Tung-I Chen 33609886; Pranav 33982032

Lab 3 Design Document

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 stocks, place orders to buy or sell a stock with certain probability. It can also lookup previous order information stored in the remote server database by providing the transaction ID.

The client **first sends a lookup request** to the frontend server with a random stock name, then it will **make a follow-up order request with a pre-determined probability**. The **random seed, number of total lookup requests** and **probability p** can be assigned through arguments and the configuration file.

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:

- **Session management**: The code uses a single session object to make multiple requests to the same host, which allows reusing the underlying TCP connection and thereby improving performance.
- **Random seed**: The code uses a random seed to ensure reproducibility of the requests made by the client.
- **Error handling**: The code handles exceptions when making requests and retries the requests if they fail.
- **Argument parsing:** The code uses argparse to handle command-line arguments, which makes it easier to modify the parameters without changing the source code.
- **Data storage**: The code stores the information of each successful order in a list and checks if the server reply matches the locally stored data before exiting. This ensures data consistency between the client and server.
- **Random stock selection**: The code randomly selects a stock from a list of available stock names. This simulates a user choosing a stock to trade.
- **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.

Tung-I Chen 33609886; Pranav 33982032

- **Trading request probabilities**: The code uses a probability parameter to determine the likelihood of sending a buy or sell trading request. This allows the user to specify the risk tolerance of the trading algorithm.
- order_prob >= random.uniform(0., 1.): **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 stock information and place orders. It is built with Python and uses the requests module to send HTTP requests.

The client is configurable with several parameters that can be set using command-line arguments, including the server's host and port, the number of requests to make, the probability of making an order, and a random seed.

Usage

To run the client, execute the following command in the terminal:
python client.py [--port PORT] [--host HOST] [--prob PROBABILITY] [--seed SEED] [--n_request N_REQUESTS]

The available command-line arguments are as follows:

- --port: The port of the front-end server. Default is 8080.
- --host: The host of the front-end server. Default is '0.0.0.0'.
- --prob: The probability of making an order request after each successful lookup request. Default is 0.5.
- --seed: The random seed to use for generating random stock names. Default is 0.
- --n_request: The number of sequential requests to make. Default is 20.

Functionality

The client works as follows:

1. It extracts the command-line arguments passed to it.
2. It initializes a session with the server to reuse the underlying TCP connection.
3. It loops through the number of requests specified in the command-line argument.
4. For each request, it chooses a random stock name from a predefined list of stock names.
5. It sends a GET request to the server's /lookup endpoint with the chosen stock name as a parameter.
6. If the 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.

8. After all requests have been made, the client sleeps for 3 seconds to ensure all requests are processed by the server.
9. It then retrieves the order information and checks if it matches the locally stored data.
10. 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).

Tung-I Chen 33609886; Pranav 33982032

Design Document - 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 socket 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 to a stock 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 stock prices, and route them to different order services based on the requested stock symbol. The server caches stock prices and allows the cache to be dumped if necessary. It also sends regular health checks to the order services to determine which service is the leader. Also, the server can be configured using a YAML configuration file or environment variables. The YAML configuration file contains the host, port, and other settings for the server, while the environment variables are used to overwrite the settings in the YAML file if they are set.

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 uses sockets to communicate with the order servers. The function tries to connect to an order server, starting from the highest ID (order_id='3') and sends a ping message to receive a response. If a response is received, the leader is selected, and a message is sent to notify the other replicas of the new leader. If no response is received, the function moves to the next order server and repeats the process.
The leader selection function is active, using a separate thread to periodically check the health of the current leader. If the leader is found to be dead, the function re-does the leader selection process.
A log file is implemented to store cache dump data in JSON format. This log file allows for the cache to be reconstructed if the server fails or is restarted. The JSON data is dumped into the log file periodically to avoid the overhead of continuously writing to the log file.

The StockRequestHandler class inherits from http.server.BaseHTTPRequestHandler to handle HTTP requests to the stock 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.

Tung-I Chen 33609886; Pranav 33982032

When a GET request is received, the code checks the URL's validity and extracts the stock name. If the cache is used, it checks the cache for the stock's information; if the cache misses, it forwards the request to the catalog server to retrieve the stock information. If the stock information exists, it sends the response to the client and updates the cache if necessary. If the stock 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.

Using a threaded HTTP server : As we expect a high volume of requests and want to handle them concurrently.
Caching stock prices : To reduce the number of requests to the order services, which can be slow or have a limit on the number of requests they can handle.
Sending regular health checks to the order services : To determine which service is the leader and prevent sending requests to a service that is down.
Using a YAML configuration file/environment variables : To make the server more configurable and easier to deploy in different environments. The YAML file can be version controlled, while the environment variables can be set in different deployment 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: int, log_path: str) : Initializes the LRU cache with the specified capacity and log_path.
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, returns 1.
dump(self) : Dumps the contents of the cache to the log file.

