Lab 3 Design Document

client.py

Overview:
The client code is for a client that interacts with a server using HTTP requests. The client makes requests to the server to look up the information about toys, place orders to buy with certain probability. It can also lookup previous order information stored in the remote server database by providing the Order number.

The client **first sends a lookup request** to the frontend server with a random toy name, then it will **make a follow-up order request with a pre-determined probability**. The probability p can be assigned through arguments. We have considered different probabilities for buying requests from 0 to 80%, with an increment of 20% i.e 0, 0.2, 0.4, 0.6 and 0.8.

Before exiting, the client will check the successful order IDs stored in its memory, **retrieving the corresponding order information from the order services** by sending **GET /orders/<order_number>** to check whether the server reply matches the locally stored order information. If the information from servers is mismatched with the locally stored records, there will be messages showing on the screen. Otherwise, the program exists and shows average lookup and order request latency on the terminal.

Design Choices
The code above is a Python script that implements a client to interact with a toy server. The client sends HTTP requests to the server to query toy information and place orders. Here are some design choices made in the code:

- **Session management:** The code uses a requests.Session object to make multiple requests to the same server while reusing the underlying TCP connection. This improves performance by reducing the overhead of establishing a new connection for each request.
- **Error handling:** The code handles exceptions when making requests and retries the requests if they fail.
- **Data storage:** The code stores the information of each successful order in a dictionary and checks if the server reply matches the locally stored data before exiting. This ensures data consistency between the client and server.
- **Random toy selection:** The code randomly selects a toy from a list of available toy names. This simulates a user choosing a toy to order.
- **Sleep function:** The code uses the time.sleep function to wait for a short period of time between requests. This can help prevent overwhelming the server with too many requests at once.
- **Handling of response status codes:** The code checks the status code of the HTTP response and continues to send requests if the lookup() request fails.

- **Order request probabilities:** The code uses a probability parameter to determine the likelihood of sending an order request. This allows the user to specify the risk tolerance of the order algorithm. prob >= random.uniform(0.,.8): **With a given probability order_prob, the code sends an additional order request.**
- **Latency calculation:** The code measures the average lookup and order request latency and prints it for debugging purposes.
- **Locally stored order information:** The client saved successful order information via a locally maintained dictionary, which will be used to check data consistency across servers and clients before exiting.

API Documentation

This program is a client that makes sequential requests to a server to retrieve toy information and place orders. It is built with Python and uses the requests module to send HTTP requests

Usage

To run the client, execute the following command in the terminal:
python client.py --prob PROBABILITY
The available command-line arguments are as follows:
--prob: The probability of making an order request after each successful lookup request. Default is 0.5

Functionality

The client works as follows:

1. It extracts the arguments from the .env file.
   --port: The port of the front-end server.
   --host: The host of the front-end server.
2. It initializes a session with the server to reuse the underlying TCP connection.
3. It loops through the number of requests.
4. For each request, it chooses a random toy name from a predefined list of toys.
5. It sends a GET request to the server's /lookup endpoint with the chosen toy name as a parameter.
6. If the response code == 200 and probability of making an order is greater than a random number, it sends a POST request to the server's /order endpoint with order data.
7. If the order request is successful, it records the order information. After all requests have been made, the client sleeps for 1 second to ensure all requests are processed by the server.
8. It then retrieves the order information and checks if it matches the locally stored data.
9. Finally, it prints the average lookup request latency, the average order request latency (if there were any order requests), and any inconsistencies between the local and server databases (if any).

The client interacts with the server using the following HTTP requests:

GET /product?<toy_name>
This endpoint retrieves the information about the specified toy.

Request parameters
- toy_name (required): The name of the toy to retrieve information about.

Response
- If the request is successful, the endpoint returns a JSON object with the following fields:
    - status: The status of the request. Can be a success or error.
    - data: An object containing the information about the toy. The fields in this object are:
        - name: The name of the toy.
        - quantity: The quantity of the toy available.
        - price: The price of the toy.

