Shopping List Manager

SDLE FEUP - 2023/24

1MEIC05
Mário Ferreira
Pedro Barbeira

Context Description

- Develop an application which allows users to manage shopping lists
- Users interact with server through an User Interface
- Users can create, read, update and delete shopping lists
  - These operations are also valid for shopping items, since that counts as updating the list
- Shopping items have an user-defined target quantity
  - This quantity can be altered by the user through the UI
- Users can check an item to mark the acquisition of the target quantity
- Users can share their shopping lists with other users
  - Shared users are given both read and write permissions

The Solution

- Implement a cloud architecture to ensure high availability and scalability
- Data Server
  - Comprised of several Servers
  - Each server has a configurable number of virtual nodes
- Broker
  - Hides the distributed architecture from the client
  - Handles data sharding and node routing
- User Interface
  - Implemented as a Command Line Interface
  - Easily substituted for or wrapped by a Graphical User Interface

Data Server

- Comprised of a several nodes
  - Nodes are fully configurable
- Ensures data consistency through CRDT’s and Replicas
  - CRDT’s are handled through CRDTExecutionService
  - Replicas are handled through ReplicaService
- Each Node has its own Port, Thread and Repository
  - This way we can decouple request handling logic
  - Each Node functions as a Server once booted
- Exchanges CRDTOperations with Replicas
  - Server publishes to the nodes where it’s replicated
  - Server subscribes to the nodes it replicates

Broker

- Knows about the whole server topology
  - Mapped in a configuration file
- Runs several Threads to increase availability
- Routes requests from client to the appropriate node
  - Done through a mapping between ID prefixes and addresses
- Reroutes requests to a replica if the main node isn’t available
- Handles authentication and token validation
  - Implemented through a simple AuthService which can be easily extended
- Does the bare minimum required computation

User Interface

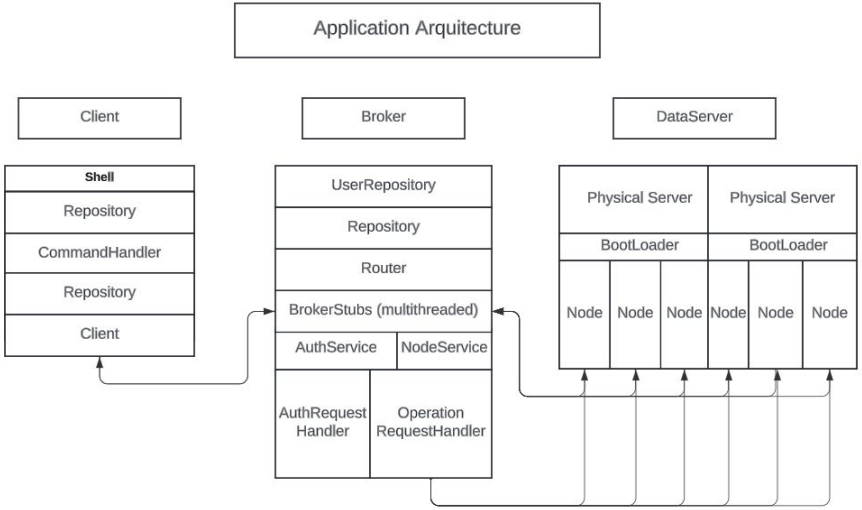
- Relatively straightforward implementation
- Uses a Command Handler to parse user commands
  - Implements a state machine
- Command Handler calls Client to send requests
  - Routed to the Data Server by the Broker
- Results are displayed in the console using a simple table format
- Implements local-first by updating physical data
  - Done before sending request to broker and after receiving response
- Likely not the best possible solution as far as UI’s are concerned
  - However, it gets the job done

