Shopping Lists on the Cloud

Large Scale Distributed Systems

T6g06:
Sérgio Nossa, up202206856@up.pt
Pedro Marinho, up202206854@up.pt
Carlos Costa, up202205908@up.pt

Introduction

- The objective of the project is to create a local-first shopping list application
- In addition to saving lists locally, the app must sync shopping lists to cloud storage so users can share and recover their data.
- Therefore, each group must design an always-online server architecture that is highly available and resilient to crashes, using the architecture described in the Amazon Dynamo paper as a base

Design Challenge #1 - Scalability

- **Problem:** Support millions of users without bottlenecks
- Amazon Dynamo Inspiration:
    - Decentralized architecture
    - Data partitioning (sharding)
    - Replication for fault tolerance
    - Eventual consistency
    - No single point of failure

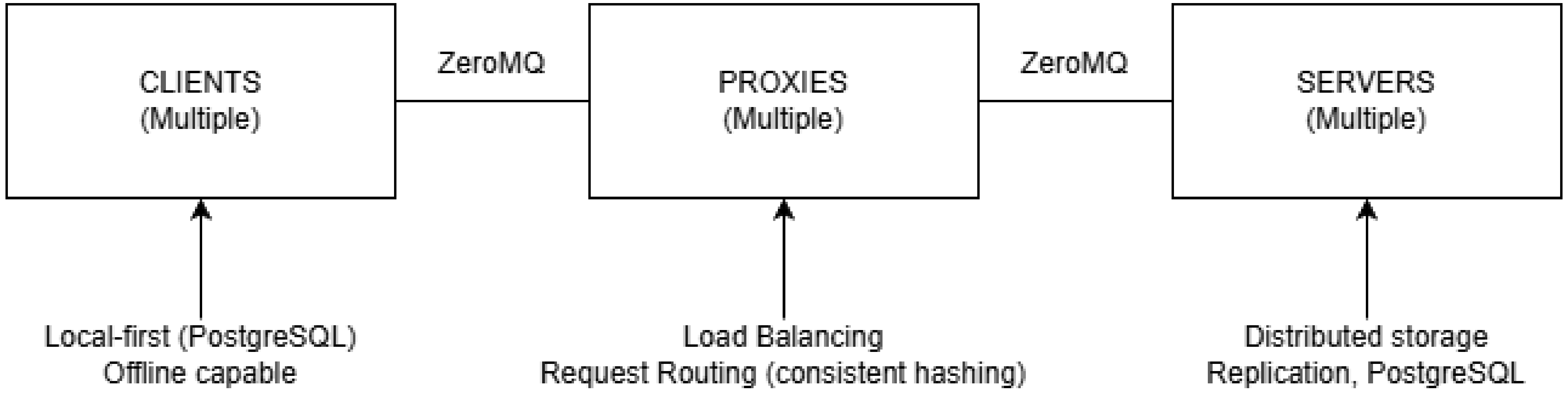
Design Challenge #1 - Scalability

- Our Application:
  - Each shopping list is independent (perfect for sharding)
  - Lists distributed across servers using consistent hashing
  - Multiple replicas per list for availability
  - Proxy layer for load balancing

Technologies

- **Python:** Primary programming language used for all application logic.
- **Pyzmq:** Python wrapper for ZeroMQ, enabling communication between the client, proxy, and server.
- **Postgres:** Database system used by both the server and client to store shopping-list data.

System Architecture



Client Architecture

- **Local-First Design:**
  - Full PostgreSQL database on device
  - Complete CRDT state stored locally
  - All operations work offline
- **Components:**
  - Storage Layer: Database operations, CRDT serialization
  - Communication Layer: ZeroMQ connection to proxy
  - Interface Layer: CLI for user interaction

Client Architecture

- **Technical Features:**
  - Unique client ID for CRDT operation tagging
  - Read-Write locks for thread-safe access
  - Automatic subscription to list updates
  - Background sync when online

Client Interface

- CLI (Command Line Interface) based interaction (directly with the proxy):
  - **List Management:** create, list all lists, show items and quantities, delete from local storage
  - **Item Operations:** add item with needed/acquired quantities, modify quantities, delete item
  - **Network:** request full list from network, pulling from cloud and merging. All modifications auto-sync in background

Proxy Layer - Responsabilities

- Request routing using consistent hashing
  - SHA256 hash of list UUID → server position on ring
  - Deterministic list-to-server mapping
- Load balancing across server pool
  - Hash-based distribution of lists
  - Ring traversal for failover

Proxy Layer - Responsabilities

- Server discovery via gossip protocol
  - Continuous peer discovery (every 0.5s)
  - Dynamic server pool updates
  - Stale server removal
- Failure detection and handling
  - Timeout detection with exponential backoff
  - Clockwise ring traversal to find available replica

Proxy Layer – Consistent Hashing

1. Hash Ring setup
   - Each server assigned unique position on ring
   - Servers ordered by their hash values
2. List Assignment
   - Shopping list hashed to determine position
   - Assigned to first server found clockwise on ring
3. Failover (when needed)
   - Primary server unavailable → move to next clockwise
   - Continue around ring until server responds
   - Ensures high availability despite failures