POST /order
This endpoint places an order to buy a toy.

Request parameters
- toy (required): The name of the toy to buy.
- quantity (required): The quantity of the toy to buy.

Response
- If the request is successful, the endpoint returns a JSON object with the following fields:
    - status: The status of the request. Can be a success or error.
    - data: An object containing the information about the order. The field in this object:
        - order_number: The number of the order generated to buy the toy.

GET/order?<order_number>
This endpoint retrieves the order information (order number, toy name, quantity) about the specified order number.

Request parameters
- order_number (required): To retrieve information about the placed order.

Response
- If the request is successful, the endpoint returns a JSON object with the following fields:
    - status: The status of the request. Can be a success or error.
    - data: An object containing the information about the toy. The fields in this object are:
        - order number: The order number of the order.
        - name: The name of the toy.
        - price: The price of the toy.

Frontend.py

Overview:
The frontend code includes an implementation of a Least Recently Used (LRU) cache and leader selection for a distributed system. The LRUCache class is implemented with a Read-Write Lock and uses an OrderedDict to store items. The leader selection function uses a HTTP to send and receive messages from order servers, selects a leader, and notifies the other replicas. The code implements a simple HTTP server that handles GET and POST requests for a toy and an order service.
The code also defines a threaded HTTP server that allows for multiple concurrent requests. It is extending the built-in HTTPServer class and using the ThreadingMixin class to make it threaded. The server is designed to receive requests related to toys, and route them to different order services based on the requested stock symbol.
The frontend file implements a front-end service that communicates with catalog and order services over HTTP. It listens for incoming HTTP requests, handles GET and POST methods, queries the catalog service for product information, and places orders through the order service. The implementation utilizes a threaded HTTP server for concurrency, ensuring efficient handling of multiple client requests concurrently.

Design Choices:
The **LRUCache** implementation includes a Read-Write Lock to ensure thread safety when accessing the cache. The lock is used to restrict multiple threads from writing to the cache simultaneously and to allow concurrent reads. This ensures the cache remains consistent across threads and does not lead to data races.

The cache is implemented with an OrderedDict, which allows for O(1) insertion, deletion, and updating of elements. This data structure is ideal for implementing an LRU cache as it maintains the order of elements based on their last access time.

The **leader_selection** function plays a pivotal role in our system's architecture, particularly in the dynamic identification and establishment of a leader among available order servers. This process is crucial for maintaining system stability, ensuring efficient coordination, and enabling fault tolerance mechanisms.

The function begins by attempting to connect to order servers, starting from the highest ID order id (order_id='3') and iterating through the list of servers. It utilizes HTTP requests, specifically GET and POST methods, to send ping messages (/ping) to each server, assessing their operational status based on the response received. Upon successfully identifying an operational server, it designates that server as the leader, signaling this status change through a /leaderselection API and also informs all replicas within the system about the newly elected leader through /inform_replica API

The **FrontEndRequestHandler** class inherits from http.server.BaseHTTPRequestHandler to handle HTTP requests to the catalog and order services. The do_GET() method overrides the

default do_GET() method to handle HTTP GET requests. The do_POST() method overrides the default do_POST() method to handle HTTP POST
requests.

The **FrontEndService** class represents the front-end service itself, responsible for handling client requests and interacting with the catalog and order services. It subclasses ThreadedHTTPServer.. It initializes the front-end service with the provided port number and addresses of the catalog and order services.

- Within this, the *query_product* method sends a GET request to the catalog service to query information about a specified product. It handles the response from the catalog service, processing the product information or returning an error message if the product is not found.
- The *place_order* sends a POST request to the order service to place an order for a specified product. Handles the response from the order service, processing the order number or returning an error message if the order cannot be placed.
- The *query_order_number* function orchestrates order queries to order service by selecting the appropriate order server via the leader ID, forming and executing GET requests, and handling responses.

