CS 677 S21 Lab 3 - Pygmy

Design Document V2

Submitted By
Yelugam Pranay Kumar
Date: 30 April, 2021

The project has been implemented in Python using Flask and can be run using a single bash script.

[1] Project Setup:

Execution:

LOCAL

- To execute this project **locally**, execute **run.sh**

AWS:

- To execute this project on **AWS**, execute **sh aws_run.sh $(cat config.txt).** Before executing this script, make sure that the **config.txt** file has the path to the **KP**(key pair) file and the **DNS** for the category, order and the front-end are set up.

  An example config.txt would look like this:

  ./677kp.pem
  <ec2-3-83-245-8.compute-1.amazonaws.com> #load balancer
  <ec2-3-83-245-8.compute-1.amazonaws.com> #frontend
  <ec2-3-83-245-8.compute-1.amazonaws.com> #catalog-1
  <ec2-3-83-245-8.compute-1.amazonaws.com> #catalog-2
  <ec2-3-83-245-8.compute-1.amazonaws.com> #order-1
  <ec2-3-83-245-8.compute-1.amazonaws.com> #order-2

Docker:

- **cd docker**
- **Compile:** Execute sh **build_docker_images_locally.sh** for building the containers locally.
- **Run:** To execute this project on **docker,** execute sh **run_docker_images.sh** in the docker folder of the repository. This will run the micro-services on the local machine. More instructions on how to run the docker application will be provided in the Readme.md in the docker folder.

- The client will run on the users local system and the output will be stored in the local system. The test cases will also run on the user's local machine and the output will be stored in the local system.

The script will run all the processes and carries out the API calls in the client process.

Each test case will run and the results from the client and the tests files will be stored in the **ClientOutput.txt** and **tests.txt** in their corresponding folders.

The project structure is as follows:

Pygmy

- src
  - load balancer
    - loadBalancer.py
    - requirements.txt
  - Frontend-server
    - Frontend.py
    - cache.py
    - Requirements.txt
  - Catalog-server
    - catalog.py
    - Requests.txt (Storage for the update requests)
    - requirements.txt
  - Order-server
    - order.py
    - Requirements.txt
- Client process
  - Client.py
  - Client-output.txt
- docker
  - Src
    - Catalog server
    - Order server
    - Frontend server
    - load balancer
  - Build_docker_images_locally.sh (builds the docker images locally)
  - Run_docker_images (runs the built images)
- run.sh (Runs the servers on the local machine)
- Tests
  - Tests.py (test cases checking for search, lookup and buy requests)
  - tests.txt (Contains the outputs from the tests.py)
- Config.txt (contains the path to the key-pair and the dns of the servers)

- aws_run.sh (includes commands to run the servers on different AWS EC2 instances)

The servers: Catalog, Order and frontend operate on ports 8080, 8082 and 8081 respectively.

[2] Assumptions

The project is designed based on the following assumptions:

- Items are never restocked and the servers will respond with no result once the stock reaches 0.
- Cache is designed to be an in-memory thread safe data structure.
- Sqlite database is used to store the data in the catalog server.
- The database is assumed to handle the concurrent read/write operations.

[3] Structure

Each process/server is written in a separate folder and have the following properties:

Client-server:
Client contains the API request calls for the front-end server. It logs the requests and the corresponding responses in *ClientOutput.txt*.

Front-end server:
App implements the following routes:

- Search(topic): The requests coming on this route calls the load balancer server with the request parameters
- Lookup(itemNumber): The requests coming on this route calls the load balancer server with the request parameters
- Buy(itemNumber): The requests coming on this route calls the load balancer server with the request parameters. If there's no stock, the route responds with a message *The item is out of stock*.
- Invalidate(request): This route is called by the backend whenever there is a write request. The invalidate route removes the request key from the cache to maintain cache consistency.

