Shopping Lists on the Cloud

Large Scale Distributed Systems

Diogo Silva, up202004288
Henrique Silva, up202007242
Tiago Branquinho, up202005567

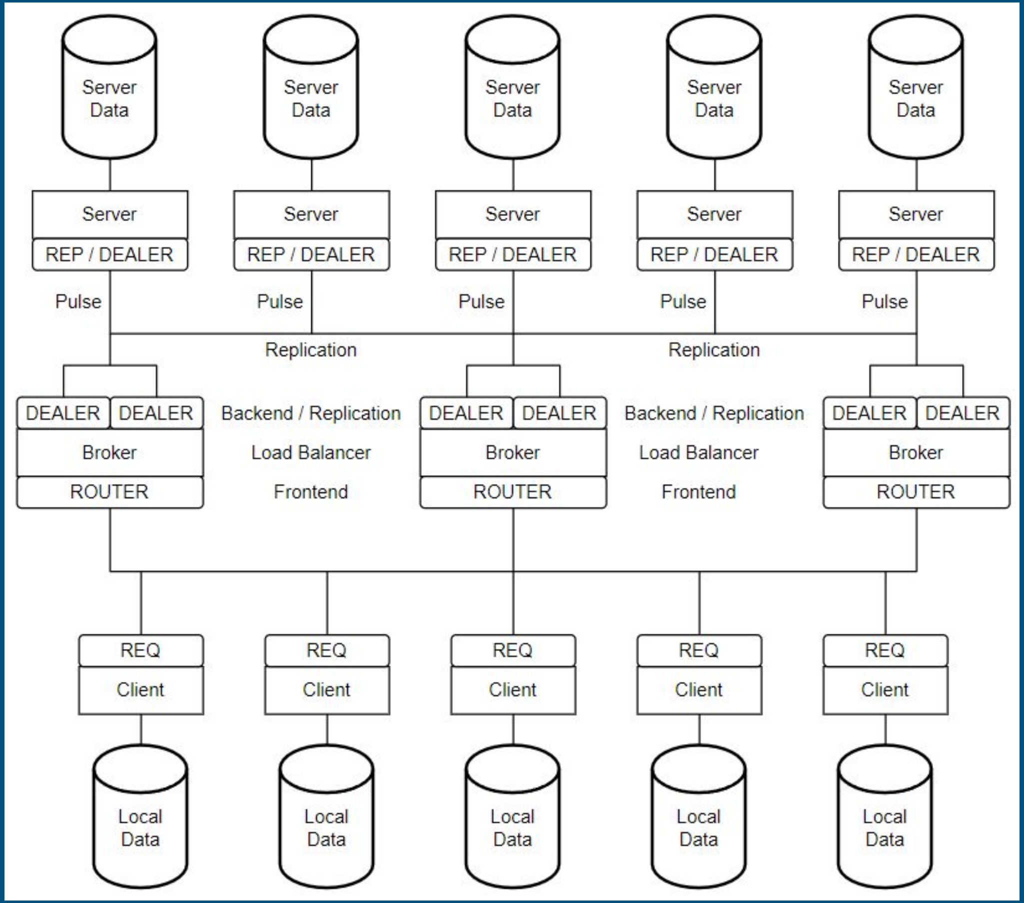
Application Requirements

- Local-first shopping list application:
  - **client-side** - persists data locally
  - **cloud component -** data sharing and backup storage
  - application must be **scalable**
  - achieve 2 of the concepts from:
    - **Consistency**, **Availability** and **Partition Tolerance** (CAP)
- **Shopping lists**:
  - have unique IDs
  - can be created or deleted by users
  - can be shared between users (using the ID)
- **Shopping list items**:
  - can be added or removed by users
  - have quantities, which the users can update (quantity 0 → item purchased)

System Architecture

- **Decentralized distributed** system
- 5 servers - each with 20 virtual nodes
- 3 brokers
- "**Pulse check**" system
- **Data replication** through brokers

- Python zmq (**ZeroMQ**)
- **SQLite** database:
  - for each client
  - for each server
  - lists with **removed** flag
  - items with **timestamps**
  - knowledge of lists to be **replicated**
- **TCP** in all sockets
- Message serialization (**JSON**):
  - in all communications



Conflict Handling - Items

- **LWW-Element-Set** CRDT
- Add set - added/updated items
- Remove set - removed items
- Elements:
  - **(name: (quantity, timestamp))**