Tung-I Chen 33609886; Pranav 33982032

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

health_check(server, time_interval)
server : A parameter that represents the server.
time_interval : A parameter that represents the time interval for health checks.
Active health check: if the leader is dead, re-do leader selection.

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

Methods:

- **do_GET()** - This method overrides the default do_GET() method of BaseHTTPRequestHandler class. It handles the GET requests for stock lookup and order query.
- **do_POST()** - This method overrides the default do_POST() method of BaseHTTPRequestHandler class. It handles the POST requests for stock orders.

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.

GET Requests:

- **/lookup?stock=<stock_name>** - This URL is used to lookup the details of a particular stock. The do_GET() method extracts the stock name from the URL query parameter and checks whether the request can be served from the cache. If it's a cache miss, the request is forwarded to the Catalog server using HTTP GET request with the stock name as a query parameter. If the response status code is 200, the response is sent to the client and the cache is updated (if cache is used). If the stock name does not exist in the catalog, a 404 error response is sent. If it's a cache hit, the response is fetched from the cache and sent to the client. If the cache is used, the log is dumped after every lookup request.
- **/order?order_number=<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. If the response status code is 200, the response is sent to the client. If the order number does not exist, a 404 error response is sent. If the URL is invalid, a RuntimeError is raised.

Tung-I Chen 33609886; Pranav 33982032

POST Requests:

- **/order?stock=<stock_name>&quantity=<quantity>&type=<type>** - This URL is used to place a new order. The do_POST() method extracts the stock name, quantity, and order type from the URL query parameters and forwards the request to the Order server using HTTP POST request with the order details in the request body. If the response status code is 200, the response is sent to the client. If the URL is invalid, a RuntimeError is raised.

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, streamhandler, config)

- host_port_tuple: A tuple of (host, port) to which the server should bind.
- streamhandler: A request handler class to handle incoming requests.
- config: A dictionary object containing configuration parameters for the server.

server_bind(self)

- Overrides the server_bind function to enable socket reuse and bind to the server address.

Main Function

The main function initializes the server, reads configuration parameters from a YAML file or environment variables, selects the leader order service, starts a daemon thread to send health check messages to the leader, and serves requests.

main(args)

args: Command-line arguments passed to the script.

The following command-line arguments are supported:

- --config: Path to a YAML file containing configuration parameters for the server.
- --dump: Boolean flag indicating whether to dump cache items into a log file.

If --config is not provided, configuration parameters are read from environment variables. The following environment variables are required:

- CATALOG_HOST: Hostname of the catalog service.
- CATALOG_PORT: Port number of the catalog service.
- ORDER_HOST1, ORDER_HOST2, ORDER_HOST3: Hostnames of the order services.
- ORDER_PORT1, ORDER_PORT2, ORDER_PORT3: Port numbers of the order services.

- ORDER_LEADER_BROADCAST_PORT1, ORDER_LEADER_BROADCAST_PORT2, ORDER_LEADER_BROADCAST_PORT3: Port numbers for broadcasting leader selection messages to the order services.
- ORDER_HEALTH_CHECK_PORT1, ORDER_HEALTH_CHECK_PORT2, ORDER_HEALTH_CHECK_PORT3: Port numbers for sending health check messages to the order services.
- FRONTEND_HOST: Hostname of the frontend service.
- FRONTEND_PORT: Port number of the frontend service.
- OUTPUT_DIR: Directory where cache log files should be stored.
- HEALTH_CHECK_INTERVAL: Interval (in seconds) between health check messages.
- CACHE: Boolean flag indicating whether caching should be enabled.
- CACHE_SIZE: Maximum number of items to cache.
- CACHE_LOG_PATH: Path to the cache log file.
- DUMP: Boolean flag indicating whether to dump cache items into a log file.

The function initializes the cache log file if caching and dumping are enabled. It then creates a ThreadedHTTPServer object with the given host and port, a StockRequestHandler object as the request handler, and the configuration parameters. It selects the leader order service, starts a daemon thread to send health check messages to the leader, and serves requests.