Error handling is implemented to handle cases where the catalog or order service is unavailable or returns errors. It provides meaningful error messages in the response to inform clients about the encountered issues.

- When a GET request is received, the code checks the URL's validity and extracts the toy name. If the cache is used, it checks the cache for the toy's information; if the cache misses, it forwards the request to the catalog server to retrieve the toy information. If the toy information exists, it sends the response to the client and updates the cache if necessary. If the toy information does not exist, it sends a 404 response to the client.
- When a POST request is received, the code checks the URL's validity and extracts the stock name, quantity, and type. It forwards the request to the order server to add a new order. If the order number exists, it sends the response to the client. If the order number does not exist, it sends a 404 response to the client.

Send_Custom_Response Method constructs and sends a custom HTTP response to the client, according to the input received and sets the status code and the appropriate response headers i.e. content-type based on the response data. It encodes the response data as JSON and writes it to the response body.

Threaded HTTP server : As we expect a high volume of requests and want to handlethem concurrently.

External Dependencies: The implementation relies on the requests library to communicate with the catalog and order services over HTTP.

Using environment variables: To make the server more configurable and easier to deploy in different environments.

API Documentation:

RWLock class:

__init__(self) : Initializes the RWLock class.
r_acquire(self) : Acquires the read lock.
r_release(self) : Releases the read lock.
r_locked(self) : A context manager for acquiring the read lock.
w_acquire(self) : Acquires the write lock.
w_release(self) : Releases the write lock.
w_locked(self) : A context manager for acquiring the write lock.

LRUCache class:

__init__(self, capacity) : Initializes the LRU cache with the specified capacity.
get(self, key) : Gets the item associated with the key. If the key is not found, returns -1.
put(self, key, item) : Puts an item in the cache, with the specified key. If the key already exists in the cache, replaces the item associated with the key.
pop(self, key) : Removes the item associated with the key. If the key is not found, returns -1. If the operation is successful, and returns 1.

leader_selection(server)

server : A parameter that represents the server. Tries to connect to an order server, starting from the highest id.

The implementation of a stock request handler class that handles HTTP GET and POST requests.

Attributes:

● self.server - This attribute is used to access the server object that is created with http.server.HTTPServer. It has attributes like catalog_host_url, catalog_port, order_request_addrs, cache, etc.
● self.path - This attribute is used to get the URL path of the incoming request.

1. GET /product/{product_name}:

Overrides the default do_GET() method of BaseHTTPRequestHandler class. Retrieves information about a product from the catalog service.

- Parameters: product_name: Name of the toy to query.
- Response:
    - 200 OK: Returns JSON data with information about the product - Name, Price, and Quantity.
    - 404 Not Found: Returns an error message if the product is not found.
    - Other HTTP error codes may be returned for various error scenarios, so the error json is created and returned.

2. GET /order/{order_number}:

This URL is used to query an existing order.
The do_GET() method extracts the order number from the URL query parameter and forwards the request to the Order server using HTTP GET request with the order number as a query parameter.

- Response:
  - 200 OK: Returns JSON data with information about the order - Order number, product name, and quantity.
  - 400 If the order number doesn't exist.
  - If the URL is invalid raise RuntimeError

3. POST /orders/{product_name}/{quantity}:

This method overrides the default do_POST() method of BaseHTTPRequestHandler class. Place an order for a product through the order service.

- Request Body: JSON data containing details of the order i.e. Product name and Quantity.
- Response:
  - 200 OK: Returns JSON data containing the order number.
  - 400 Insufficient Stock: Returns an error message if the requested product is out of stock or if the required quantity is invalid.
  - If the URL is invalid raise RuntimeError

ThreadedHTTPServer Class

This class extends the ThreadingMixIn and HTTPServer classes to allow for multiple concurrent requests. It overrides the __init__ and server_bind functions to initialize and bind the server, respectively.