- **Quantity update** (add set):
  - compare timestamps - update vs item:
    - if update is more recent:
      - change quantity and timestamp of item

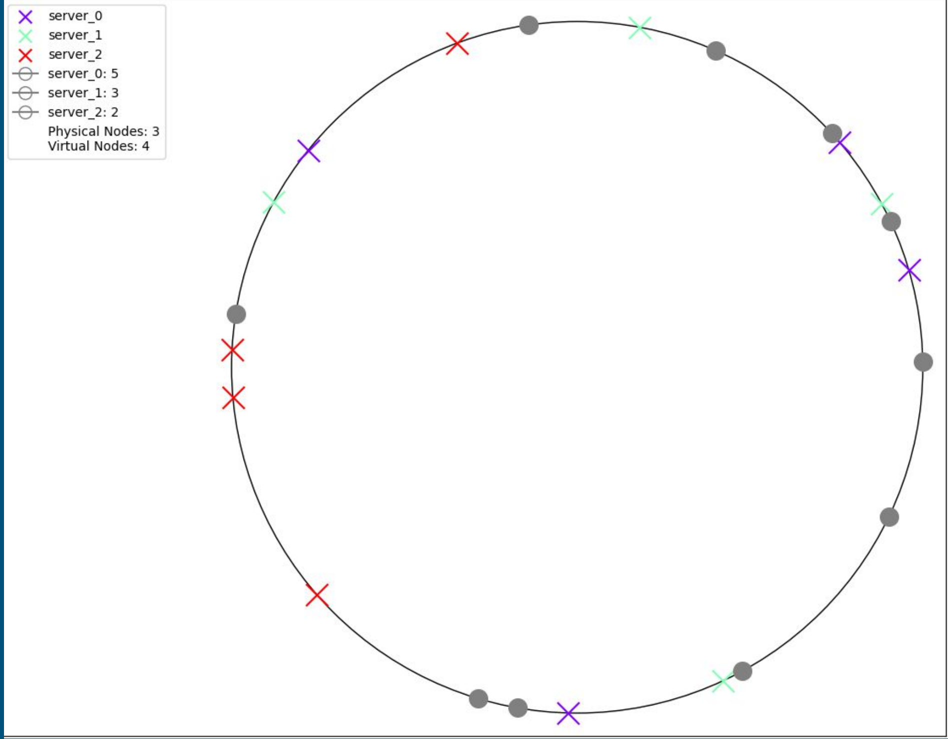
- **Merge between CRTDs**:
  - in each set, same items - more recent timestamps win
  - item in both sets - compare timestamps, find true state:
    - last action wins
- Major requirement:
  - **precise/reliable timestamps** for events comparison
  - avoid clock synchronization issues
    - universal clock in clients (NTP)

Conflict Handling - Lists

- **OR-Set** CRDT
- Add set - created shopping lists
- Remove set - deleted shopping lists
- Elements:
  - **(ID, name)**
- **Merge between CRDTs**:
  - server:
    - add sets union
    - remove sets union
  - client:
    - unknown lists don’t matter
    - remove sets union
    - remove intercepted with add set
- **Removal** always wins
- **Data structure** contains:
  - lists
  - respective items
- Merge process merges:
  - lists
  - respective items

Data Sharding

- Shopping lists distributed across servers
  - **distributed** system
- Each server represents a shard
- **Horizontal sharding** (by rows):
  - better scalability and load balancing
- **Consistent hashing** (SHA-256):
  - assignment of shopping lists to servers
  - directory-based sharding:
    - server's position on ring determines its responsibility
- **Load balancing**:
  - achieved through consistent hashing
  - target server (and backup nodes):
    - list ID applied to hashing ring



- Test configuration:
  - 3 Physical nodes
  - 4 Virtual Nodes
  - 10 shopping lists
  - 8 bits of salt

- All hashing done using SHA-256 (instead of 512)
  - Faster → less processing power
  - Efficient → less memory
  - Still provides good results (worse than 512)
  - 8 bits of salt used when hashing virtual nodes:
    - server_X_virtual_Y_SALT