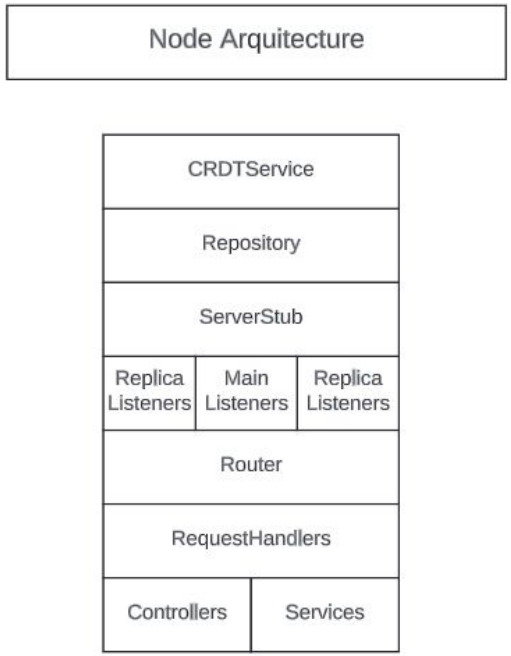
Server Topology

- 5 physical nodes, each with 2 replicas
- Each physical node runs 3 virtual nodes
- Configurable though server-*-settings.json

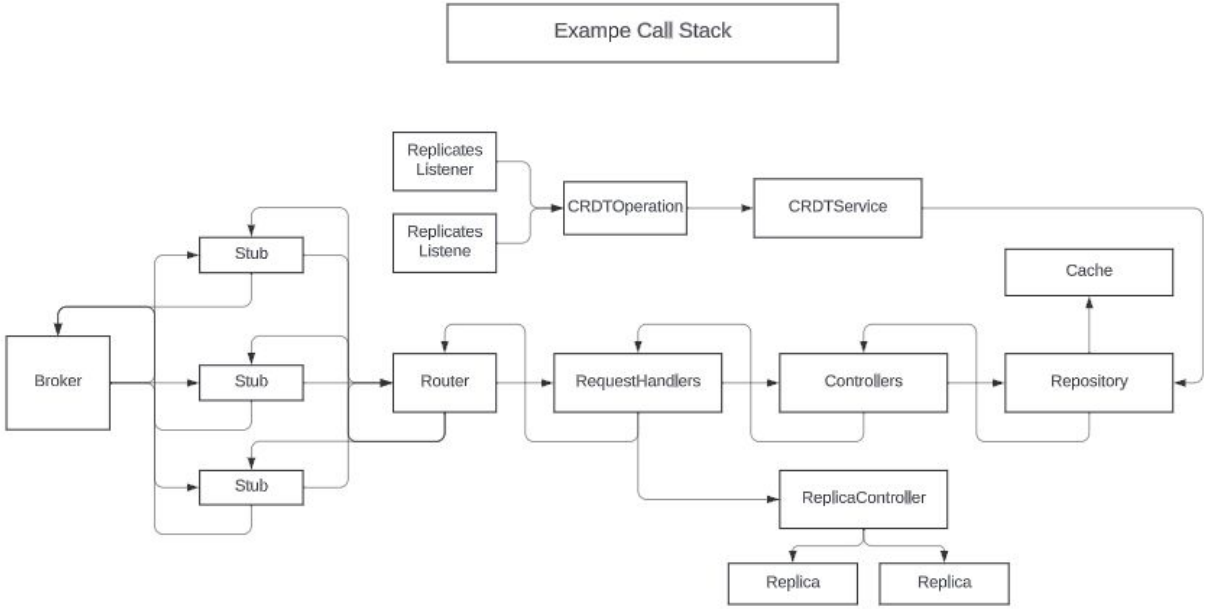
Node	Replicates	Replicated On
1	4, 5	2, 3
2	5, 1	3, 4
3	1, 2	4, 5
4	2, 3	5, 1
5	3, 4	1, 2

Design and Architecture - System Overview





Design and Architecture - Server Call Stack



Main Challenges - Server Boot

- Problem: upon booting, the server needs to request data from both the nodes it is replicated on and the nodes it replicates, since said data could have been mutated while it was down
- Solution: implement a Bootstrapper
  - Initializes Main Node’s services
  - Opens replica listeners
    - Done before requesting data, to ensure nothing is lost
  - Requests data from the replicas
  - Temporarily listens to nodes where the server is replicated on
    - This ensures no operation is lost while booting
  - Starts listening to broker on several threads
    - Terminates the temporary listeners through a flag