Tung-I Chen 33609886; Pranav 33982032

Design Document - order.py

The order service exposes the following APIs to allow the front-end to query existing orders and allow the replicas to synchronize with each other:

* `GET /orders/<order_number>`
* `POST /orders`
* `GET /synchronize`
* `POST /propagate`

Design choices & API Documentation:

lookup: This function takes two arguments, a file path and an order number, and searches for the corresponding order record in the CSV file located at the specified file path. If the order record is found, the function returns the row as a list of strings. If the order number does not exist in the file, the function returns -1.

- The function uses the csv module to read the CSV file and iterate over its rows. The newline='' argument is used to ensure that the file is read correctly on all platforms.
- The function returns -1 instead of raising an exception or returning None to indicate that the order number was not found. This choice allows the function to be used in a wider range of contexts, but it also means that the caller must check the return value for -1 explicitly.

leader_broadcast: This function listens on a specified port for messages from a frontend that indicate changes in the current leader of a distributed system. When a message is received, the function updates the current leader ID on the server object accordingly.

- The function is designed to run as a daemon thread in the background, meaning that it does not block the main thread of execution. This choice allows the function to run continuously without interfering with other tasks.
- The function uses a TCP socket to receive messages. The socket.AF_INET argument specifies that the socket should use IPv4 addressing, and the socket.SOCK_STREAM argument specifies that the socket should use a reliable, stream-oriented protocol.
- The function uses a while loop to listen for messages continuously. This choice ensures that the function can handle multiple messages without exiting prematurely.
- The function defines three message types: "Ping", "You win", and an order ID. When a "Ping" message is received, the function sends an "OK" message back to the frontend. When a "You win" message is received, the function updates the current leader ID on the server object to the ID of the current server. When an order ID message is received, the function updates the current leader ID on the server object to the ID of the server that sent the message.

Tung-I Chen 33609886; Pranav 33982032

health_check: This function listens on a specified port for regular health check messages from a frontend. When a message is received, the function sends an "OK" message back to the frontend.

- The function is designed to run as a daemon thread in the background, meaning that it does not block the main thread of execution. This choice allows the function to run continuously without interfering with other tasks.
- The function uses a TCP socket to receive messages. The socket.AF_INET argument specifies that the socket should use IPv4 addressing, and the socket.SOCK_STREAM argument specifies that the socket should use a reliable, stream-oriented protocol.
- The function uses a while loop to listen for messages continuously. This choice ensures that the function can handle multiple messages without exiting prematurely.
- The function defines a single message type: "Ping". When a "Ping" message is received, the function sends an "OK" message back to the frontend.

The code also defines a request handler class called OrderRequestHandler that inherits from BaseHTTPRequestHandler in the http.server module. It contains a do_GET method that handles HTTP GET requests to the server.

do_GET method:

- It first checks if the URL path starts with /order?order_number. If so, it extracts the order_number query parameter from the URL and uses the lookup function to look up the order in a CSV file. If the order is not found, it returns a 404 error response with a JSON error message. If the order is found, it returns a 200 OK response with a JSON data message containing the order information.
- The **do_GET** method also handles requests to the /synchronize URL path, which is used to synchronize the replica after a crash. In this case, it simply returns the transaction number as a JSON string.
- If the URL path does not match either of the above cases, it returns a 400 Bad Request error response with a JSON error message.

Design choices in this code include the use of the http.server module for handling HTTP requests, the json module for encoding and decoding JSON data, and the lookup function for looking up orders in a CSV file. The code also uses HTTP response codes to indicate the success or failure of requests, and returns JSON data in a standardized format for easier parsing by client applications.

do_POST method:
This function handles the **POST** request. If the path starts with "**/propagate**", it checks if this is the first propagation after coming back from a crash. If it is, it sends a request to each peer asking for the order data that does not exist in the local database. If the local database has outdated information, it synchronizes it with the peers. Afterward, it saves the updated information in the local database and sends a response to the leader. If this is not the first

propagation, it saves the new information in the local database, sorts the list of order records by transaction number, writes the file, and sends a response to the leader.

If the path starts with "/order," the do_POST function handles a GET request. It checks the validity of the URL and extracts the values of the parameters from the request_body. Then it checks if the quantity is valid and the order type is valid. If everything is valid, it saves the new order in the local database and sends a response to the client. If there is any invalid input, it sends an error message to the client.

Overall, the code design follows a simple client-server architecture, where clients send requests to the server, and the server handles those requests. The server handles requests in a non-blocking fashion using a thread pool. The server saves all the order records in a local database, which is updated when new orders arrive, or when synchronization is needed with other peers. The code is built using Python's built-in libraries such as socket, threading, and csv.

- One of the design choices is to use a thread pool for handling incoming requests. It is a scalable approach, and it can handle multiple requests simultaneously.
- Another design choice is to use a simple CSV file as a database to store order records.

