SDLE

David Fang (202004179)
Mariana Azevedo (202005658)
Pedro Correia (202006199)

Introduction

This work was carried out as part of the Large Scale Distributed Systems course, with the purpose of creating a local-first shopping list application.

The application that was developed divides into three components:

- Client, broker and server.

This project is being developed in Python, using the ZeroMQ library as the base ground for socket communication.

ZeroMQ

Python

Client

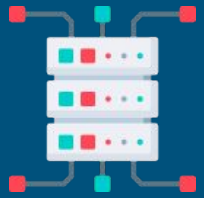
Broker

Server

Architecture

Client

Makes requests to the server

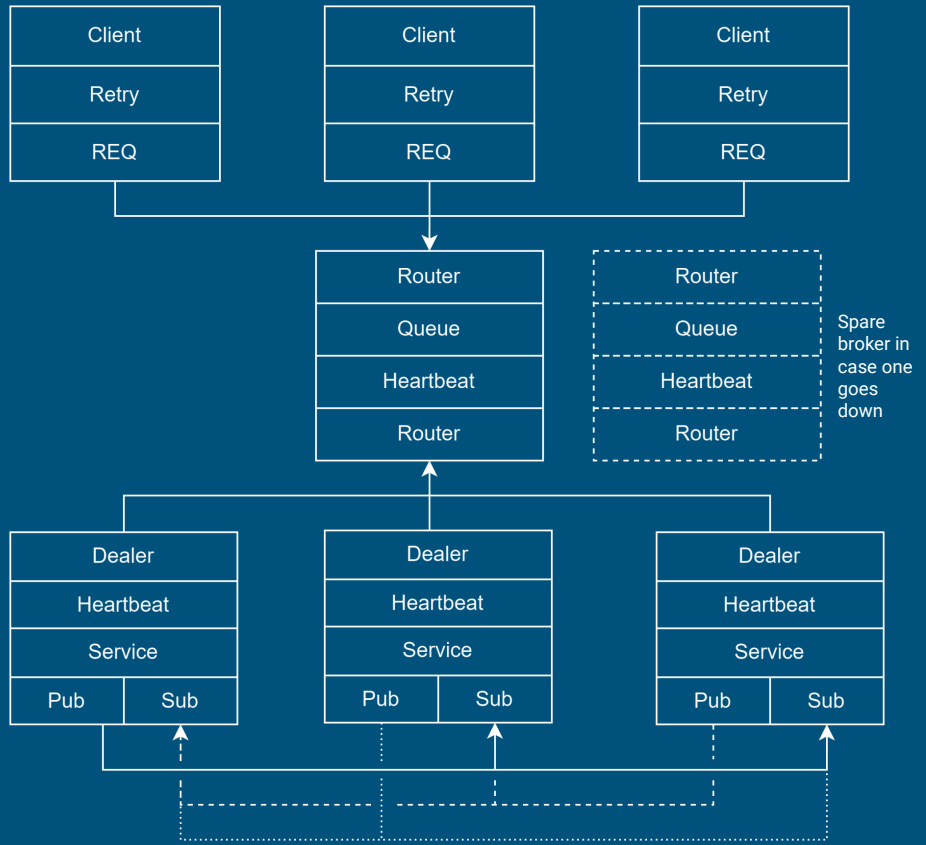


Broker

Distributes clients requests to the different servers

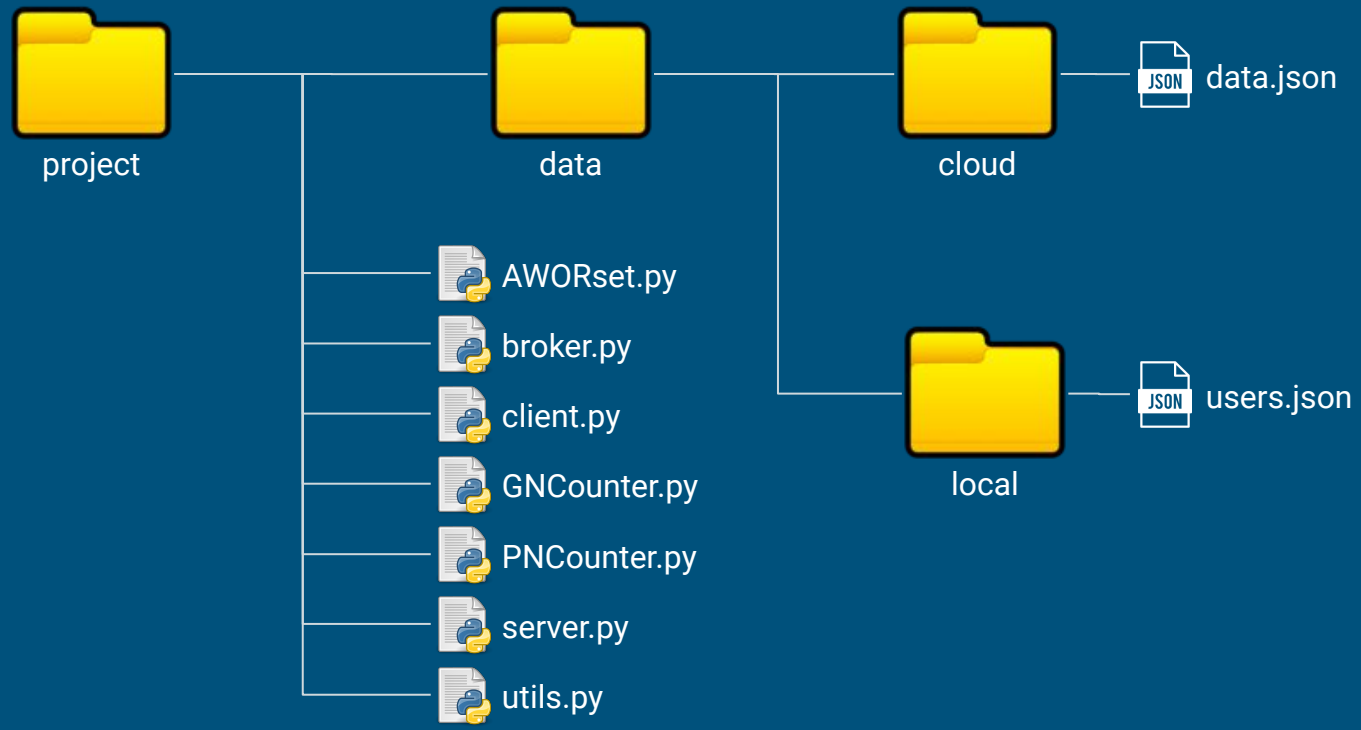
Server

Handles client requests



Paranoid pirate pattern adaption from zmq guide

Data structure



Data structure

Class definitions:

- AWORset.py
- broker.py
- client.py
- GNCounter.py
- PNCounter.py
- server.py
- utils.py

data.json

list_id: String
list_name: String
owner: String
cCounter: Integer
context: Set of tuples
items: Dictionary - **contexts** as keys & **Items** as values

users.json

name: String
password: String
lists: List with same attributes at **data.json**

Item

name: String
quantity: PN Counter
bought_status: PN Counter

Client

What is it?

Description:
Terminal-based app that uses **REQ** socket to communicate with the server.

Features:
- Simple authentication
- Attempt-retry mechanism
- Local data storage

Challenges

Problem:
- Ensure reliability.

Answers:
Reliability - attempt-retry mechanism

How?

Flow:
- Begins with authentication
- Interface with interactions
- Saves information locally
- Pings broker to check liveness
- Sends requests to server

case not received reply:
Wait until timeout to send again
Once reached maximum number of attempt aborts

case received reply:
Database updated

Implementations

Authentication

```
def auth():
    While True:
        asks username and password