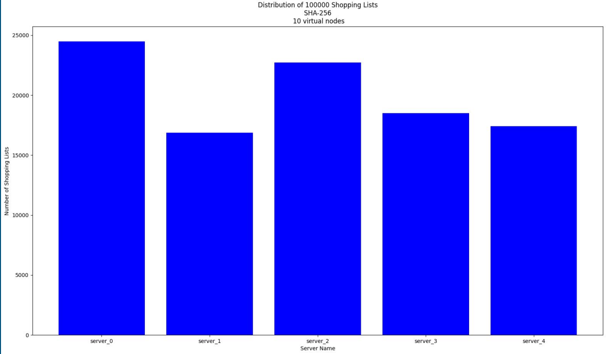
- Physical Nodes:
  - + Resource efficiency
  - +Hotspots

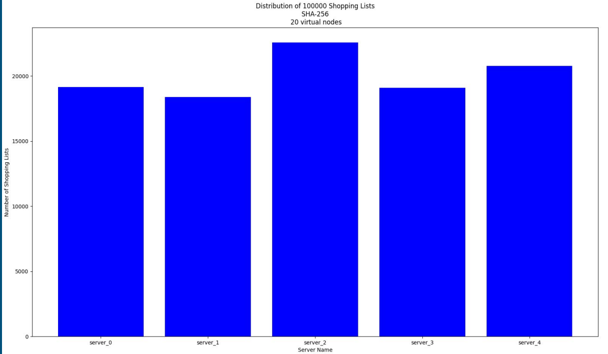
- More Physical Nodes:
  - + Fault tolerance
  - + Complexity

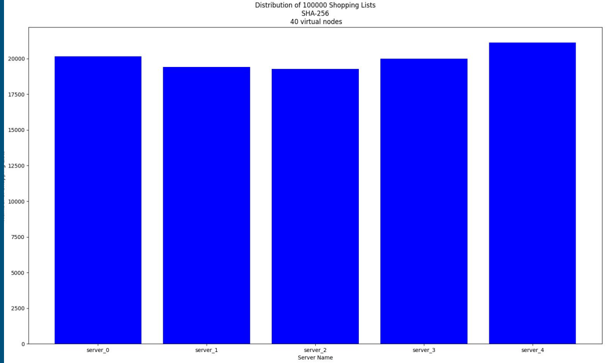
- Virtual Nodes:
  - + Work distribution
  - + Dynamic scaling
  - + Spreading in the ring

- Too many Virtual Nodes:
  - + Computational overhead
  - + Hashing complexity

- Used resources:
  - 5 Physical Nodes
  - 20 Virtual Nodes



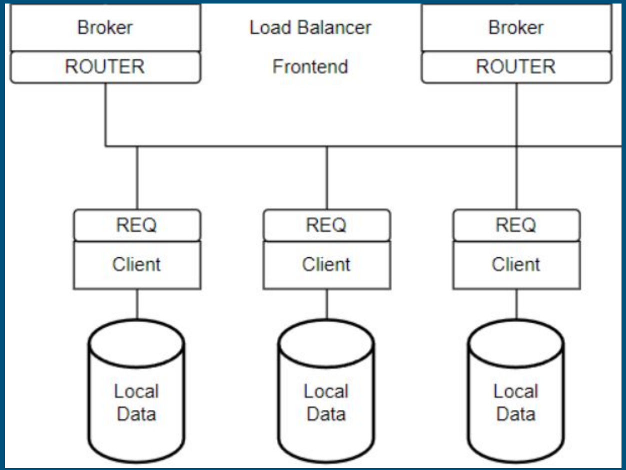




Client Logic

- Read database - store in CRDT
- Write action - CRDT/database locally updated
- **REQ** socket:
  - unique identity
  - connected to random broker
  - registered for **polling**

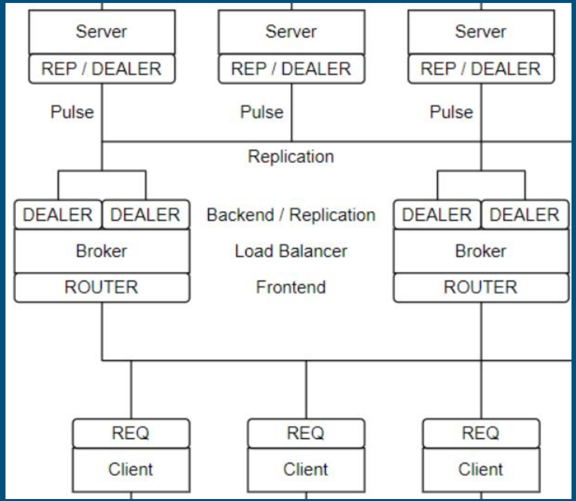
- Possible actions/requests:
  - **access a shopping list** - ID
  - **synchronize with cloud**:
    - CRDT sent to be processed
    - CRDT merged with response CRDT
    - database update
    - GUI refresh



