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

Lab Overview

This project added caching, replication, and fault tolerance to the stock bazaar application in lab2.
Like in lab 2, the server part consist of **3** kinds of server: **Catalog** server,**Order** server and **Front_end** server.
The front-end server communicates with the client directly,then the back-end servers: catalog server and order servers will proceed the requests depending on what type the request is.

We used **Flask** web framework to help the implementation. Flask is a micro web framework written in Python, which support client requests and send responses from the server to the clients with routing.For instance, with `@app.route()`, we pass a URL into the function `route()`, and define a specific function call related to that URL.

Since the flask framework comes with **multi-threading** support, we just need to set the `**threaded**` parameter to `**True**` when we run the application.

We also have implemented functionality like caching, replication, and fault tolerance in this lab.

Servers

Front_end Server

front_end.py composites with serveral parts.
We use a list as the **cache** to store some stock info for save response time.
A **lock** is used to synchronize access to the cache that guarantees no concurrent access leading to data corruption.
REST API enables the server listens to **HTTP** requests on specify port and URL routes the requests to according functionalities.
Leader Election is done periodically with a leader election algorithm.
The algorithm pings the order servers periodically, where the server is considered dead if it fails to respond the ping. Implement leader election to select a new leader order server upon the time when the current leader fails.To maintain the app work correctly we create an addition thread that executes the ping function concurrently with the main thread of execution.

APIs:
@front_server.route('/rm', methods=['GET']) removes a stock item from the cache.
Request format:

`http://<frontEnd_IP>:30001/rm?stock_name=<stock name>`

Response format:

{“stock_name”:<stock name>,”message”:”removed successfully”}

{“message”:”this stock isn’t exist in cache”}

@front_server.route('/stocks', methods=['GET']) retrieves stock JSON data with the stock name.
The server first checks if the stock data is available in the cache. If it's available in the cache, the server will return the stock JSON object.
Otherwise the server will retrieve the stock data by sending a query request to the catalog server, then caches the data in the cache and return the stock JSON object.

Request format:

`http://<frontEnd_IP>:30001/stocks?stock_name=<stock name>`

Response format:

{“stock_name”:<stock name>,”price”:<price>,
”trade_volume”:<trade volume>,”quantity”:<quantuity>} or
{“error”:{“code”:404,”message”:”Stock not found”}}

@front_server.route('/orders', methods=['POST', 'GET']) will create a new order on the order server if the method is 'POST' and will retrieve the order status for a given order number if the method is 'GET'.

Request format:

POST:

`http://%s:30001/orders`

data(json):{“stock_name”:<stock name>,
”trade_type”:<trade_type>,”quantity”:<quantity>}

GET:

`http://<leaderServer_IP>:30001/orders?order_no=<order number>`

Response:

POST:

Success:
{“order_no”: <order number>}
Failed:
{“order_no”:-1,”message”:<from catalog server>}

GET:

Success:
{“data”:{“order_no”:<order number>,”stock_name”:<stock name>,
”trade_type”:<trade type>,”quantity”:<quantity>}}
Failed:
{“error”:{“code”:404,”message”:”order not found”}}

@front_server.route('/leader',methods=['GET']) will return the current leader order server ID.

Request format:
`http://<frontEnd_IP>:/leader?leader=<leader port>`
Response format:
{“leader”:<leader port>}

Functions:
isExist(stock_name) determine if this stock info exist in cache.

add(item) append new stock info into local database.

ping() invoke /ping API to check if the leader server alive.

leader_election() execute leader election via invoke /ping API to check other nodes’ status and select the node who own largest ID as the new leader.Then invoke /leader API to notify every order server the new leader.

Catalog Server

The **catalog database** is implemented as a list of dictionaries where each dictionary represents a stock. Each stock has the fields in **stock_name**, **price**, **trade_volume**, and **quantity**.
The catalog database is initialized by loading the data from disk named '**catalog.txt**'

Catalog address the concurrency problem by using two **locks**. First lock ‘lock’ is used when querying and updating the stock database. Second lock "disk_lock" is used when reading/writing to the catalog file on the disk.

APIs:
@catalog_server.route('/query', methods=['GET'])
Request format:
`http://<catalogServer_IP>:10001/query?stock_name=<stockName>`
Response format:
Success:
{“data”:{“stock_name”:string,”price”:float,”trade_volume”:int,”quantity”:int}}
Failed:
{“error”:{“code”:404,”message”:”Stock not found”}}

Use the `stock_name` as parameter and returns a JSON object that contains corresponding stock information from the catalog database.If the stock is not found in the database, it returns a JSON object with code 404.