_init__(self, host_port_tuple, catalog_address, order_ids,cache_s,c)

host_port_tuple: A tuple of (host, port) to which the server should bind.
catalog_address: The address of the catalog service that the front-end server will communicate with.
order_ids: A list of order IDs associated with the front-end service.
cache_s: The size of the cache, used for caching frequently accessed data.
c: A boolean flag indicating whether caching should be enabled or not.

catalog.py

Overview:
The code defines five functions that work with a catalog file containing information about various toys. The init_catalog() function initializes the catalog file with some initial data if it
does not exist already. The lookup() function performs a lookup operation on the catalog file to check if a toy exists or not and if it exists then returns the price and quantity of a given toy. The order() function performs an order operation on a given toy for a valid quantity. The update() function updates the catalog csv with the current catalog data after any modifications. The restock() function keeps on restocking the quantity if it is less than or equal to 10.

The code defines two classes -

CatalogServer - Manages a toy catalog using a hashmap data structure for efficient lookup and manipulation of toy information.
Implements a background thread for periodic restocking of toys based on a predefined interval.
CatalogRequestHandler - Handle HTTP requests for a toy catalog, and a subclass ThreadedHTTPServer of HTTPServer that uses threading to handle multiple requests concurrently.

The CatalogRequestHandler class has two methods for handling GET and POST requests, do_GET and do_POST, respectively.

The do_GET method handles requests for toy lookup, and if the requested stock is found in the catalog, it returns its data as a JSON response; otherwise, it returns an error response.

The do_POST method handles requests from the order service to order a specified toy. It parses the request to get the toy name, quantity, and orders the specified toy according to the request if the required quantity is available. It sends an appropriate response based on the result of an order operation, and sends an invalidation request to the frontend. If the frontend response status code is not 200, it raises a RuntimeError.

The ThreadedHTTPServer class overrides the __init__ method to save metadata in the server, including the protocol version, the frontend host and port.

In terms of design choices, the use of CatalogServer streamlines toy catalog management by handling, automated restocking, and cache synchronization, enhancing system reliability and scalability. The CatalogRequestHandler class allows for easy handling of HTTP requests for the toy catalog. The ThreadedHTTPServer subclass allows for concurrent handling of multiple requests, improving performance. The use of a reader-writer lock ensures thread-safety when accessing the catalog file. The sending of an invalidation request to the frontend ensures that the frontend's cache is updated when an order or restock occurs, preventing stale data from being displayed.

Design Choices

- The max_workers count in ThreadPoolExecutor is set to the number of available CPU cores to efficiently utilize system resources.
- The functions use a reader-writer lock (rwlock) to ensure that access to the catalog file and DB is synchronized between threads or processes.The locking mechanism is defined in the RWLock class, and it is implemented with the use of Lock objects provided by the threading library.
- The lookup() function only requires read access to the file, while the init_catalog(), order() and restock() functions require write access.
- The use of a reader-writer lock allows multiple threads or processes to read from the file simultaneously, while ensuring that only one thread or process at a time can modify the file.
- The initial data for the catalog in memory is provided through the csv file in the init_catalog() function, with the toy name, price, and quantity. This allows for easy addition or modification of toy data.
- The lookup() function returns the data of the requested toy if it exists in the in-memory DB, or -1 if it does not. This allows the calling code to detect when a requested toy is not in the catalog.
- The order() function returns an integer indicating the result of an order operation. A value of 1 indicates a successful order, while negative values indicate various error conditions such as invalid quantity or insufficient available stock. This allows the calling code to handle different order outcomes appropriately.
- The restock() function periodically checks toy inventory at a given time interval (Eg- 10s) and increases quantities for out-of-stock items i.e. stock <= 10, ensuring catalog availability. Implementing this in a separate thread enhances system responsiveness and resource utilization, crucial for efficient handling of user requests and scalability during peak loads.
- The invalidation_request() function sends an update to the frontend whenever there is restock or purchase of the toy. So that changes can be made over the cache also.