Main Challenges - Ensure Data Consistency

- Problem: in a highly distributed environment it’s very easy to run into unpredictable race conditions, which may corrupt data
- Solution: implement Conflict-Free Replicated Data Types
  - Implemented with generics (CRDT<T>)
  - Each CRDT has a value, a version and a timestamp
  - Requests which mutate the CRDT trigger the publication of a CRDTOperation to replicas
  - CRDTOperations are put in a Queue as they arrive
    - This way we can have several threads feeding CRDTOps to the CRDTExecutionService
  - Commutative operations are applied sequentially
    - This ensures eventual data convergence
  - Non-Commutative operations are applied with Last-Write-Wins
    - This way we know we have the most recent data

Main Challenge - Route Requests to correct Node

- Problem: since the data is sharded, we need a way to ensure the Broker knows in which node a given data item is
- Solution: append a prefix to the data item id and keep a prefix-address map in memory
  - High scalability:
    - 3-character case-sensitive strings map up to 17576 nodes
    - <String, String> maps have a low memory cost
    - Using a HashMap we ensure 0(1) lookups
  - No concurrency issues:
    - Prefix map is only written on at boot time

Main Challenge - Route Create Requests

- Problem: to ensure an even distribution of data across nodes we need to define in which node a given item is created on
- Solution: use a round-robin distribution and a Queue
  - Broker is responsible for creating ID's and appending the prefix
    - Use UUID to ensure we never run out
  - We can use an auxiliary data structure containing a prefix and it's address
  - Round-robin on queues can be done through pop() and push()
  - Pop and push on queues are 0(1), meaning fast processing times
  - Although simple, the solution can handle concurrency
    - Exact order of circular attribution doesn't really matter
    - Prefixes ensure we always know where a given data item is
    - If we need to lock the queue, the critical section is very small (and very fast)

Main Challenge - Server Failure

- Problem: what if one of the servers suddenly crashes
- Solution: use replicas to ensure availability
  - The broker keeps a table of alternate addresses to a given prefix
  - If the request to the data server times out the broker tries the alternate addresses in order
    - Can be improved by sharing load between replica nodes
  - Since this table is configured at boot time, we run into no concurrency issues
  - The broker always attempts to route the request to the main node
    - This way the main node joins the party as soon as it opens the port

Main Challenge - Node Insertion

- Problem: how do we insert nodes into the network?
- Solution: use configuration files and insert nodes manually
  - In this type of context, node insertion is a likely uncommon operation
    - We're not in a peer-to-peer or blockchain context
  - It's also likely nodes won't be inserted individually
    - It would be better to insert a few at once
  - Node insertion is likely a delicate operation
    - We must ensure the broker is properly configured
    - We must ensure data is properly copied
    - Relying on automatism to do this would boil down to needless complexity
      - Specially consider the degree of parallelism we're working with
  - Rebooting the broker is a very fast operation
    - It's a relatively small piece of software

Improvements

- CLI architecture is somewhat rigid
  - Could be improved by extracting classes and interfaces
- Data stored in file system
  - Would be better to use a Database
- Requests are not encrypted
  - Anybody can steal our shopping list information
- Node insertion is a lot of work
  - Persisted data must be manually altered
  - Perhaps a daemon could handle it better
- Each Thread processes requests synchronously
  - Handling requests asynchronously in broker and server would greatly increase our capacity
  - We tried to implement this, but ZMQ does not allow it (as far as we know)

Conclusions

- Cloud computing is a challenging and fascinating field
  - The amount of parallelism forces non-linear thinking
- Ensuring data consistency is hard work
  - We had to restructure half the project to implement the bootstrapper
  - Without the bootstrapper our call stack would be all over the place
- Sometimes changing approaches makes the difference
  - Using the prefix-address map solved many broker-related problems
- Architecture is crucial
  - There were times were certain components grew out of control
  - Refactoring and redesigning them resolved those situations
  - Considering how many parts we’re handling, organization is key