Part 2 of project:

Turning the Bazar into an Amazon: Replication, Caching and Consistency.

❖ **Group Names & IDs:**

- **Amna Khdair [11819646].**
- **Rawan Hassoun [11819677].**

❖ **Instructor Name:**

- **Dr.Samer Arandi.**

➢ What we do in this part?

In this part we asked to:

- **Make the frontend server doing load balance between two catalog servers and between order servers too:**

✓ **To do this we use "itertools.cycle" method that walk on given list in order and then go back to the beginning, so we give it a list of IP address and port number for each server, as shown below in Fig.1, and we choose request balance:**

```
#List of ip address for catlog servers
IPs=["http://192.168.56.101:6001","http://192.168.56.101:6002"]
iterator_IPs = itertools.cycle(IPs)

#List of ip address for order servers
OrdersIPs=["http://192.168.56.102:6001","http://192.168.56.102:6002"]
iterator_OrdersIPs = itertools.cycle(OrdersIPs)
```

Fig.1: show the list of the catalog and order servers to load balance between them using "itertools.cycle".

Then call "next (list of servers)" method to call to the server whose role it is, when the frontend server want to send request.

- **Make the order server also doing load balance between the catalog servers when it want to query and update certain book:**

✓ **As we describe above, we do the same here, as show below:**

```
IPs=["http://192.168.56.101:6001","http://192.168.56.101:6002"]
iterator_IPs = itertools.cycle(IPs)
```

Fig.2: show the list of the catalog servers to load balance between them using "itertools.cycle".

- **Make replicate for each of catalog and order servers:**

We suppose in this step create a 2 additional virtual machine, one for the replica catalog server and, one for the replica order server, but we don't did this because The size of the laptop of one of us was not enough, and the other had problems with the virtual machine to recognize the network, so to solve this, we used the same catalog virtual for the another replica, but we run on

different ports, as we learned on network course we can run the app or the servers if the port number or IP address or both different, so we copied the catalog server but we defined on different port, then copied the dB file too for the replica instances, and we did the same thing for the order server, as we shown on Fig.3.

Fig.3: show the replica servers on each virtual machine, also show the defined ports for each server for the catalog and for the order servers.

- Create a cache on frontend server, even if its separate or inside the frontend:

✓ To do this, we choose to create the cache inside the frontend server, we defined it as an array, then we stored inside it for each request come to frontend server, we store {what the query, what the result, and how many times ask}.

```
cache.append({"query":"/info/{}".format(idInt),"result":json.loads(result.content),"count":1})
```

Fig.4: show what we store inside the cache.

- Make check the cache before send any request to see if founded on cache or not, if not found we should to store the result after receive the response:

✓ We created method named "checkCache" that called before send any request to see if the query stored inside the cache or not, if not send the request as usual, then when received the result, we stored on the cache, but before send the request we make sure the request is not full(the length of array =5), but if f it full we apply the replacement algorithm that depend on the how many times the query used, all of this shown below:

```
q=checkCache("/info/{}".format(idInt))
result=requests.get(next(iterator_IPs)+"/info/{}".format(idInt), verify=False)
#if cache miss then check if the cache is not full add to cache but if not call repalecment algo
if q=="cache miss":
    if len(cache)==5:
     index=replacmentAlgo()
     cache[index]={"query":"/info/{}".format(idInt),"result":json.loads(result.content),"count":1}
    else : cache.append({"query":"/info/{}".format(idInt),"result":json.loads(result.content),"count":1})
else: q
```

Fig.5: show before send the request we called the "checkCache" method, then check is not full.

```
#this function for check if the query inside the cache or not
def checkCache(query):
    for x in cache:
     if query==x['query']:
         x['count']=x['count']+1
         return x['result']
    return "cache miss"
```

Fig.6: show "checkCache" method body.

```
#this function for doing replacement aldorthim if the cash full
#replace liss query used
def replacmentAlgo():
      min=cache[0]['count']
      index=0
      w=0
      for y in cache:
        if y['count']<min :
            min=y['count']
            index=w
        else: w+=1
      return index
```

Fig.7: show "replacmentAlgo" method body.

- **Check if the request is write, we should to delete the item from cache, then reflect the changes on dB to another one:**

✓ **For this step, we created method that we called "updateCache" that we called for each write request, this method delete the book from cache according to id, and we call the request for each server to reflect the changes, as shown in Fig.8,Fig.9:**

```
idInt=int(id)
amount=request.json['AMOUNTS']
amountInt=int(amount)
print (cache)
q=updateCache(id)
print(q)
print (cache)
result=requests.put(next(iterator_IPs)+"/queryNumbers/"+str(id),data={'AMOUNTS':amountInt})
result=requests.put(next(iterator_IPs)+"/queryNumbers/"+str(id),data={'AMOUNTS':amountInt})
return (result.content)
```

Fig.8: show when call "updateCache" method, and when send the request for both servers.

```
def updateCache(id):
    removed=False
    for x in cache:
     str=x['query']
     strid=str.split("/")
     for s in strid:
       if id==s:
         cache.remove(x)
         removed=True
    if removed==True:
        return "removed!"
    else: return "cache miss"
```

Fig.9: show the "updateCache" method body.