The code is for handling of orders, which are expected to have a URL path starting with '/order'. It first extracts the order type, stock name, and quantity from the query parameters. If the order type is 'buy', it checks if there is enough stock available in the server's local inventory. If there is enough, it deducts the quantity from the inventory and sends a successful response with a JSON payload containing the transaction number. If there is not enough stock available, it sends an error response with a JSON payload containing an error message.

- If the order type is 'sell', the code forwards the order to a catalog service using an HTTP **POST** request. If the order is successful, it sends a successful response with a JSON payload containing the transaction number. It also updates the server's order log and propagates the order information to the follower nodes.

- If the order type is neither 'buy' nor 'sell', the code sends an error response with a JSON payload containing an error message.

The code appears to be following a RESTful architecture, using HTTP methods and URLs to represent resources and actions. It is using JSON as the payload format for responses.

Design choices in this code include the use of the Python http.server module to implement the HTTP server, the use of the requests library to send HTTP requests to other services, and the use of JSON as the payload format for responses. The code also implements a transaction number system to keep track of orders and ensure consistency across the system. The code also includes error handling for exceptions that may occur when sending HTTP requests or updating the order log.

Some components are:

- The self.send_response() function sends an HTTP response code to the client.
- The self.send_header() function sends an HTTP header to the client.
- The self.end_headers() function indicates that the HTTP headers have been fully sent to the client.
- The self.wfile.write() function writes data to the client.
- The json.dumps() function serializes a Python object into a JSON formatted string.
- The requests.post() function sends an HTTP POST request to another web service.

The code also implements a threaded HTTP server that extends the functionality of the ThreadingMixIn and HTTPServer classes. The HTTPServer class is used to create a basic HTTP server, while the ThreadingMixIn class is used to make the server multithreaded, allowing it to handle multiple requests simultaneously.

The ThreadedHTTPServer class overrides the init() and server_bind() functions to add custom metadata and implement socket reuse, respectively. The metadata added includes protocol version, order records, read-write lock, leader node information, paths to local database, catalog information, peer request address, and whether the server is resuming from a crash. These metadata are essential for the server's functionality and enable it to perform various tasks, including reading and writing orders, obtaining and updating the catalog, and communicating with other nodes.

The implementation design choices in this code are geared towards achieving high performance and scalability while maintaining the reliability of the server. The use of multithreading ensures that the server can handle multiple requests concurrently, thus reducing response times and increasing throughput. The use of socket reuse enables the server to reuse existing sockets, reducing the overhead associated with creating new sockets for each incoming request. The metadata added to the server provides the necessary context required for performing different tasks, including handling orders, obtaining and updating the catalog, and communicating with other nodes.

main function:
To start a server that listens to HTTP requests and serves order requests. The server is configured based on command line arguments provided to the script.

Here are some notable design choices in this code:

- The script uses the argparse module to parse command line arguments. This is a good choice for handling command line input in a structured way, and provides a clean interface for users to interact with the script.
- The script checks the validity of the assigned ID, and raises a RuntimeError if the ID is not one of the expected values. This is a good defensive programming technique, as it ensures that the server is started with a valid configuration.

Tung-I Chen 33609886; Pranav 33982032

- The script loads configuration options either from a YAML file or from environment variables. This provides flexibility in configuring the server, and allows the script to run in different environments without modification.
- The script uses a ThreadedHTTPServer class to create a threaded HTTP server that can handle multiple requests concurrently. This is a good choice for a server that needs to handle multiple requests at once, and provides good performance under load.
- The script initializes a local database when the server is started, and resumes from the last order record if the "--resume" flag is provided. This allows the server to recover from crashes and resume serving requests without losing data.
- The script runs two daemon threads: one that listens to leader selection broadcasts from the frontend, and another that listens to regular health checks from the frontend. These threads run in the background and do not block the main thread, allowing the server to handle requests concurrently while also performing other tasks.

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.

Tung-I Chen 33609886; Pranav 33982032

Design Document - catalog.py

The code defines three functions that work with a catalog file containing information about various stocks. 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 find the price of a given stock. The trade() function performs a trade operation on a given stock. The code defines a class CatalogRequestHandler to handle HTTP requests for a stock catalog, and a subclass ThreadedHTTPServer of HTTPServer that uses threading to handle multiple requests concurrently. Also 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 stock 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 trade a specified stock. It parses the request body to get the stock name, quantity, and trade type, and trades the specified stock according to the request. It sends an appropriate response based on the result of the trade 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 output directory, a reader-writer lock, the protocol version, the frontend host and port, and a list of stock names. It also overrides the server_bind method to set a socket option for reusing the address.

