Design Document - Part 1, 2, 3

Lab 3: Caching, Replication and Fault Tolerance

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

GET request URL:
http://127.0.0.1/orders/order_num=1000

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 (except the restock communication between catalog and frontendservice).

We will discuss in brief what we are doing in caching, replication and fault tolerance. In the later part of the document we will discuss what happens inside each service and the locking mechanisms we used.

Caching:

We have a local cache in frontendservice, which keeps the data data of toy details (quantity available and price). When a get order detail request is successfully processed, we add those details to the cache. Now when the next request comes, instead of sending the request to the catalog, we directly return the information from cache.

There are order requests being placed and continuous restock happening. So whenever there is a successful order placed or restock message from catalog, we remove those toy details from cache. This is because the quantity is changed in the backend.

So whenever we receive a 'get request' for a toy, the frontendservice first checks the cache. If it is present in the cache, it will return the response, else it communicates with the catalog.

Replication:

We have three order services (with the same code). Basically we have three replicas of order services. The frontendservice will choose a leader node and send the order requests to the leader node. The leader node communicates the catalog and processes the request. The leader node updates its log and sends the request to other two order services to update their logs. This way all the order logs are in sync.

Fault tolerance:

If an order service is down, and comes up after some time, it should first sync all its data with the leader node and then start updating each line as the requests are being processed. We will describe what is happening in the service mechanisms.

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' and process the request accordingly.

We have two locks in this service. We will be locking, whenever we are trying to access or write to the shared memory space.

1. A lock to leaderNodePort
2. A lock to cache

As soon as this service starts we first find the order service which is up and which is having the highest id and set that service as a leader node. We also read the cache flag from the terminal, to enable or disable cache in this service.

After the leader election is done, we start listening to the client requests. There are five type of requests the frontend service can encounter.

1. GET the product details request from client - Thread per session model
2. GET order details request from client - Thread per session model
3. POST order request from client - Thread per session model
4. Give me the leaderNode request from order service - Thread per request model
5. Restock details from the Catalog - Thread per session model

Design for handling all the above request types are described below:

When we receive a ‘get product’ request, we will first check the cache. If the toy is present in the cache, we will return the details from the frontendservice itself. If the toy is not present in the cache, we will contact the catalog for the details. After we get a successful response from the catalog, we send the response to the client and we will add those details to the cache.

When we receive a ‘get order_number’ request, we first try to talk to the leader node. If the leader node is up, then we pass the request to the leader node. If the leader node is down, we once again do the leader node election and set the global leader node flag. After that we redirect this request to the newly elected leader node. We receive the response from the leader node and send it to the client.

When we receive a ‘POST order’ request to the frontendservice, it first tries to communicate to the leader node. If the leader node is up, then we continue talking to it, else we do the leader election again and send the request to the new leader. The leader node processes the order request and sends back the response to frontendservice. If the request is successful, we remove that toy from the cache, because the toy quantity is changed at the backend. Then, we communicate the response to the client.

Whenever an order service is up, it wants to know who the leader is, so that it can sync its log with the leader. Hence the order service asks the frontendservice for the leader. This communication is a thread per request model, because it is a single request that is sent during the fault tolerance of an order service. The frontendservice communicates its leader node details to the order service that requested it.

We also receive a restock request from the catalog, whenever there is a restock happening. This is a thread per session model. There is one thread in the fontendservice that always talks to catalog about the restock. The catalog keeps restocking every 10 seconds. If there is a

restock, the catalog sends all the toys that are restocked. Then the frontendservice removes all those toys from the cache.

Orders Service:

We have one lock in this service:

1. This lock maintains the variable : globalOrderNumber and also the order log. We are using a single lock because we will always be touching both the variables in the same critical section. So one lock should be sufficient to handle these two global variables.

Flow of the Orders service:

As soon as we start an order service, it will first read its own log and set the globaleOrderNumber variable to the latest order number.

Then it contacts the frontendservice and asks for the leader node details. After it gets the leader node details from the front end, it tries to contact the leader node, asking for all the rows from the last order in this log. If at all the leader node is down in the meantime (which is rarely possible), we try to connect to another order service for the data (without asking frontend, because the other service would automatically be a leader now).

This order service sends the latest order number to the leader node and receives chunks of 1024 bytes messages as response. It updates its own log and then it starts its own server. After that we close the socket to the leader node. Then the leader node will start processing other requests. If an order service is doing sync, we will block the process of other requests in the leader node. Only after the order service closes the connection, the leader node will release the lock and allow other requests to process.

If the frontendservice is itself down or the other order servies are down, this order service simply starts listening without any syncing.

Now we start the threads and keep accepting the requests from other services. Following are the types of request an order service can receive:

1. If the service is not a leader node, then it get requests to update each line in its log. Then the service simply update its log with that line and set the globalOrderNumber
2. If a service is a leader node, then it can receive a file sync request from another order service which is freshly up. We will send all the lines from that order number to the new order service in chunks of 1024 bytes. The leader node will be locked till all the data is transferred and the new order service starts listening to requests.
3. If the order service is a leader node it gets the order requests from the frontendservice. Then it communicates with the catalog and gets the response. After it gets the response,

it sends the update line request to other two order services if at all they are up. Then it communicates the response to frontendservice which in turn communicates to the client

4. If the order service is a leader node, it gets the ‘get request’ for order_number. Then the service simply queries its log and send the order details to frontendservice, which inturn communicates it with the client.

Catalog Service:

This service uses one lock, which will be used to maintain critical sections that reads and writes the database.

As soon as the catalog service is started, we load the database into our memory. Then we start a restock thread which makes a thread per session connection with the frontend service and keeps restocking the toys every 10 seconds. After restock, it sends the list of all toys it restocked to the frontendservice.

After the restock thread is started, we will start accepting requests in the catalog service. The

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. If the buy is successful, we will do a get request again with the order number and compare if we are getting the right result from the response.

In this way, using this interface, we can run multiple clients simultaneously with a buy probability, by passing the parameters as command line arguments.

Infinite Client:

Infinite client interface can be run on multiple clients that run infinitely, with a specified buy probability. We take the host name, number of clients and probability as command line arguments. By default the host is 127.0.0.1, number of clients is 5 and the buy probability is 0.75. We are using this interface to test replication and fault tolerance.

Latency Testing:

aggregateLoadTest.py

This interface starts a client interface with 5 clients and with buy probability varying from 0.0 to 0.8. We do it in a loop increasing the buy probability every time. At the end of each loop we will wait for 10 seconds so that the restock happens and the buy requests won't be blocked in the next loop.

Then we plot the latency per request plot while we keep varying the buy probability.

Basic Design Diagram: