Design Document

Part 1: Implement Your Multi-Tiered Toy Store as Microservices

Before getting into the design details we want to discuss what are the request types we support and results we get from the frontend server.

We support only ‘GET’ and ‘POST’ methods. For ‘GET’ requests, we need to use the resource ‘/products’ and take the parameter product_name. For ‘POST’ requests we use the resource ‘/orders’ and send a raw json along with the HTTP request.

GET request URL:
http://127.0.0.1/products/product_name=Tux

POST request url:
http://127.0.0.1/orders

We post the request with a raw json inside the body of the HTTP request.

We handled multiple corner cases and returned respective outputs based on the input request to the server. You can check the unit tests in order to know all the different error messages we return to the client.

Design:

Note : We are using socket communication between all the services. The frontend and client communicate on a thread per session model. The frontend, order and catalog services communicate on a thread per request model.

Frontend Service:

This is the server which is getting requests from the client. We are using socket communication in between the client and this service. It creates a threadpool of 10 threads and assigns each thread to the client based on the ‘thread-per-session’ model.

A thread assigned to the client continuously communicates with the client until the client closes its socket. The thread processes the requests and sends back the responses. In this service we parse the HTTP request and analyze if it is a ‘GET’ or a ‘POST’. We get the data from the request and build a dictionary of data that needs to be communicated to either catalog or orders service.

Example of the data that is communicated to orders or catalog service from frontendservice:

Message sent by frontendservice to catalog:

{‘type’ : ‘get’, ‘name’: ‘Tux’}

Message sent by frontendservice to orders:

{‘type’ : post, ‘name’: ‘Tux’, ‘quantity’: 5}

If the HTTP request is a ‘GET’ request, we forward this message directly to the catalog. If the HTTP request is a post request we forward the message to the orders service. We receive responses from either of the services and use it to format a HTTP response which will be sent back to the client.

Orders Service:

We maintain an order log that is maintained by this service. As soon as the service is initiated, we get the latest order number from the log and start appending new successful order details to the log.

This service receives the messages from frontendservice and forwards them to the catalog. The catalog processes the request and replies back with a json object which contains an attribute ‘code’. If the order is successfully placed by the catalog, the code would be 1, else it would be 404 along with the message for the failure. We update the orderlog if we get a success code from the catalog service. We have a lock implemented whenever we are updating the orderlog. Since orderlog is a shared memory that is modified by multiple threads, we are using the lock to safely edit the log.

After we receive the message from the catalog and edit the orderlog, we communicate back to the frontendservice with the order_number. The frontend service processes this response and sends back the reply to the client.

Catalog Service:

This service maintains the database. We have a json file as the database. As soon as this service is initiated, we get the json file into memory. This service receives requests from both frontend and order services. If the request is a get request, we query the database and send back the response to frontendservice. If the request is post, we modify the database and send back the request to order service. This service is also ran on multiple threads where each thread is assigned to process a request. We are also locking other threads when one thread is accessing the shared memory space.

Client Service:

This is an automated client service that does query and buy requests based on a buy probability. We have threads implemented to make this interface run multiple clients simultaneously. This service takes hostname, number of clients and probability as an input from the command line and runs accordingly. We will now discuss how the probability of buy requests are implemented:

In each client we make 100 iterations. In each iteration we do a query request of a randomly chosen item. If the quantity of the item is > 0 then we do the buy of the same item (quantity 0-5) with the buy probability.

So we can run multiple clients simultaneously with a buy probability, by passing the parameters as command line arguments.

Latency Testing:

We have multiple interfaces implemented to get the latency reports of different types of requests.

All these interfaces have classes defined, which can store the latency data in them and can be called by other interfaces by using the class objects.

client.py -> This runs multiple clients simultaneously with each client sending query requests and buy requests with a probability. We defaulted the buy request probability to 0.75 and number of clients to 5.

onlyBuyRequests.py -> This runs multiple clients simultaneously with each client sending only buy requests. This interface takes hostname and number of clients as command line arguments. We defaulted the number of clients to 5.

onlyQueryRequests.py -> This runs multiple clients simultaneously with each client sending only query requests. This interface takes hostname and number of clients as command line arguments. We defaulted the number of clients to 5.

We have a major latency testing interface which runs all the above mentioned interfaces and gets us the graphs required.

aggregateLoadTest.py

This interface runs ‘onlyBuyRequets’ objects by varying the number of clients from 1 - 5. After we get the latencies we plot the graph. Similarly it also runs ‘onlyBuyRequets’ objects by varying the number of clients from 1 - 5. We plot the graphs. After that we also run ‘client’ objects by

varying the number of clients from 1-5 by defaulting the buy probability to 0.75. We can also pass the buy probability through command line arguments while running aggregateLoadTest.py.

Basic Design Diagram: