Dos Project Documentation

Quick Review

Before we dive into the changes we have added, let's do a quick review of what was previously done in this project. You can skip this part it's a just way to remember what our program used to do.

In the first part of our project we have implemented these types of requests, "lookup", "search" and "buy". In our project we had three servers that were located on different machines to create a distributed system. The servers were front-end, order and catalog. The front-end server was responsible for receiving all of the requests from the client, and then redirecting them to their appropriate server. The "lookup" and "search" request were redirected to the catalog server where the database resides. The "buy" request was redirected to the "order" server, but it also sent an "update" request to the database to update the contents of the database.

In this part, we are going to continue our work on this application by implementing some key elements that include replicating, balance loading, consistency and caching.

Replication and load balancing

I am going to talk about replication and load balancing together since they are highly integrated.

Replicating the catalog and order servers is pretty simple, I have just replicated their code written in part one to two new virtual machines. So, the two replicas are entirely the same except in one small aspect, which is the IP address of the host machine, that is the only difference.

Now we need to do load balancing, since replicating the servers without implementing the load balancing mechanism leaves the replicas useless. The load balancing will be request based and it will use the round-robin method.

I have implemented load balancing mechanism using a variable for each server, these variables are called "order_id" and "cat_id". Deciding where the request should go depends on these values.

So, for example when a request wants to go to the catalog server, it first checks the "cat_id" and sends it to the catalog server corresponding with the "cat_id" value.

After sending the request the "cat_id" value is changed, so that when another request wants to reach the catalog server it will go to the other version (replica) of the catalog server.

```
if(cat_id == 0):
    cat_id = 1
    request = 'http://' + Cat_IP1 + '/search/' + topic
  else:
    cat_id = 0
    request = 'http://' + Cat_IP2 + '/search/' + topic
```

Some thought could cross your mind, that this mechanism would fail if multiple requests reach the front-end server at the same time. It's true that probably both requests would go to the same server, but this wouldn't affect the performance of the machine.

To fix this we can implement a lock so that in only one request could enter one at time, but I didn't find the need to do so.

Consistency

From the first part of the project we know that our database is only modified when the client sends a "buy" request, this requests only modifies the quantity of a book. In that case the order server sends an "update" request to the catalog server, which modifies the data base in it. Due to replicating the database on two machines, we now need to modify the two databases whenever an update occurs.

We need to handle consistency from two sides, the sending side which is the catalog server that receives the original "update" request from the order server, and the receiving side which is the catalog server that receives the database update from the other catalog server.

Firstly, the sending side, when this side receives an "update" request it should also send a "modify" request to the other catalog server to keep the data bases consistent, this request contains the id of the book and the new book quantity. It also should sends an "invalidate" request to the front-end server which contains the id of the book, this will be touched upon later in the caching section.

Http requests format:

```
Modify request: 'http://' + Cat_IP2 + '/modify/' + id + '/' + book_quantity
Invalidate request: 'http://' + Front_IP + '/invalidate/' + id
```

Now when the other catalog server receives the "modify" request, it will change its database contents according to the book id and book quantity passed in the "modify" request.

This implementation would work when the system doesn't work in a parallelized system. If we want to make it work, we need to implement mutual exclusion. I didn't implement this in the application because it will degrade the performance so much, but I will mention some headlines of what should be done if we want to implement mutual exclusion.

First of all, you need to choose one of the catalog servers to put the implementation in, it doesn't matter which one. We need to add to add some new routes. In addition to that the server will have a boolean variable called "lock".

When taking the lock, the leader node will check if the "lock" value is true (lock is not in use) or if it is false (lock is in use). If the lock isn't in use it will give it the lock and change the value of "lock" to false. If the lock is taken, it will wait until the lock is released

Releasing the lock is simple you only need to return the value of "lock" to true.

```
lock = True
@catalog.route('/take' ,methods=['POST','GET'])
def take_key():
    global lock
    while not lock:
        pass
    lock = False
    return str("dummy value")
@catalog.route('/release' ,methods=['POST','GET'])
def release_key():
    global lock
    lock = True
    return str("dummy value")
```

Caching

A cache could be implemented in several ways, for this project I have chosen to implement my cache as a dictionary. My choice was based on that dictionaries are fast data structures, and values in them are accessed via a key, which pretty much resembles a cache. In addition to that, operations on dictionaries are simple and the don't require complicated coding. Also, it is easy to limit the size of the cache when using a dictionary.