Catalog-server:

Catalog implements the following routes:

- */query/<int:itemNumber>:* Queries the database to get the item with the corresponding item number, serializes it and returns the serialized result to the response if the number of result items is greater than 0.
- */query/<string:topic>*: Queries the database to get the item with the input topic, serializes it and returns the serialized result to the response if the number of result items is greater than 0.
- */updateStock/<int:itemNumber>/<int:value>:* Given the item number and the stock value, this route finds the item with the itemNumber and updates its stock value with the value from the input. It returns the updated item from the database. The server takes a lock once it reaches this route and sends the updateStock request to the replica as well.
- */reduceStock/<int:itemNumber>:* Given the item number of the catalog item, this route's functionality is to reduce the stock of the item by 1 and then return the updated item from the database. The server takes a lock once it reaches this route and sends the reduceStock request to the replica as well.
- */updateCost/<int:itemNumber>/<int:value>:* Given the item number and the cost value, this route finds the item with the itemNumber and updates its cost value with the value from the input. It returns the updated item from the database to the response. The server takes a lock once it reaches this route and sends the updateCost request to the replica as well.
- */update_replica_cost/<int:itemNumber>/<int:value>:* Updates the cost of an item in the replica.
- */update_replica_stock/<int:itemNumber>/<int:value>:* Updates the stock of an item in the replica.
- */update_replica/<int:itemNumber>:* Reduces the value of the stock of an item in the replica by 1.

Order server:

Order server implements the following routes:

- /buy/<int:itemNumber>: Given the item number as input this route calls the load balancer server with the request parameters for the item and gets the response for the matching itemNumber. If the stock of the matching number is greater than or equal to 1, it sends an updateStock request to the load balancer. The Order server then returns the response from the load balancer request to its response.

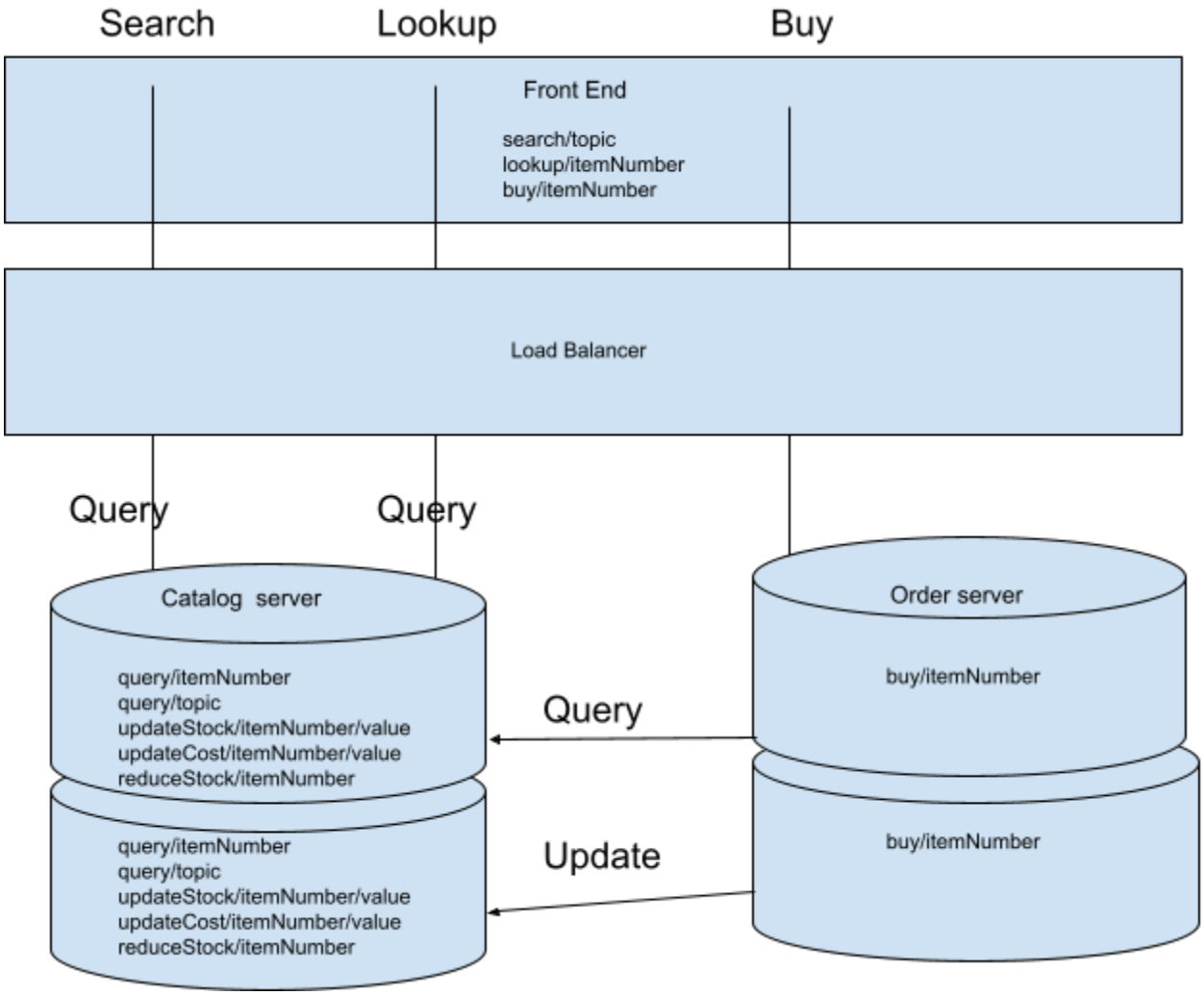
Load balancer server:

