Design Document

By: Aishwarya Malgonde, Sahil Jindal
Course: CS 677 | Professor Prashant Shenoy

Lab 3: Asterix and Double Trouble -- Replication, Caching, and Fault Tolerance

April 29, 2023

System Overview

The goal of this assignment is to ensure high performance and tolerance to failures for the simple trading platform created using microservices in Lab 2. Specifically, we have added caching, replication, and fault tolerance to the application. This application consists of a Frontend service that interacts with clients, a replicated Order service that handles order processing, and a Catalog service that manages stock catalog data. We have reused the Lab 2 code and added all these new features on top of it. We have referenced the Lab 2 design documentation wherever required and the github link is provided in the references section.

In the second part, we have deployed our application on AWS. Finally, we have performed multiple experiments for latency comparisons.

Architecture

In addition to the Architecture section mentioned in Lab 2 design documentation, we have incorporated the following in each module -

1. The **Frontend** service will now additionally manage the replication for the Order Service, do caching for stock lookup and provide an endpoint for order query lookup
2. The **Catalog service** will now additionally make a cache invalidation request to the Frontend, if it makes an update to the stock db
3. The **Order service** will now be replicated with three replicas and will have fault tolerance capabilities
4. The **client** will now additionally make GET order query requests to the Frontend service, for all the successful order trade requests made by it

Technologies Used

In addition to the Technologies Used section mentioned in Lab 2 design documentation, we have incorporated the following -

1. Caching - We have used **Redis server** for in-memory caching

Design Considerations

In addition to the Design Considerations section mentioned in Lab 2 design documentation, we have considered the following -

1. Why Redis for caching?
    a. It can significantly improve the response time of a microservice by caching frequently accessed data in-memory, thereby reducing the number of database queries and network requests.
    b. It is highly scalable and can be easily clustered to handle large volumes of data and traffic
2. Improved functionality
    a. The Frontend service will now cache the stock lookup data in-memory for faster responses
    b. The Order service is now fault-tolerant and has been replicated. In the event of a crash failure, the Frontend automatically selects another replica as the leader to ensure seamless processing of client requests. These failures are completely hidden from the client and are handled entirely by the Frontend.
3. Code structure and Ease of Running
    a. All the microservices can be started from the **src** directory itself
    b. We have created the **start.sh** and **stop.sh** bash scripts to start and stop the entire application in one command
    c. The backend services are started first, then the frontend service is started, so that the frontend is able to choose the Order leader

Main Application Features

Microservice 1 - Frontend Service

In addition to the features of the Frontend Service section mentioned in Lab 2 design documentation, we have added the following -

Order Query API

The Frontend now also provides an endpoint for getting the order details when the order number is provided. It forwards this request to the Order service and sends the response from the Order service back to the client

Part 1: Caching

For In Memory Caching, we are using Redis server. The caching is done by the front end server only.

1. We are initializing the redis connection at the beginning of the frontend_service.py file
2. In the front end we are configuring the maximum memory size and eviction policy so that the cache memory doesn't bloat and become a bottleneck.
3. Whenever a lookup request comes we first check if the stock name is present in the cache, if it is not present or if it is invalid, then only the request is forwarded to the Catalog service. All the responses from the lookup request are cached with a fixed expiry time.
4. Whenever the Catalog service updates the stock, it makes a requests for cache invalidation, and that particular stock is deleted from the cache by the frontend server.
5. This cache is synchronized with read-write lock for thread safety

Please note, that the role of Frontend in Replication is described in the Part 2: Replication section under Order Service section

REST API Endpoints

In addition to the previous list, we have implemented the following API endpoints in the Frontend:

1. Order query GET url - /orders/<order_number>
    a. Sample successful response -

```
{
  "data": {
    "number": 33,
    "name": "GameStart",
    "type": “sell”,
    "quantity": 150
  }
}
```


b. Sample error response -


```
{"error": {
            "code": 400,
            "message": “Bad request. 9999999 order not found in ORDER LOG”
          }
 }
```