➢ The Design of our project:

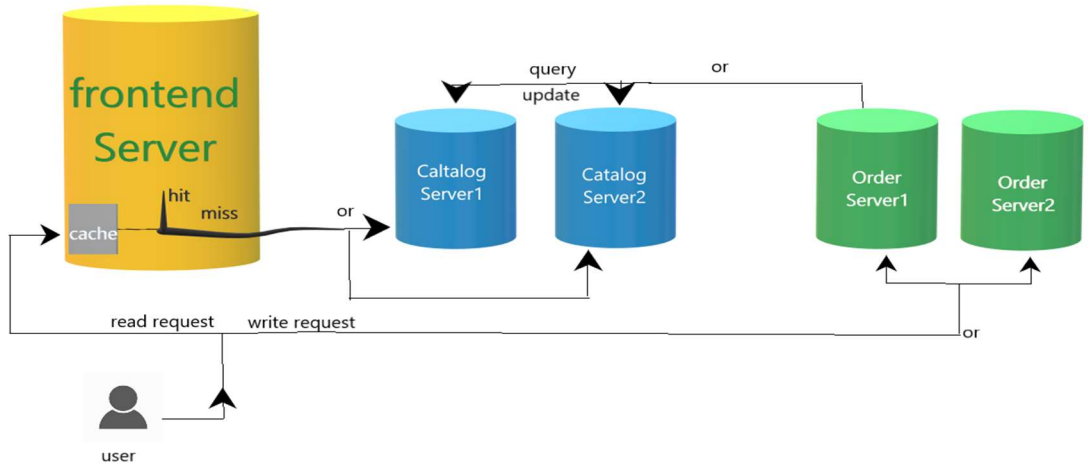

Fig.10: The design of bazar project.

➢ The Demo for the outputs and how to run the program:

The video link:

Copy the link to open the video:

https://drive.google.com/file/d/1s_t6WlvyfVKH2AnL6eEHsmg9jL4qgXS3/view?usp=sharing

Where the first three minutes show how to run all the servers (front end server at local host, tow catalog server at one virtual machine, tow order server at second virtual machine)

Then start to apply all operation:

- **Reading request:**

  1- **Info operation: second 3:00 – second 4:35**
  2- **Search operation: second 4:38 - second 5:51**
  3- **Info operation again to show that the information will get from cache not from the database of catalog server: Second 4:53 - second 7:19**
  4- **show replacement algorithm that happen when cache is full and we make a request that not exist on the cache, as a result we replace the query with least request query and if there more than one query with the same number of use then choose the first entering query to the cache: second 7:20- second 9:15**

- **writing request:**

  1- **Update cost: second 9:25 – second 11:20**
  2- **Purchase operation which make load balance at order server): second 11:20- second 15:18**

- **Note: at the last two minutes at 14:50 must load balance happen but it's not, although the result is true because every server when is called for the first time is started at the given begin port (6000)**

➢ possible improvements and extensions to the program:

The main objective of this part of the project is to relieve some of the pressure on the servers, since some buyers were complaining about the time it took the system to process their requests, the solution was to increase the number of servers locally by replication to reduce the pressure on the single server, to improve this solution in our opinion is that this problem can be solved without a local server, but by moving it to the cloud and leveraging its characteristics, such as Amazon servers, thus increasing the flexibility of the server so that it is only used in proportion to the number of requests from buyers. This reduces the pressure on the server, especially during black friday and other events that run into it. It also reduces the financial cost, in addition to the fact that Amazon's spread around the world allows it to process buyers' requests from far from the server easily.

➢ The Design for improved project:

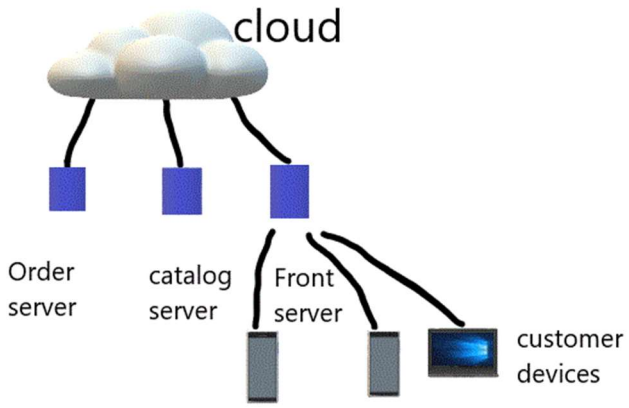

Fig.11: The design of improved bazar project.

