Lab 3 - Design Document

Tanay P Joshi
Megha Singh

Client:

The client script simulates interactions with an online toy store server. It establishes a connection to the server, queries for toy information, and places orders based on a given probability and if the toy's quantity is positive.

The client connects with the frontend and maintains the connection to send consecutive requests and finally closes the connection when all requests are served. The client randomly selects a product from the list and sends a GET request for the product to the frontend. For every response received for the GET request, the client decides to place an order based on the probability prob. This is implemented in the following way:

The variable prob denotes the probability of placing an order, while total_requests is assigned a value of 20, representing the total number of requests to be generated. Next, the script calculates the orders_to_place by multiplying total_requests by prob and converting the result to an integer, determining the number of orders to be placed based on the given probability. Subsequently, the script randomly selects order positions by creating a set order_positions, containing randomly chosen positions within the range of total_requests, with the count matching orders_to_place. This step emulates the stochastic process of selecting requests that will result in order placements, guided by the calculated probability. The script then iterates through the generated requests, simulating their processing. Within each iteration, a flag order_request_flag is initialized, and a request is made to retrieve product information. If a valid response is received and the current request position is in order_positions, indicating a probable order placement, the flag is set, a random quantity is selected, and an order is placed via a POST request. Upon successful placement, the order number is extracted and printed, and the order information is saved to a CSV file. Additionally, the script records the request time and categorizes it based on whether an order was placed or not. Finally, if the probability is non-zero, suggesting pending orders, the script reads order numbers from the CSV file and proceeds to retrieve and process their details via additional requests.

The client does not send a POST request if the status code of the GET request for that item is 404, i.e., when the item does not exist.
For placing an order, the client randomly selects an item quantity between [1, 5].

The client is a http-client and persists connection using the header option **“Connection”: “keep-alive”**.
For each iteration, the client records the order number and order information if a purchase request was successful. Before exiting, the client retrieves the order information of each order that was made using the order query request, and checks whether the server reply matches with the locally stored order information.

When the leader node dies, the client retries the request three times - each separated by a 2 second interval.

We tested 5 clients concurrently with each client sending 20 requests where the ratio of query and order requests are decided by prob which is varied from 0 to 1 in increments of 0.2

Frontend:
The frontend connects with the client and routes its requests to the relevant backend. Client’s GET request is routed directly to Catalog service whereas POST requests are routed to order service which in turn sends request to the Catalog service. The frontend uses **requests library** to send GET and POST requests to the microservices.
The frontend exposes well-defined REST API URLs to the client to accept GET and POST requests. The frontend and the client communicate with REST API.
Frontend also maintains a cache to reduce a round-trip to the microservices for GET requests. The cache capacity is limited to 7 toys.

There are 3 order microservices processes running. Frontend conducts an election immediately when it boots up to select a leader order process while the other order microservices processes serve as replicas to implement availability through replication. This is done by setting a priority number to the order processes. The order process with the highest priority is elected to be the leader. When a leader is elected, frontend propagates the leader port number to all other replicas. The frontend uses the LRU cache eviction policy. The **evict_lru** method is defined to manage the cache size. When a new item is added to the cache and the capacity is exceeded, the least recently used item is evicted to make room for the new one. The frontend brings the most recently used item in front. Naturally, the least recent item takes the back seat which is then later evicted.

When the frontend receives a BUY/POST request from the client, it forwards the request to the Order service. The response from the order service is sent back to the client, Further logic is discussed in relevant Order and Catalog sections.

Method definitions:

1. **do_GET:** It handles incoming GET requests from the client. It checks if the path of the URL starts with “/products”, extracts the item name from the URL and forwards the request to the Catalog service.

2. **do_POST:** It handles incoming POST requests from the client. It checks if the path of the URL starts with “/orders”, extracts the request body, encodes it into stringified JSON format and forwards it to the Order Service which further routes it to the Catalog Service.

3. **get_product_details**: The do_GET method calls this function internally to actually send the request and return the response back to **do_GET**.

4. **place_order**: The do_POST method calls this function internally to actually send the request and return the response back to **do_POST**.

5. **get_session_id:** This function returns the Session-ID for a client from the request header.
The frontend closes the client connection when all requests from a client are served.

6. **check_heartbeat_status:** This function sends a ping to the leader order process to check for its availability. Response status 200 means **OK.**

7. **conduct_election:** Frontend loops over order ports and their corresponding priorities (hard-coded and stored in a dictionary) and sends requests to one order process at a time. If the process with the highest priority responds **OK**, it stops the election and declares the result to other order nodes otherwise continues until a leader is chosen.

8. **periodic_leader_check:** This function encapsulates the check_heartbeat_status and conduct_election functions. It calls heartbeat and if the response is OK, it does nothing. Otherwise, it re-conducts elections. **periodic_leader_check** is called every 3 seconds.

9. **evict_lru:** This method evicts the least recently used item from the cache if the cache size exceeds the CACHE_CAPACITY.

10**. get_order_details:** Client uses this function to get order details from the Order service to compare its order logs with the service.

The Frontend uses **Thread-per-Session** Model. All requests from a single client are handled by a thread. To do this, we use http.server.ThreadingHTTPServer. The threadpool size is dynamic.

