, ?, ?)", (url, name, creator, client_id))
        self.db.commit()
    return {"url": url, "name": name, "creator": creator}
```



This action is then synchronized through requests sent to the server. If receiving an answer of success, it means that action was successfully synchronized.

For synchronization, the client connects to the server through a ZeroMQ DEALER socket. It connects to the server at a fixed address (`tcp://localhost:5555`).

It synchronizes changes (new lists, items) from the local database to the server and fetches updates from the server.

```
request = {"action": "sync_list", "list": new_list, "client_id": client_id}
synchronization_response(client, request)

manager.list_is_sync(new_list["url"])
```

It also marks the list or item as synchronized after synchronization.

For polling updated, a background thread periodically check and synchronize unsynced lists and items with the server.

```
threading.Thread(target=polling_and_sync, args=(client, manager, client_id), daemon=True).start()
```



Proxy

The proxy serves as a central intermediary between the client and the workers. Its main role is to route messages effectively using a **hash ring** for consistent hashing, ensuring balanced load distribution across workers.

Receives requests from clients and forwards them to the appropriate worker based on consistent hashing.

Routes responses from workers back to the originating clients.

Uses a **hash ring** to distribute the workload among available workers.

Ensures that requests related to the same client or data are routed consistently to the same worker.

Manages two sockets: a **ROUTER** socket on port 5555 for client communication and a **ROUTER** socket on port 5556 for worker communication



Router sockets for communication

```
# ROUTER socket for clients
frontend = context.socket(zmq.ROUTER)
frontend.bind("tcp://*:5555")

# ROUTER socket for workers
backend = context.socket(zmq.ROUTER)
backend.bind("tcp://*:5556")
```


Hash ring to distribute the workload among available workers

```
worker_addresses = [f"worker{i}".encode() for i in range(1, 6)]
ring = HashRing(nodes=worker_addresses, replicas=10)

key = client_id.decode()
target_worker = ring.get_node(key)
```

Send message from client to server/worker

```
backend.send_multipart([target_worker, client_id, request])
```



```
# Get the list of replica workers
key = client_id.decode()
target_workers = ring.get_nodes(key, count=3) # 3 rep
for worker in target_workers:
    backend.send_multipart([worker, client_id, request])
# Send the request to all replica workers
for worker in target_workers:
    backend.send_multipart([worker, client_id, request])
```

Receives message from serves and send to client

```
worker_msg = backend.recv_multipart()
worker_id, client_id, response = worker_msg
print(f"Proxy received message from worker: {response} to client {client_id.decode()}")
frontend.send_multipart([client_id, response])
```

A poller mechanism is used to handle multiple sockets simultaneously. It monitors activity on multiple sockets (frontend and backend) at the same time, efficiently handle incoming messages from either clients (via the frontend socket) or workers (via the backend socket) without blocking and ensure the proxy remains responsive to both clients and workers, routing messages in real-time.

```
poller = zmq.Poller()
poller.register(frontend, zmq.POLLIN)
poller.register(backend, zmq.POLLIN)

while True:
    sockets = dict(poller.poll())
    if frontend in sockets:
        client_msg = frontend.recv_multipart()
```

```
if backend in sockets:
    worker_msg = backend.recv_multipart()
```



Worker

Workers are responsible for processing client requests forwarded by the proxy. It interacts with a server database to storage the local changes received by the proxy by the client to synchronize.

Each worker has its own SQLite database managed by the ShoppingListManager object.

Performs operations such as creating lists, adding items, retrieving data, and updating synchronization statuses.

Handles synchronization of shopping lists and items by storing them in its database upon receiving requests.

Operates as an independent unit in a distributed system, ensuring scalability and fault tolerance.



The worker sets up a ZeroMQ **DEALER** socket to connect to the proxy.

Listens for requests from the proxy via the DEALER socket.


Processes requests based on the specified action field such as:

- Viewing all lists.
- Synchronizing a new list or item.
- Viewing items in a specific list.

Sends a response back to the proxy after processing a request, including status and any relevant data.

```
worker = context.socket(zmq.DEALER)
worker.identity = f"worker{worker_id}".encode()
worker.connect("tcp://localhost:5556")

db_path = f"server_{worker_id}.db"
print(f"Worker {worker_id} is using database: {db_path}")
manager = ShoppingListManager(db_path)
```



The worker receives a multipart message, decodes the client ID and request, and extracts the action to determine the task.

```
message = worker.recv_multipart()
client_id, request_raw = message
request = json.loads(request_raw.decode())
action = request.get("action")
```

Retrieves all active shopping lists from the worker's database.

```
if action == "view_all_lists":
    try:
        lists = manager.view_all_lists()
        response = {"status": "success", "lists": lists}
    except Exception as e:
        response = {"status": "error", "message": str(e)}
```



Saves a new list in the local database.

```
elif action == "sync_list":
    try:
        list_details = request.get("list", {})
        url = list_details.get("url")
        name = list_details.get("name")
        creator = list_details.get("creator")
        client_id_list = request.get("client_id")
        manager.save_list(url, name, creator, client_id_list)
        response = {"status": "success", "message": "List synchronized successfully."}
```

Handles periodic synchronization requests for lists or items.

```
elif action == "polling_list":
    try:
        list_details = request.get("list", {})
        manager.save_list(
            list_details["url"],
            list_details["name"],
            list_details["creator"],
            client_id
        )
        response = {"status": "success", "action": "sync_list", "list_url": list_details["url"]}
    except Exception as e:
```

Sends the response back to the client via the proxy

```
worker.send_multipart([client_id, json.dumps(response).encode()])
```



Thank you!

Anu Atolagbe 202400090

Catarina Canelas 202103628

Dany Ferreira 202108799

Margarida Pinho 201704599