@catalog_server.route('/trade', methods=['POST'])
Request format:
`http://<catalogServer_IP>:10001/trade`
data(json):{“stock_name”:<stock name>,”type”:<trade type>,”quantity”:<quantity>}

Response format:

Success:

{“meaasge”:”order had been trade successfully!”}

Failed:

{“massage”:”Trade failed!”} or

{“message”:”Stock not found!”}

Use stock_name, action choice of buy/sell, and transaction quantity as a request body. Meanwhile, it updates the catalog database accordingly. If the stock cannot be found in the database or the requested transaction cannot be fulfilled due to insufficient quantity or overstock, it returns a 404 error. After the buying or selling, it updates the information in the front-end cache and disk data file.

Functions:

update_cache() is implement to notify the front-end server of any changes made to the catalog database by invoke a `GET` request.

init_database(): Read disk data file `catalog.txt` and store the data into the list type database.

write() used to update the disk data file from list type database.

Order Server

order.py processing orders, syncing orders with leader nodes, and querying orders.

APIs

@order_server.route("/query", methods=['GET']) answers for querying orders from the order database. It takes in order number and return the matched order. If one order is not found, query() returns a JSON object with code 404

Request format:

`http://<orderServer_IP>:<leader_port>/query?order_no=<order number>`

Response format:

Success:

{“data”:{“order_no”:<order number>,”stock_name”:<stock name>,

”trade_type”:<trade type>,”quantity”:<quantity>}}

Failed:

{“error”:{“code”:404,”message”:”order not found”}}

@order_server.route("/trade", methods=['GET']) handles stock trades. It takes in the stock name, trade type, and quantity of shares to be traded as the parameter. It will forward a POST request to the catalog server with the trade info data.

If the catalog server returns a status code of 200, it generates an order number, write the order record into the order database, and notifies other order nodes about the newly created order. It then returns a JSON response containing the order number to front_end server.
If the catalog server returns a status code of 404, it sets the order number to -1 and returns a JSON object with error message and a status code 404.

If the request it sent returns other status code, it returns the error message string containing the status code.

Request format:

'http://<orderServer_IP>:<leader_port>/trade?stock_name=<stock_name>&trade_type=<tarde type>&quantity=<quantity>'

Response format:

Success:

{“order_no”: <order number>}

Failed:

{“order_no”:-1,”message”:<from catalog server>}

@order_server.route('/sync', methods=['POST']) synchronizes order information by receiving data from leader node and updating the order information in the database.

Request format:

`http://<orderServer_IP>:<order_port>/sync`

@order_server.route('/ping', methods=['GET']) is used to ensure that the order server is alive and responses to the requests.

Request format:

`http://<orderServer_IP>:<order_port>/ping`

Response format:

{“alive order server”: <orderServer id>,”port”:<order_port>}

Functions:

sync(data) used to sync the order data as same as leader node via invoke **/sync** API.

init_order() read disk data file and store the data into local order database.

sync_log() sync data via compare the timestamp to select the latest modify disk file as source file.Used while a crashed server restart.

miss_orders(pre_amount) used to find which order be missed while this server shut down.

Client

Client sends HTTP requests to the front-end server for query and trade action. It offers various operation mode.

Client() simulates the primary functionality query and trade. A query request is sent to the server to retrieve information about a random stock among the list of 10 stocks. The response contains the name of the stock and its current quantity.

It uses the `requests` library to send HTTP requests to the front-end server, and it stores the URLs for the stocks and order in the `stocks_url` and `order_url` variables. The `order_type` list contains two string values: "buy" and "sell".

If the quantity is greater than 0 which means the stock is available for trade, client() randomly decides whether to send an order request to the server with a probability `p`. If the decision is to make an order, the function will send an order request to the order server for either buy or sell(which is also chosen randomly) with a random quantity of stocks(quantity number between 1 to 10).

If the order request is successful, for example, the returning message contains an HTTP status code of 200, the order number and the dictionary containing the order information will be appended to the list of successful orders. At the same time, the running time of the query and order requests will be printed out, too.

query_stock() is for testing stock query service. It allows the users input a stock name and how many times they want to query the stock. With the stock name and the number of stocks, the function sends HTTP GET requests to the server to request stock information. It prints the server's response and running time at the end.

query_order() supports the user to query the order with the order number and a specified number of times.

trade() allows the user to make a trade by taking inputs for the stock name, trade options of Buy or Sell, and quantity. It then sends a POST request to the front-end server with all the information. The function also allows the user to make repeating same trade by using the `num` parameter. At the end, it prints the response received from the server and the total running time of the function.

check_consistency() checks the consistency of successful orders made by the `client()` function. If a successful order is made, it send a GET request to the order server to find the order details. If it matches the local order information in the `success_order` list, the function prints "order info match with local!". Otherwise it prints "not match with local".