11. **do_DELETE**: This function deletes/invalidates the item from the cache.

Output from terminal:
Thread ID: 6193885184, Session ID: 9k7V6XDhcO, Message: "POST /orders HTTP/1.1" 200 -
Thread ID: 6176485376, Session ID: yyLCZ2MjYP, Message: "GET /products/elephant HTTP/1.1" 200 -
Thread ID: 6193885184, Session ID: 9k7V6XDhcO, Message: "GET /products/whale HTTP/1.1" 200 -
Thread ID: 6176485376, Session ID: yyLCZ2MjYP, Message: "POST /orders HTTP/1.1" 200 -

Order Backend:

The Order service gets POST requests from the Frontend on behalf of the client. It further sends a decrement call to the Catalog service to reduce the item quantity by the requested amount. The Catalog service may either send a **200 OK** response or a **400 Bad Request** back to the order service. The order service forwards the response back to the frontend.

The order leader writes to its own log file and also sends a POST request to tell other replicas to write the log to their own file to maintain consistency.

When any order process starts, it immediately calls the **get_missed_orders** method provided the Leader Order Service is alive. This is done because we assume that an order service replica can die and become alive at any moment. At this point, it is safe to assume that it has missed on some orders processed and logged by the leader. In order to maintain consistency, the replica requests the leader (via a GET request) to send the missing logs. The implementation is simple:

Each order node keeps track of **latest_order_number**. It is the last index of the log csv file of the corresponding process. The node then calls **get_missed_orders** and passes this as an argument to tell the order leader that this was the last index it had knowledge of and that it needs logs from this point on.

For easy accessibility, each order node also saves order logs in a list. The **latest_order_number** serves as an index to this list. The leader iterates over starting from **latest_order_number + 1** and sends a JSON response containing all the missed orders. Upon receiving the response, the requesting node writes the records into its log file.

Method Definitions:
1**. do_POST**: Handles incoming POST requests from clients. If the request path starts with "/orders", it reads the order data from the request body, establishes a connection with the Catalog service, forwards the order data, and processes the response accordingly. If the order is successfully placed, the method also writes the order data to the csv file to maintain persistence. Since the orders file is a shared resource, a lock is required to read from and write to the file, to control access and maintain consistency.
Note: If the orders csv file exists, it initializes the counter from the last record no. otherwise the order count starts from 0.

2. **send_order_data:** This function sends a log row to be written to the other replicas

3. **get_leader_info:** This function returns the Leader Order process port number to the requesting order node. It is especially useful when a node comes back alive and needs to know who the current leader is.

4. **get_missed_orders:** As discussed above, it is used to get all the logs missed by an order replica.

5. **find_order:** Internally used to compare the sent client log with the log maintained by the order service.

6. **do_GET**: Responds to heartbeat requests, validates client order logs, etc.

The Order service also implements the http.server.ThreadingHTTPServer to achieve multi-threading.

Catalog:

The catalog receives a POST request from the order to deduct the item quantity by some requested amount. It checks if the available item quantity is enough to serve the request. If yes, it deducts the item quantity by that amount and updates the data file as well as in-memory data. The catalog also responds to Frontend's GET/QUERY requests.

It runs a restock function every 10 seconds to check if any toy is out of stock and restocks it to 100.

To maintain cache consistency, the catalog service sends an invalidation request to the frontend to delete the item from the cache which was restocked by the catalog service. Similarly, it also sends an invalidation request to the frontend after each toy purchase.

Method Definitions:

1. **do_GET**: This method is responsible for handling GET requests, it checks if the request path starts with "/products". On violation, a response with **status 404 - Page Not Found** is sent. If the requested item does not exist in the data dictionary, Catalog sends an error message with **response code 404 - product not found.** If the item exists, it sends the item details with **response code 200 - OK.** Note **-** All responses from GET requests are sent back to the frontend and then back to the client.

2. **do_POST:** This method is responsible for handling POST requests, it checks if the request path starts with "/orders". On violation, a response with **status 404 - Page Not Found** is sent. If the requested item exists and the requested order is less than or equal to the available quantity, the quantity is reduced by the requested amount. The cost of the order is the cost of the item times the quantity requested. The catalog sends **response code 200 - OK** along with the response - item and order cost (a JSON object). At the same time, the catalog also updates the corresponding item quantity in the csv file to reflect the change. If the quantity is not sufficient, **response code 400 - Bad Request** is sent back along with error body - response code and error message. Note - all responses from POST requests are sent back to the Order Microservice which in turn sends them back to the frontend and back to the client.

3. **restock**: This function loops over toys and restocks any out of stock toy back to 100 and sends an invalidation (DELETE) request for that item to the frontend.

4. **schedule_restock**: This is just a wrapper function that calls the restock function every 10 seconds.

The Catalog Service also implements a locking mechanism since in-memory items and the persistent csv file is a shared resource. The Catalog service also implements the http.server.ThreadingHTTPServer to achieve multi-threading.

Communication:
Every microservice communicates with each other and the frontend using REST APIs. In

our system architecture, communication between services, the frontend, and clients relies heavily on clearly defined API endpoints. We ensure that these interfaces send HTTP requests to specific endpoints provided by backend services, like the catalog service or order service.