The code defines the start_server function in CatalogServer class, which is responsible for running a server. It starts by the __init__ method of the class, initializes key components of the server, sets the server's port and CSV file path, creates an empty catalog dictionary for storing toy data, and establishes a read-write lock (RWLock) for concurrency management. Additionally, it enables or disables caching based on the provided boolean parameter through .env. Furthermore, the method starts a separate thread (restock_thread) to periodically restock toys, ensuring continuous availability and optimal resource utilization. Next, the instance of the class creates a ThreadedHTTPServer instance that listens on the specified port and handles requests with the CatalogRequestHandler class. Finally, the function starts serving requests on the specified port using httpd.serve_forever().

Overall, our code makes use of environment variables to provide flexibility in configuring the server. It also uses a threaded server to handle multiple requests concurrently, which is a good choice for a high-traffic server.

API Documentation:
The server provides the following API:

GET /product?<toy_name>

- Description: This endpoint is used to handle GET requests from the frontend service
- Parameters:
  - toy_name: The name of the toy to look up.
- Returns:
  - If the request URL is invalid, raise RuntimeError
  - If the toy name exists in the catalog file, returns a JSON object containing the price and quantity of the toy. Example: {'data': {'name': 'Fox', 'price': 20.99, 'quantity': 9903248}}
  - If the toy name does not exist in the catalog file, returns a JSON object containing the error message. Example: {"error": "Toy not found"}
- Response Codes:
  - 200 OK: The order was successfully processed.
  - 404 Not Found: The requested toy could not be found in the catalog service.

POST /orders?<toy_name>?<quantity>

- Description: A method that handles POST requests from the order service.
- Parameters:
  - toy_name: The name of the toy to be ordered.
  - quantity: Number of toys that are to be purchased.
- Returns:
  - If the request URL is invalid, raise RuntimeError
  - If the quantity is invalid or excessive, return a 400 Bad Request error response with a plain text message.
  - If an order operation is successful, return a 200 OK response with a plain text message.
  - Send an invalidation request to the frontend. If the request fails, raise RuntimeError.

ThreadedHTTPServer(ThreadingMixIn, HTTPServer):
Overrides the __init__() method of HTTPServer to save metadata in the server.
Parameters:

- host_port_tuple: A tuple containing the hostname and port number to listen on.
- streamhandler: A subclass of BaseHTTPRequestHandler that handles incoming requests.

order.py

Overview:
The code defines a HTTP server that serves requests for placing orders for toys. The server receives POST requests, extracts the parameters for the order, validates the request parameters, and then sends a request to another catalog service to check for stock availability, and places the order if the quantity is enough. The program is written in Python, and the HTTP server is implemented using the http.server module. Here, we also implement Replication and Fault Tolerance.

Design Choices:
The program has a single class OrderRequestHandler which inherits from the http.server.BaseHTTPRequestHandler class. The class OrderRequestHandler handles the POST requests, validates the order parameters, and forwards the request to another catalog service for checking the availability of the toy and its stock. The server is multithreaded to allow multiple clients to connect to the server concurrently. The multithreading is implemented using the ThreadingMixIn class provided by the socketserver module. The ThreadPoolExecutor class is used to manage a pool of threads for executing incoming requests.
It creates a pool of worker threads with a maximum number of workers set to the number of CPU cores (multiprocessing.cpu_count()). When a new request is received, the executor assigns it to an available thread from the pool, allowing multiple requests to be processed concurrently. The program uses a read-write lock to ensure thread safety when accessing the catalog service.