In terms of design choices, the use of a CatalogRequestHandler class allows for easy handling of HTTP requests for the stock 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 a trade occurs, preventing stale data from being displayed. The socket option for reusing the address allows the server to quickly restart after a crash or a graceful shutdown.

Design Choices

- The file path to the catalog file is provided as a parameter to all three functions, allowing the code to be flexible in terms of where the file is stored and what it is called.
- The functions use a reader-writer lock (rwlock) to ensure that access to the catalog file is synchronized between threads or processes. The lookup() function only requires read access to the file, while the init_catalog() and trade() 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 file is provided as a dictionary in the init_catalog() function, with the stock name as the key and a dictionary of attributes (name, price, remaining quantity, and accumulated volume) as the value. This allows for easy addition or modification of stock data.

- The lookup() function returns the price of the requested stock if it exists in the file, or -1 if it does not. This allows the calling code to detect when a requested stock is not in the catalog.
- The trade() function returns an integer indicating the result of the trade operation. A value of 1 indicates a successful trade, while negative values indicate various error conditions such as invalid trading volume or insufficient available stock. This allows the calling code to handle different trade outcomes appropriately.

The code defines the main function, which is responsible for running a server. It starts by parsing command-line arguments using the argparse module. If the --config argument is specified, the function loads a YAML configuration file with the specified path, otherwise it creates an empty configuration object and populates it with values from environment variables.

Next, the function creates a ThreadedHTTPServer instance that listens on the specified port and handles requests with the CatalogRequestHandler class. The init_catalog function is then called to initialize the stocks and prices in the catalog, which is stored in a JSON file.

Finally, the function starts serving requests on the specified port using httpd.serve_forever().

Overall, our code makes use of environment variables and configuration files 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. The use of argparse module to parse command-line arguments makes it easier to configure the server in a command-line interface.

API Documentation:

Class:

- CatalogRequestHandler: A class to handle HTTP requests for the stock catalog.

Methods:

do_GET(self): A method that handles GET requests from both the frontend and the order service.

Returns:

- If the request is for stock lookup:
- If the request URL is invalid, return a 400 Bad Request error response with a JSON object containing an error message.
- If the stock is not found, return a 404 Not Found error response with a JSON object containing an error message.
- If the stock is found, return a 200 OK response with a JSON object containing the stock's data.

- If the request is not for stock lookup, return a 400 Bad Request error response with a JSON object containing an error message.

do_POST(self): A method that handles POST requests from the order service.

Returns:
- If the request URL is invalid, return a 400 Bad Request error response with a JSON object containing an error message.
- If the specified stock is not found in the catalog, return a 404 Not Found error response with a plain text message.
- If the trading volume is invalid or excessive, return a 400 Bad Request error response with a plain text message.
- If the trade 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 a 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.
- config: A dictionary containing configuration data for the server.

Overrides the **server_bind()** method of HTTPServer to set a socket option for reusing the address.

def main(args): The main function that runs the server.
Parameters:
- args: A namespace object containing the command-line arguments.

The catalog service maintains the following APIs:
* `GET /stocks/<stock_name>`
* `POST /orders`
* `POST /invalid /<stock_name>`

Calling `POST /orders` and `GET /stocks/<stock_name>` will update and return the data in the catalog database, as they were defined in the lab2.

Tung-I Chen 33609886; Pranav 33982032

Caching
The front-end server maintains a in-memory LRU cache the records the stock information of the last n `GET /stocks/<stock_name>` requests. Upon receiving a stock 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/<stock_name>`, which causes the front-end service to remove the corresponding stock from the cache.

Replication & Consistency
Each order replica will open a socket, and the front-end service will contact each replica via the socket to perform leader selection. The front-end maintains a `leader_selection()` function, which will send a "Ping" message to each reaplica, staring 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 asign it as the leader and nofity the result towards other replicas.

Propagation
When a trade request or an order query request arrives, the front-end service only forwards the request to the leader. In case of a successful trade, the leader node will propagate the information to the follower nodes by sending a `POST /propagate` 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. In our design, this is 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 reqeust 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 in the configuration file. If the leader has no response, the frontend will restart leader selection to find an available order server of the hightest ID.

Resuming mechanism
When a replica comes back online from a crash, it will frist 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 return the latest order number saved in their database. Thus, the resuming replica knows exactly what's the orders it has missed since the previous crash. Afterwards, it will send

Tung-I Chen 33609886; Pranav 33982032

sequential `GET /orders/<order_number>` requests to the node that maintains the latest order information to acquire the missing orders.