The Load balancer implements the following routes:

- /<request>: This route processes all the requests received from the front end and sends the requests to the Catalog server replicas or the Order server replicas.

[4] Design & Workflow

- The requests from the client are sent to the frontend server where it calls the load balancer server for each request.
- The front-end also has cache implemented for caching results. The cache results are stored in in-memory data structures and are thread safe. The cache has three functionalities. Put, get and erase. The frontend caches every result in the data structure and gets the response from the cache if the request appears again. If a write request is performed on the database, the catalog/backend servers send an invalidate request to the frontend to remove the cache result for the invalidated cache item.
- The load balancer has the load balancing mechanism for both the catalog servers and the order servers. Whenever a Catalog or an Order server comes alive/online, it registers itself with the load balancer making its presence known. The load balancer then uses the replica details from the registration to route requests to different servers using round-robin or least loaded mechanism. Currently, I am using a round-robin mechanism. So, the load balancer becomes the main component of the whole application. If the load balancer fails, there's no way of knowing it for now. I have implemented a heartbeat policy to check for the failed servers and if the server is found to be dead, the load balancer keeps track of the dead server and executes a command to respawn it. After respawning the server, the load balancer waits for 5 secs to make sure the server is up and then issues an db sync request for the dead server. The sync requests get the data from the replica and updates the data of the dead server.
- I have used a **primary-based** consistency protocol to keep the **write consistency** between the two catalog replicas. Whenever a write request comes to a server, the server takes the lock for that request, updates its db and then sends a request to its replica. After the execution of the update replica request on the replica server, the lock is released. So, in this scenario I'm compromising the availability of the server to accompany the consistency of the servers.
- The load balancer queries the catalog server and the order server and responds back to the frontend with the result.



Fault Tolerance

- A heartbeat thread starts as soon as the server boots up to check for the failed servers every 2 seconds.
- If a server is found to be dead, the load balancer keeps track of the dead server and executes a command to respawn it.
- After respawning the server, the load balancer waits for 5 secs to make sure the server is up and then issues an db sync request for the dead server.

- The sync requests get the data from the replica and updates the data of the recently up server.
- If a server fails, the next server/replica gets the request and the up replica executes the requests and updates its db. Once the server is up and running, the db gets synced.

[4] Design Considerations

- I have used Flask to implement the REST apis.
- The three servers start up at 3 different ports. 8080,8081,8082.
- The routes in the servers are type casted.
- In-memory cache data structure is implemented.
- Primary-based consistency protocol was used for providing the write consistency.
- The logging of the requests are done in a text file and is stored on the catalog server.
- Included requirements.txt in every server to install the dependencies on all the servers when it is run on AWS
- The load balancer is a single point of failure. If the load balancer dies, there's no way to respawn it. Also, the load balancer can become a bottleneck if there are too many requests coming through the server.

[5] Improvements and Extensions

- We could use a more strict form of consistency to improve the consistency performance.
- Paxos can be used for the state machine replication to improve the consensus among the replicas.
- Can use implementations to remove the load balancer as a single point of failure. Could implement a heartbeat mechanism on the frontend server to see if the load balancer is dead and respawn it if necessary.