First of all, I have declared two dictionaries, "cache" and "lru". The "cache" dictionary has the responses of the http request, and the "lru" dictionary has the time since the cache entry wasn't requested, this will help us in implementing a least recently used mechanism to evict entries from the cache. I could have combined these two dictionaries into one, but that would have made the code more complicated and maybe difficult to understand.

The most important part of the cache implementation is the operations that it does, I have implemented some functions that will help us handle the cache probably.

```
cache = {}
lru = {}

add_request(key,value): #adds entry to the cache
delete_request(key): #deletes entry from cache
increment_lru(): #increment the value of entries in the lru dictionary
maximum_key(): #return the key with maximum lru value
update_lru(key): #resets the lru value of a key to zero
cache_full(): #checks if the cache is full
```

The implementations of these function could be seen in the front.py file in the front-end server folder.

Now we have implemented the function we need to know when to use them.

When talking about cache we generally have two cases, a cache hit or a cache miss.

In the case the we encounter a catch hit; all we need to do is return the value of the specified request that is stored in the “cache” dictionary, and update the “lru” value to zero. But in the case of a cache miss, we need to send the request to its proper server and get the response from it. When we receive the response, we need to create a new cache entry with the request and response as key and value respectively.

In the case of adding a new entry to the cache we might face some problems related to the cache size, for that reason we need to check the number of entries in the cache, if that number is the maximum cache size then we need to evict a cache entry. This is done using the least recently used mechanism with the help of the “lru” dictionary. Now that we have a place for the entry we add it to the cache and return the response to the client.

For this project I have sat the maximum number of cache value at 4, since we only have seven books, and increasing the cache size will make the eviction implementation useless.

Note that whenever we receive a “lookup” or “search” request we need to call the “increment_all()” function.

The final case is when we receive an invalidate request from one of the catalog servers, in this case we need to delete the specified request from the cache.

How to run it

Every server that is needed for this project is located on a separate folder on the master branch of the repository, so download each folder to its desired machine. You can also run this application on one machine which I will explain in a moment.

In each folder you will find a python file which is the program that we want to run, before running the program you will need to change some variables. When opening the file, you will notice some variables ending with "_IP", these need to be changed since they are the IP addresses of the virtual machines on my device. Make sure that the IPs are on the same network or it will not work, to do that we need to change the network setting of our virtual machines and make it attached to a "Bridge Adapter".

In case you don't want to run these files on separate machine, you can assign each file to a different port, this is not recommended as it violates the purpose of the project, but it's a good way to test it. To change the port number just add the attribute in red which can be found in the _main_ found in all python files.

```
if __name__ == "__main__":
    app.run(host= IP, port = Port_No)
    app.run(debug=True)
```

The last thing you need to do is install some necessary packages, below are the commands needed to install these packages, just run them on the terminal and you are all done, now you can run all the programs.

The commands needed:

```
$sudo apt install python3-pip
$pip3 install flask flask_marshmallow flask_sqlalchemy
```

Please notice that in the "Front End Server" folder, I have two python files. One where the cache is implemented and on where the cache isn't implemented. Run only on of these files depending on whether you want cache or not.

Now you can run the programs and test how they communicate with each other, you can use some programs to send the http request. For example, you can use "Postman", which is an easy way to send request and display the response.

The results of this application are found in the output file, with the performance testing as it was required.

Also, notice that I have changed the repository from the first part of the project since I wanted to keep them separated.

Part 1 repository: https://github.com/abueideh/Online_Library

Part 2 repository: https://github.com/abueideh/Online_Library_PART2