2. Order leader GET url - /get_order_leader
   a. Sample successful response -


```
{
  "data": {
    "order_leader": 9092
  }
}
```


   b. Sample error response -


```
{"error": {
            "code": 500,
            "message": “Internal Server Error. Check get_order_leader logs”
          }
 }
```


3. Cache invalidation GET url - /cache_invalid/<stock_name>
   a. Sample successful response -


```
{
```

```
    "data": “cache invalidation successful”
}
```

b. Sample error response -

```
{"error": {
        "code": 500,
        "message": “Cache Invalidation unsuccessful. Check error log”
    }
}
```

Microservice 2 - Backend Catalog service

In addition to the features of the Catalog Service section mentioned in Lab 2 design documentation, we have added the following -

If the Catalog service makes a successful update to the

PyRO Interface

Connection with the Catalog service can be established using fields -
Service name (string) - backend.catalog
Name server host (string) - 0.0.0.0
Name server port (integer) - 9096

The Catalog service’s exposed endpoints remain the same as in Lab 2.

Microservice 3 - Backend Order service

In addition to the features of the Order Service section mentioned in Lab 2 design documentation, we have added the following -

Part 2: Replication

To make sure that our stock bazaar doesn't lose any order information due to crash failures, we have replicated the Order service. When we start the stock bazaar application, we first start the catalog service (as can be seen in src/start_app.sh). Then we start three replicas of the order service, with each replica having its own PyRO nameserver. The port numbers of these nameservers are predefined, mainly, 9090, 9092 and 9094 (as we can see the run commands here - src/back-end/order_service/run_replication.sh). We treat these port numbers as replica id numbers, since they are unique and required for making a connection. They are defined in setup.py of the Frontend service and the Order service.

The Frontend service is started after the backend services are up and running. Whenever the Frontend service starts, it sends a ping to the replica with the highest port number to see if it's responsive. If it is not, then the Frontend will try the replica with the second highest port number. The process repeats until a leader has been found. If no leader is found, which means no Order replica is running, then the Frontend service will display this error message, abort and terminate. The frontend forwards all the trade POST requests and order query GET requests to the Order leader replica only.

Once the leader is found, the Frontend service notifies all the replicas that a leader has been selected with the selected port number, and the LEADER_FLAG is updated inside the replicas. The LEADER_FLAG is defined in the Order class, and it helps the replica know whether it's a leader or not. If the replica is a leader, then it propagates all the successful trade details to other replicas. During this propagation, the transaction number (TXN_NUM) of the replicas is also updated, so in case the leader goes down and another replica becomes the leader, it can update new trade orders with correct TXN_NUM.

The Frontend also exposes an endpoint for getting the leader of the Order service. So whenever an Order service replica is revived, it contacts the Frontend at start time to see if there is a leader. If the leader is chosen, then the replica tries to synchronize order logs and latest transaction number with the leader.

Part 3: Fault Tolerance

We have designed the fault tolerance features under the following assumptions -

1. All Order service replicas are synced at application start time
2. Order leader crashes only before executing any order or after notifying all the replicas, not in between these processes

In this part we handle failures of the order service, crash failures only. First, whenever any replica crashes (including the leader), trade requests and order query requests are still handled correctly. When the front-end service finds that the leader node is unresponsive, it will redo the leader selection algorithm, by selecting the healthy replica with the highest id number .

After a replica crashes and comes back online, it can synchronize with the other replicas to retrieve the missed order information. The replica will check its database file and obtain the latest order number it has, then ask the other replicas for any missed orders since that number. This synchronization process occurs in parallel to avoid missing any current trade requests that the leader is processing. This is achieved by acquiring the write lock on the leader's order log file while taking the difference.