➢ Answering the questions:

Info(query)	With cache:49milisecond	Without cache:0.9milisecond
Search(query)	With cache:60milisecond	Without cache:0.9milisecond
Purchase(buy)	With cachehit:113milisecond	With cachemiss72milisecond

Table.1": This table show the average and answer all the questions.

The screenshot for the result above:

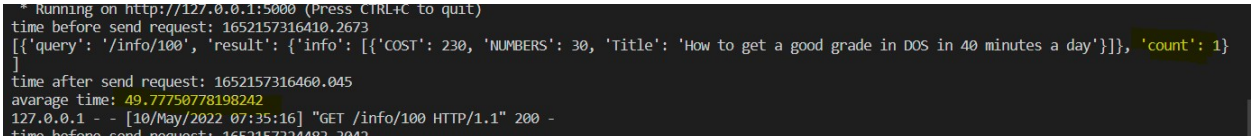

Fig.12: the average time without cashed for info query.

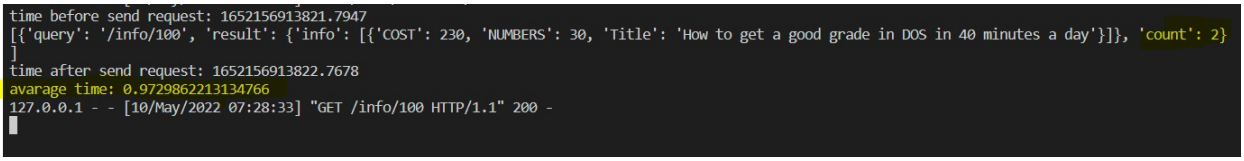

Fig.13: the average time with cache for info query.

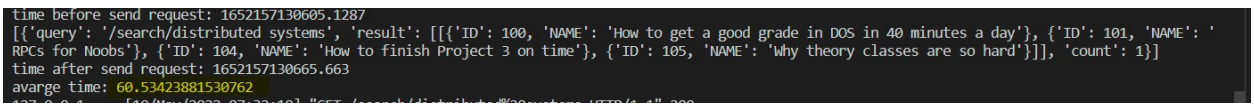

Fig.14: the average time with cache for search query.

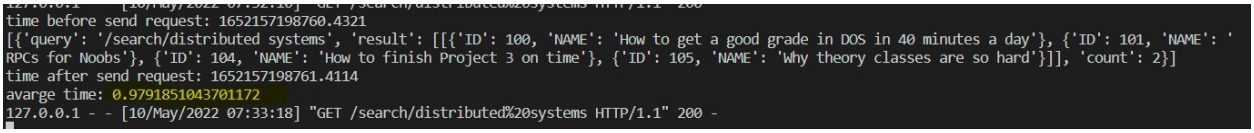

Fig.15: the average time with cache for search query.

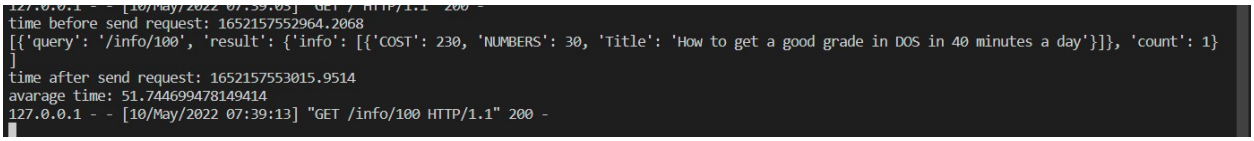

Fig.16: when we send the info query and store it on cache.

Fig.16: average time when send purchase request, and the same book id store in cache, so it send invalid msg.

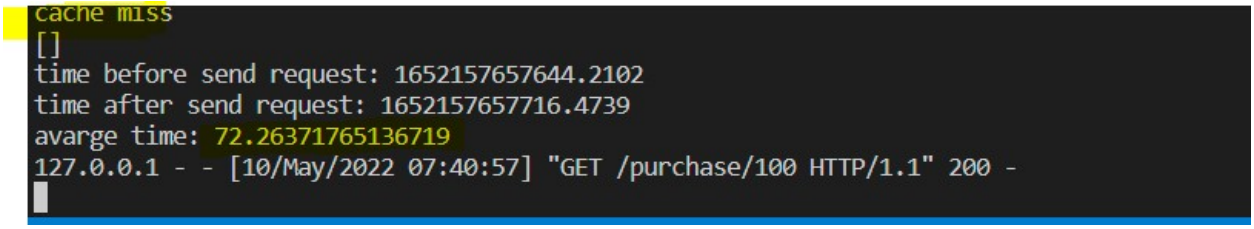

Fig.17: average time when send purchase request, book it cache miss.

- **From above the result, we see how the cache help and saves time and also how the invalid msg overhead the time**