- **Communication failure**:
  - timeout:
    - issue in broker or server communication
  - connect to next broker, repeat
  - all brokers dead:
    - end current request
    - maintain responsiveness

Broker Logic

- 3 sockets (**polling**):
  - **ROUTER** (frontend)
  - **DEALER** (backend)
  - **DEALER** (replication)
- Periodic "**pulse check**" - servers online/offline status



Frontend polling

- **Access to shopping list**:
  - ask servers, return to client
- **Sync with cloud**:
  - CRDT with several lists:
    - split/distribute to servers
    - append each server response as CRDTs are sent/resolved/received
    - send it all back to client

Backend polling

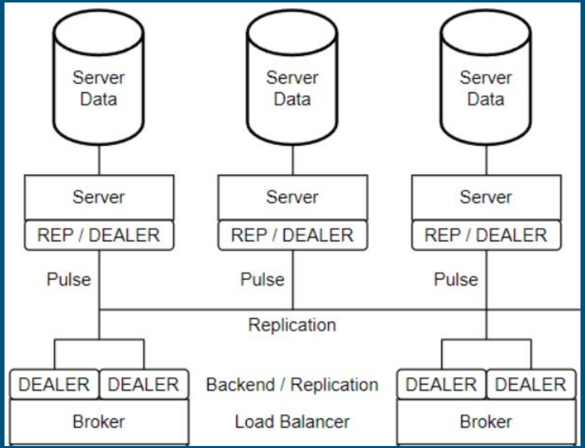
- Received **CRDT from server (replicate)**:
  - split/distribute to servers
  - replication socket used
  - message updated as each CRDT is sent/processed/received across servers
- Reply success to original server

Server Logic

- Database reading at start - data stored in CRDT

Socket in REP mode

- Response to **pulse** check
- Client **access to list** - database query, list data sent
- **Client sync**:
    - CRDT merged with message CRDT
    - database - recognize **lists to be replicated**
    - resolved CRDT sent
- **Data received to replicate**:
    - same as client sync
    - only difference:
        - database - update with **lists from replication**



Socket in DEALER mode

- **Replication** (at regular intervals) if:
    - database has lists with updates → replicate
- CRDT to broker, get feedback:
    - if replication worked:
        - database - clear info of lists to be replicated
- **Communication failure**:
    - same logic used by clients for brokers

System Trade-Offs

- **Client-cloud sync intervals** - higher VS smaller:
  - bandwidth - more VS less
  - overhead - reduced VS increased
- **Waiting timeout for broker** - higher VS smaller:
  - error rate - decreased VS increased
  - error detection - delayed VS premature
- **Replication intervals** - higher VS smaller:
  - data staleness - increased VS decreased
  - network resource demand - less VS more
- **Brokers** - many VS few:
  - coordination - complex VS simple
  - fault tolerance - high VS low

- **Storing removed shopping lists**:
  ❌ increased storage
  ✅ faster/better access and recovery
- **Periodic ARE YOU THERE messages**:
  ❌ increased network overhead
  ✅ better re-routing (less downtime)
- **Data replication across servers using brokers**:
  ❌ potential increased latency
  ✅ enhanced monitoring, load balancing

Conclusions

- CAP theorem - **AP system**:
  - **Availability** - read/write requests get responses, regardless of success/failure of other nodes
  - **Partition Tolerance** - disruption of node segments recognized/bypassed to maintain communications
  - **Consistency** - not all nodes have the same data at the same time, replication works towards it (eventual)

- **System scalability considerations**:
  ✅ collaborative editing of lists well managed - **data synchronization**
  ✅ clever server monitoring strategy and disruption recovery - **fault tolerance**
  ✅ fluctuating user traffic not very problematic - **load balancing**
  ❌ network issues, bandwidth limitations - **detrimental speed/efficiency**
  ❌ more dependency towards brokers - **point of failure concerns**

- **Future work**:
  - maybe compare data redundancy strategies
    - implement server-to-server direct communication
    - see how it affects network performance and system overall data