Proxy Layer – Consistent Hashing

- Benefits:
  - Deterministic routing (same list → same server)
  - Scalable (dynamic server add/remove)
  - Built-in failover through ring traversal
  - Load distribution via hashing

Server Architecture

- Data Management:
  - Each server responsible for subset of lists (via consistent hashing)
  - PostgreSQL for persistent storage
  - CRDT-serialized shopping lists
- Replication:
  - Lists replicated to 2 servers
  - Server-to-server gossip for peer synchronization
  - CRDT merge on read/write operations

Server Architecture

- Fault Tolerance:
  - Multiple replicas ensure high availability
  - Crash recovery from replica servers
  - Eventually consistent across all replicas
- Operations:
  - Receive list updates from clients/proxies
  - Store and merge with existing CRDT state
  - Propagate updates to replica servers
  - Respond to list retrieval requests

Communication Patterns

- **Client→Proxy:** DEALER→ROUTER: requests and writes with retries/backoff.
- **Proxy→Server**: DEALER→ROUTER: reads/writes, ACK/NACK, consistent hashing + clockwise failover.
- **Server→Server:** DEALER→ROUTER: replication (REPLICA to up to 3) and hinted handoff with ACKs.
- **Proxy→Clients (updates):** PUB→SUB: broadcasts list updates on the list ID topic, and clients subscribe per list ID.
- **Gossip/Discovery:** DEALER→ROUTER: Periodic peer announcements (every 0.5s) over the request channel, sharing current members and a ring version; newly seen peers are added, and peers missing from newer versions are removed to keep the membership and hashing ring consistent.

Data Flow Example – Client Adds An Item

1) Client A:
   - Adds "Milk" (CRDT operation)
   - Saves to local PostgreSQL
   - Sends update to Proxy

Data Flow Example – Client Adds An Item

2) Proxy:

- Calculates hash of list UUID
- Routes to Server 1 (via consistent hashing)

Data Flow Example – Client Adds An Item

3) Server 1 (Primary):

- Receives update from Proxy
- Merges with existing CRDT state
- Saves merged result to PostgreSQL
- Sends peer-to-peer replication requests to up to 3 servers; each replica confirms (ACK); retries on timeouts.
- Server sends ACK to Proxy; Proxy replies ACK to Client A and broadcasts a list update on the list's channel.

Data Flow Example – Client Adds An Item

4) Replica Servers (2 & 3):

- Receive replicated data
- Merge with their CRDT state
- Save to PostgreSQL

Data Flow Example – Client Adds An Item

5) Client B (subscribed to the list):

- Receives the list update broadcast from the proxy on the list's channel.
- Merges the update with local CRDT state and persists.

Design Challenge #2 – Conflict Resolution

- **Problem:** Multiple users editing same list concurrently
- **Simple approach:** Last-Writer-Wins (LWW)
  - Can result in data loss
- **Our solution:** Conflict-free Replicated Data Types (CRDTs)
  - Mathematically proven convergence
  - No coordination required
  - Works offline
  - Automatic conflict resolution
  - Eventually consistent

Initial Approach – LWW

- Simple timestamp-based conflict resolution
- Latest write wins, ties broken by peer ID
- **Problems:**
  - Data loss on concurrent edits
  - Poor for accumulating quantities
  - One user's work could be silently discarded

Improved Solution - CRDTs

- Custom implementation (no external libraries)
- Three core CRDTs:
  - **GCounter (Grow-only Counter):** base primitive for monotonic increments; per-replica counters
  - **PNCounter (Positive-Negative Counter):** built on two GCounters (positive + negative); tracks quantities; supports increment/decrement
  - **ORSet (Observe-Remove Set):** tracks item existence with unique tags; tombstone mechanism for proper remove semantics; add-wins bias in concurrent add/remove

Improved Solution - CRDTs

- ShoppingList Composite CRDT:
  - Combines ORSet (item existence) + PNCounters (quantities)
  - Logical Clock for causality tracking
  - Merge operation for replica convergence

CRDT Merge Example

- **Client A (offline):**
  - Adds "Milk" (need: 2)
  - Adds "Eggs" (need: 6)

- **Client B (offline):**
  - Adds "Bread" (need: 1)
  - Adds "Eggs" (need: 12)
  - Marks "Eggs" acquired: 6

- Both clients come online and sync
- After CRDT Merge:
  - Milk (need: 2, acquired: 0)
  - Bread (need: 1, acquired: 0)
  - Eggs (need: 18, acquired: 6) ← PNCounters merged!
- No conflicts, no data loss!

Conclusion

- What We Built:
  - Fully functional distributed shopping list application
  - Custom CRDT implementation (ORSet + PNCounter + ShoppingList)
  - Dynamo-inspired architecture (sharding + replication + consistent hashing)
  - Local-first design with offline capability
- Key Achievements:
  - High availability (offline operation, survives crashes)
  - Eventual consistency (mathematically guaranteed)
  - Zero data loss on concurrent edits
  - Scalable architecture (no bottlenecks)

Demo Video