OrderRequestHandler class:
The OrderRequestHandler class handles incoming requests and sends the requests to the catalog service. The code handles ‘order’ requests from a client and forwards them to the catalog service. It first checks if the requested toy is present in the catalog service then it validates that catalog service has enough stocks available by the order function for buy by making a request to it and retrieving the remaining quantity of the specified stock. If there are enough stocks available, it then places the order by sending a POST request to the catalog service with the required parameters such as toy name and quantity. It then checks if the order is successful by verifying the response status code, and if not, raises a runtime error. It then updates the order log with the order number, toy name, quantity, and writes the order log into orders.csv. If the order is successful, it sends a response to the client with the order number and a CSV file containing the order log. However, if there is not enough remaining quantity or toy not present in the catalog file, it sends an error response to the client with an appropriate error message. The class is responsible for validating the request parameters and returning an appropriate response to the client.

The max_workers count in ThreadPoolExecutor is set to the number of available CPU cores to efficiently utilize system resources.

Methods:
The OrderRequestHandler class has three methods, do_POST(), do_GET() and send_response().

- do_POST(): This method handles POST requests. It reads the content of the request, validates the request parameters, and forwards the request to the catalog service.
- do_GET(): Frontend forwards the order number to the order service then this method retrieves the order information from the order number.
- send_response(): This method sends a response back to the client to encode and decode JSON data in HTTP requests and responses.

Helper Functions:

1. init_order:
   Purpose: Initializes the order dictionary from data in a CSV file.
   Parameters: csv_file (str) - Path to the CSV file containing order data, order (dict) - Dictionary to store order data.
   Functionality: Reads data from the CSV file using csv.DictReader, parses each row, and populates the order dictionary with order number as key and toy name and quantity as values.
2. update_csv:
   Purpose: Updates the CSV file with new order data.
   Parameters: order_number (str) - Order number, product_name (str) - Name of the toy, update_quantity (float) - Updated quantity.
   Functionality: Checks if the CSV file exists, determines the mode of file opening (write or append), updates the order dictionary with the new order data, and writes the data to the CSV file using csv.DictWriter.
3. order_lookup:
   Purpose: Look up an order number in the order dictionary.
   Parameters: ordern (int) - Order number to look up.
   Functionality: Uses a read lock to ensure thread safety while accessing the order dictionary. Checks if the order number exists in the dictionary and returns 1 if found, else returns -1.
4. get_latest_order_number:
   Purpose: Retrieves the latest order number from the CSV file.
   Parameters: None.
   Functionality: Checks if the CSV file exists, reads the file using csv.DictReader, iterates through the rows to find the latest order number, and returns it. If the file doesn't exist, it returns 0.

API Documentation:
This API provides a service to handle stock order requests by communicating with the catalog service and trading service.

OrderRequestHandler:

1. API Endpoint : POST /**orders?<toy_name>/<quantity>:**
This is the main class of this API that handles POST requests to buy a given toy. The endpoint takes the following parameters in the request body:

● toy_name: The name of the toy being bought. Required.
● quantity: The quantity of the toy being bought. Required.

Once this api endpoint is called at the leader,i.e, whenever a new request comes to the leader, it asks all other replicas to first synchronize themselves. It hence calls the /synchronize endpoint at the replicas. Once all the replicas are synchronized, the leader fetches the toy name and quantity and updates its own csv with a new order number.
Further, it also calls the /replicate_order endpoint at the replicas, for the replicas to update the new order details in their own databases.

The following HTTP status codes are returned:

● 200 OK: The order was successfully placed.
● 400 Bad Request: Invalid order, for example invalid quantity or invalid order type.
● 404 Not Found: The requested toy could not be found in the catalog service.

The endpoint communicates with the catalog service to check the stock availability and place the order. The endpoint uses the RWLock class to ensure only one thread can check the remaining quantity and place an order at a time.

2. API Endpoint: POST /**inform_replica**
Request Body:
The request body contains the following information:
leader_id: The ID of the replica node that is becoming the leader.
Request Processing:

Retrieves the request body to extract the ID of the new leader.
Uses parse_qs function to parse the query string from the request body.

Leader Identification:
Uses the leader's ID to retrieve its host and port information from environment variables (ORDER_HOST_{leader_id} and ORDER_PORT_{leader_id}).
Constructs the leader's address (leader_address) using the host and port information.