PyRO Interface

Connection with the Order service can be established using fields -
Service name (string) - backend.order
Name server host (string) - 0.0.0.0
Name server port (integer) - 9090 / 9092 / 9094 (leader is chosen by the Frontend, client doesn't need to worry about this)

The following endpoints have been added in addition to those mentioned in the Lab 2 design documentation -

Query method name - **query**

1. Input Parameter
    a. order_number (int)
2. Output Parameters (tuple)
    a. code (integer) - can be any of the HTTP status codes listed in its section
    b. response (dict if successful, otherwise string)
        1. Sample response when code is 200

```
{
   "data": {
      "number": 33,
      "name": "GameStart",
      "type": “sell”,
      "quantity": 150
   }
}
```

        2. Sample response when code is not 200

“Bad request. <order-number> order not found in ORDER LOG”

Getting leader notification method name - **leader_notification**

1. Input Parameter
    a. leader_port (int)
2. Output Parameters (str)
    a. “OK”

Getting order details from leader method name - **record_leader_trades**

1. Input Parameter
    a. data (dict)

```
{
    "transaction_number": 33,
    "stock_name": "GameStart",
    "order_type": “sell”,
    "trade_quantity": 150
```

}

2. Output Parameters (str)
    a. “OK” or “ERROR”

Getting order log diff from leader method name - **get_order_log_diff**

1. Input Parameter
    a. rep_txn_num (int)
2. Output Parameters (str)
    a. None - if no synchronization
    b. Pandas dataframe converted to dictionary - if synchronization happens

Data Models

In addition to the Data Models section mentioned in Lab 2 design documentation, we have added the following -

Order Log

Location - replica 1 - "src/back-end/order_service/data/order_log_9090.csv",
replica 2 - "src/back-end/order_service/data/order_log_9092.csv",
replica 3 - "src/back-end/order_service/data/order_log_9094.csv"
transaction_number (integer)
stock_name (string)
order_type (integer)
trade_quantity (integer)

HTTP Status Codes

Same as in Lab 2 design documentation

Client

In addition to the Client section mentioned in Lab 2 design documentation, before stopping, the client will retrieve the order information of each successful trade that was made using the order query request, and check whether the server reply matches the locally stored order information.

Part 4: Testing and Performance Evaluation

Test Cases

In addition to the Test Cases section mentioned in Lab 2 design documentation, we have added the following scenarios -

Order Service

1. For order query, we checked with correct & incorrect order numbers

Frontend Service

1. For order query we checked with correct & incorrect order number
2. For cache invalidation we checked with correct & incorrect stock names
3. For order leader request we checked with correct and incorrect url

Caching

We ran the test_caching.py script and checked the frontend logs. We ran it for the following scenarios -

1. If cache is happening but sending same lookup request back to back
2. If cache is getting is getting invalidated after a successful trade request

Replication

We performed testing for this from the terminal and by checking the frontend and backend logs. We ran it for the following scenarios -

1. We started all the three replicas and checked if the leader was chosen currently
2. We started all two replicas and checked if the leader was chosen currently
3. We verified if the frontend was sending requests only to the leader
4. We verified if the leader propagated trade details and the order logs are same for all the replicas

Fault Tolerance

We performed testing for this from the terminal and by checking the frontend logs. We ran it for the following scenarios -

1. Stopped the leader while the client was still sending requests
2. Started a crashed replica while the leader was serving client requests

Performance measurement

We performed the latency comparisons experiments. We deployed the server on AWS and ran 5 clients parallely on our local machine. We varied the POST order trade request probability as 0, 0.2, 0.4, 0.6 and 0.8, and noted the latencies. We performed these experiments under two scenarios - caching enabled and caching disabled. We have captured and presented these results in the Evaluation doc. On the client side, we logged the timestamps in a log file and latency was calculated using them. We measured latency from the client side as -

client latency = network latency + server latency

Where network latency accounts for sending and receiving messages, and server latency is the request processing time.

References

Apart from the support given by piazza and the TAs we referred the following documents:

1. https://docs.python.org/3/library/http.server.html
2. https://requests.readthedocs.io/en/latest/
3. https://pyro5.readthedocs.io/en/latest/api/api.html
4. https://en.wikipedia.org/wiki/List_of_HTTP_status_codes
5. https://redis.io/docs/clients/python/
6. Lab 2 design documentation - https://github.com/umass-cs677-current/spring23-lab3-aishwarya-n-sahil/blob/main/docs/Design_Document.pdf
7.