        if found a match
            breaks
```

Attempt-retry

```
attempt = 0
while max_amount is not reached:
    sends message
    creates poller
    if poller received answer within TIMEOUT:
        returns the answer
    else:
        attempt += 1
if reached maximum of attempts:
    aborts
```

Overall

REQ socket:

```
client = context.socket(zmq.REQ)
```

Use of poller():

```
poller = zmq.Poller()
poller.register(client,zmq.POLLIN)
dict(poller.poll(TIMEOUT))
```

Send and receive messages:

```
client.send(message)
message = client.recv_multipart()
```

Broker

Router
Queue
Heartbeat
Router

What is it?

Description:
Intermediary between clients and servers in a distributed system. Uses **router** sockets for communication with both clients and servers.

Features:
- Client interaction
- Server interaction
- Message routing

Challenges

Problem:
- Server availability
- Data integrity

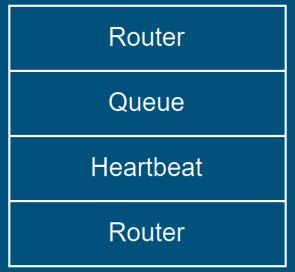
Answer:
Availability - Heartbeats to denote servers liveness

Integrity - Periodically sends a reread request to all the available servers at the moment

How?

Actions:
- **Client/Server requests:**
Validates and routes to the appropriate clients/servers.
- **Server Registration:**
Contains a queue where available server are registered.
- **Heartbeats and Expiry:**
Periodical trade of heartbeats to maintain liveness of the queue.
- **Server synchronization:**
Periodically sends a request to the servers to **reread** the database

Implementations



Classes

Server:
- address
- expiry

ServerQueue:
- queue of servers

functions:
- ready - signals availability
- purge - kills expired servers
- next - obtain the next available server

Main loop

while True:
if there is any available server:
poller will be used for both client and server
else:
poller will be used for servers only
if poller receives client_request:
deal with request
else if server_request:
deal with request and check if it's time to send **heartbeat** or **reread**