Leader Notification:
Prints a message to indicate that the specified replica node is now the leader.
Sends an HTTP response with status code 200 and a JSON-encoded message confirming the leadership change to the requesting replica node.

3. API Endpoint: POST /**leaderselection**
If this current service is a leader, it would use this endpoint to assign the leader_id. If it contains a message saying this service is a leader, it would send a response 200.
Input:
leader_id: The ID of the node participating in the leader selection process.
message: A message indicating the outcome of the selection process (e.g., "You win").
Output:
HTTP response with status code 200 if the leader selection request is successful.
JSON-encoded message confirming the outcome of the leader selection process.

4. API Endpoint: POST /**replicate_order**
When a replica server is called at this endpoint, it fetches the order number, name and quantity and updates it in its own database.

5. API Endpoint: GET **/orders**
The "order lookup" endpoint is designed to retrieve information about a specific order by its order number. It allows clients to query the system for details about an order, such as the toy name and quantity, based on its unique identifier.

Input: order_number
Output:
HTTP response with status code 200 if the order lookup is successful.
JSON-encoded response containing details of the order, including:
Order number: The unique identifier of the order.
Toy name: The name of the toy in the order.
Quantity: The quantity of the toy in the order.
HTTP response with status code 400 if the order number is not found.

6. API Endpoint: GET **/ping**
When a request is made to the "/ping" endpoint, the server performs a health check and responds if everything is normal.
It checks if the query parameter "message" exists and if its value is "Ping". This ensures that the request is intended for a health check.
If the validation passes, it sends a response with HTTP status code 200 (indicating success).
It sets the response header "Content-type" to "application/json", indicating that the response body will be in JSON format.

It writes a JSON object as the response body, containing a message indicating that the "Order service is responsive".

7. API Endpoint: /**synchronize**

The "synchronization" endpoint is responsible for synchronizing data between the follower replicas in a distributed system. It ensures that all replicas have up-to-date information by fetching missed orders from other replicas, and updating the replicas' databases accordingly.

Output:

HTTP response with status code 200 upon successful synchronization.
JSON-encoded response indicating that replicas are synchronized.

Request Handling:

Synchronization Process:

- Acquires a write lock to ensure thread safety during the synchronization process using the rwlock.
- Iterates over each follower replica's address in self.server.follower_addresses.
- Checks if the current server address is not equal to the follower address to avoid synchronizing with itself.
- Sends a GET request to the follower replica's /get_last_order_number endpoint to retrieve its last synchronized order number.
- Compares the last synchronized order number of the follower with the replica's latest order number.
- If there are missed orders on the follower, initiate synchronization by calling the synchronize_replica function.
- Handles exceptions if any errors occur during the synchronization process.

Synchronize function:

Input Parameters:

last_order_number: The last synchronized order number of the replica.
replica_address: The address of the replica to synchronize with.

Functionality:

Acquiring Lock: Initializes a read-write lock (rwlock) to ensure thread safety during database updates.

Requesting Missed Orders:

Constructs the URL for fetching missed orders from the specified replica using the replica_address and last_order_number.
Sends a GET request to the constructed URL to retrieve missed orders.

Handling Response:

Checks the HTTP response status code:
If the response status code is 200 (OK), indicating successful retrieval of missed orders:

Parses the JSON response to extract missed orders.
Acquires a write lock to update the replica's database.
Iterates over each missed order and updates the replica's database with the order details using the update_csv function.
Prints a success message indicating that the replica synchronization was successful.
If the response status code is not 200:
Prints a failure message indicating that replica synchronization failed.

Error Handling:

Utilizes a try-except block to catch any exceptions that may occur during the synchronization process.
Prints an error message if an exception is encountered.

8. API Endpoint GET /**get_missed_orders**

The purpose of this API endpoint is to retrieve missed orders from the other replicas that have occurred after a specified order number.
This api goes to the other replicas and where the other replicas access their db and retrieve all the missed orders from that particular order_number.
It basically reads through the csv, and if it encounters any order number greater than the parameter provided, it would append it to the list of missed_orders and return the same.
It sends an HTTP response with status code 200 (OK) to indicate that the request was successful.

9. API Endpoint GET /**get_last_order_number**

The purpose of this API endpoint is to retrieve the latest order number stored in the system.

Functionality: Fetching the Latest Order Number:
Invokes the get_latest_order_number function to retrieve the latest order number from the csv file.
If successful, it sends an HTTP response with status code 200 (OK) to indicate that the request was successful.

The OrderRequestHandler class is not an API endpoint but it is the class that implements the endpoint for handling requests.

<u>Example request:</u>

API - Get /product/Tux HTTP/1.1
Toy Lookup Response: {'data': {'name': 'Tux', 'price': 25.99, 'quantity': 76}}

POST /orders HTTP/1.1
Host: example.com
Content-Type: application/json
Order request: {'name': 'Fox', 'quantity': 2}

Example response:
HTTP/1.1 200 OK
Content-Type: application/json
Buy Request Response: {'data': {'order_number': 61}}

RWLock:
This class provides a read-write lock to ensure thread safety when accessing shared resources. It provides two context managers for acquiring and releasing the lock for reading and writing respectively:
● r_locked(): Acquire the lock for reading.
● w_locked(): Acquire the lock for writing.

Caching
The front-end server maintains a in-memory LRU cache the records the toy information of the last n `GET /product/<toy_name>` requests. Upon receiving a toy query request, it first checks the cache to see whether it can be served from the cache. If not, the request will then be forwarded to the catalog service, and the result returned by the catalog service will be stored in the cache. Cache consistency is maintained by `POST /invalidation/<toy_name>`, which causes the front-end service to remove the corresponding toy from the cache.

Replication & Consistency
The front-end service will contact each replica via the HTTP to perform leader selection. The front-end maintains a `leader_selection()` function, which will send a "Ping " message to each replica, starting from the node with the highest ID, and then wait for the responses. Upon receiving the first "OK" reply from the node, the front-end will assign it as the leader and notify the result towards other replicas.

Propagation
When an order request or an order query request arrives, the front-end service only forwards the request to the leader. In case of a successful order, the leader node will propagate the information to the follower nodes by sending a `POST /replicate_order` request to the follower nodes.

Fault Tolerance
When the front-end service finds that the leader node is unresponsive, it will redo the leader selection process. This can be achieved by two mechanisms: a passive health check and an actively regular health check.

Passive health check
Whenever the front-end forwards a request to the leader, it uses a try-except block to capture the `requests.exceptions.RequestException`. In case of a connection failure, the front-end will call `leader_selection()` to start a leader selection, finding an alive replica, broadcasting the new leader, and then redirect the request to it.
Active health check
The frontend will regularly send "PING" to the current leader to see if the leader is alive. The time interval between each active health check can be changed through the configuration file. If the leader has no response, the frontend will restart leader selection to find an available order server of the highest ID

In our design this is achieved through a passive health check as one drawback of implementing active health checks is the potential impact on system latency. Continuously sending "PING" requests can introduce additional network overhead and processing load on the server, leading to increased response times for other critical operations.Furthermore, since Python is primarily single-threaded, running active health checks on a separate thread may not be straightforward, limiting the concurrency and scalability of the health check mechanism within our system.

Resuming mechanism
When a replica comes back online from a crash, it will first look at its database file to get the latest order number that it has. After that, it will send a `GET /synchronize` request to its peers, which will return the latest order number saved in their database. Thus, the resuming replica knows exactly what the orders it has missed since the previous crash. Afterwards, it will send sequential `GET /get_missed_orders/<{last_order_number}>` requests to the node that maintains the latest order information to acquire the missing orders.