Overall

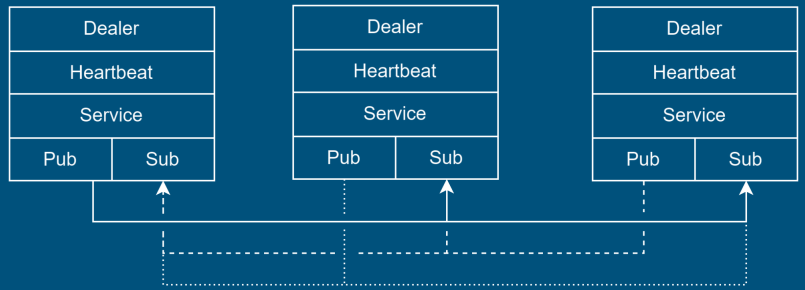
Sockets:
- **frontend** context.socket(zmq.ROUTER)
- **backend** context.socket(zmq.ROUTER)

Pollers:
- poll_servers.register(backend…)
- poll_both.register(frontend…)
- poll_both.register(backend…)

servers = **ServerQueue()**

main loop()

Server



What is it?

Description:
Service that handles requests from both broker and client side using **DEALER** socket and contains a **PUB** and **SUB** sockets to transmit data across other servers.

Features:
- Request handling
- Data replication
- Data upload
- Periodical data fetching

Challenges

Problem:
- Broker availability
- Data consistency

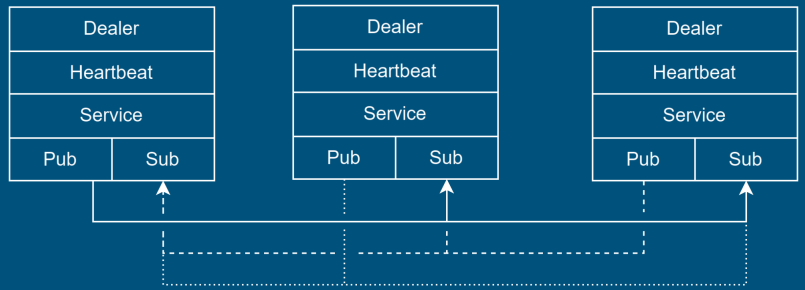
Answers:
Availability - Heartbeats to denote broker liveness.

Data consistency - Upon a request related with data (creation/update/deletion), propagate the same information to other servers.

How?

Flow:
Begins with reading the database
Creates a **PUB** and the needed **SUB** sockets
Tries to connect to the available broker
In case brokers are down:
Tries to reconnect to another one
In case any broker is up:
Handles requests and propagates information to other servers if needed

Implementations



Classes

ZmqSubscriberThread
- sub_sockets

function:
- run - checks if any sub_socket is receiving a request and handles it.

Main loop

attempts to connect to a broker

while True:
time = INTERVAL

if it receives a request from the broker in time:
Handles request

else:
Tries to connect again to the broker

if it's time:
Send heartbeat to **broker** to signal liveness

Overall

Sockets:
- **server** context.socket(zmq.DEALER)
- **pub_socket** sub_context.socket(zmq.PUB)
- **sub_sockets** sub_context.socket(zmq.SUB)

Starts **thread** for **sub_sockets**

main loop()

CRDT - Conflict-free Replicated Data Type

What is it?

Type of data structure that enables concurrent updates across multiple replicas without the need for coordination between them.

Why use it?

Enables efficient and scalable data processing and analytics in multi-node environments.

Some of the benefits:

- **Concurrent Updates**
- **Eventual Consistency**
- **Fault Tolerance**
- **Scalability**

How does it work?

CRDT's work by defining a set of operations that can be applied to the data structure. These operations are designed to commute, meaning that the order in which they are applied does not affect the final result.

Implementation - AWORSet

What is it?

A set datatype in which additions take precedence over removals. Used for the items in the list.

Attributes

list_id: String
list_name: String
owner: String
cCounter: Integer
context: Set of tuples (states of items)
items: Dictionary - **contexts** as keys & **Items** as values

Main Methods

add(Item) - adds a new context to the context set and a new item to the list of items

remove(context) - removes the item of the target context

join(aworset) - merges two aworsets

Implementation - GNCounter & PNCounter

What are they?

GNCounter:
Grow-only Counter that only supports incrementing (growing) values.
PNCounter:
Positive-Negative Counter extends the concept of GNCounter, by introducing support for both incrementing (positive increment) and decrementing (negative increment).
Used for the quantity and status of the items

Attributes

GNCounter:
- counter: Integer

PNCounter:
- positive_counter: **GNCounter**()
- negative_counter: **GNCounter**()

Main Methods

GNCounter:
increment -> increases counter
merge -> chooses biggest counter
PNCounter:
increment -> increases positive counter
decrement -> increases negative counter
merge -> merges the counters respectively
lookup -> returns the difference between the counters

Improvements

- Study the Amazon Dynamo paper and see how to implement it on the project.
- Add a way to confirm that the servers reread the data, instead of just trusting them.
- Add tests to measure the application performance.