CS677 Lab 2

Bharath Narasimhan, Ronak Zala

Pygmy.com: A Multi-tier Online Book Store

March 26, 2019

1 Readme

1.1 Environment Setup

There is one config file *sv_info.csv* that has server information in the comma-separated format *Type of Server, IP Address, Port.* Modify the config files as required to setup the environment.
There are four Python files *catalog.py*, *order.py*, *frontend.py* and *client.py* - they represent the catalog server, order server, frontend server and the client respectively.

1.2 Program Execution

Start the servers from their respective machines (as specified in *sv_info.csv* using python3 ¡server_name_here.py. Run *client.py* from the fourth machine to start operations.

2 Program Design

Framework used - Flask[1]is a micro web framework written in Python. The Flask version used is 1.0.2

2.1 File Outline

2.1.1 Catalog Server

The file catalog.py represents the implementation of the catalog server in question. Book details are stored in a persistent manner in *catalog.json*. It has the following methods:

1. *get_books* (Query) [GET]: Method to return the results of a client-side *search* or *lookup*. If the query parameter parameter passed is *topic* followed by *gs*(Graduate School) or *ds*(Distributed Systems), it returns all entries belonging to the topic category specified. If the query parameter passed is *item* followed by the item number (Refer Table 2.1.1, it returns details such as number of items in stock, cost.

[1] http://flask.pocoo.org/

Item ID	Book Name
1	How to get a good grade in 677 in 20 minutes a day
2	RPCs for Dummies
3	Xen and the Art of Surviving Graduate School
4	Cooking for the Impatient Graduate Student

Table 1: Book names and their corresponding IDs

2. *update_books* (Update) [POST]: Method to update the stock or cost of specified item. It takes in a query parameter *item* and increments the *stock* of the corresponding book by *delta*. It can also update the *cost* of the item specified. *cost* and *delta* are passed as a JSON with the PUT request. Note that negative delta corresponds to a decrement in stock (happens with each buy request)

2.1.2 Order Server

The file order.py represents the implementation of the order server in question. Transaction details are stored in a persistent manner in $order_log.txt.Ithasthefollowingmethods:$

buy_order [GET]: Method to return the results of a client-side buy request. If the query parameter passed is item followed by the item number (Refer Table 2.1.1, it ***queries*** *the catalog server to to check if the item being requested is in stock. If yes, it* ***updates*** *the stock of the given item by a delta of -1. If not, it does nothing and prints an 'Out of Stock' message.*

2.1.3 Frontend Server

The file frontend.py represents the implementation of the frontend server in question. It functions as an abstraction layer between the client and the servers. It has the following methods:

1. *search* [GET]: Method to return the results of a client-side *search* request. The frontend server just forwards the search request as a **query** to the catalog server.

2. *lookup* [GET]: Method to return the results of a client-side *lookup* request. The frontend server just forwards the lookup request as a **query** to the catalog server.

3. *buy* [GET]: Method to return the results of a client-side *buy* request. The frontend server just forwards the buy request to the order server.

2.1.4 Client

The file client.py represents the implementation of the client in question. It has the following methods:

1. *main*: The functionality of the code is tested by randomly calling one of *search*, *lookup* or *buy* every few seconds (default - 5) with a request to the frontend server. Stock is updated by calling *update_stock* every few seconds.

2. *test_response_times*: A utility function to perform *num_req* number of sequential client requests and measure the average response time. Call with *num_req* and *mode* (*search, lookup or buy)* to get per-tier response times written to *times* directory.
3. *update_stock*: A function to periodically increment the stock of a random book by 2 (default). This is done by directly making a POST call to the catalog server.
4. *pp_json*: A utility function to pretty-print a given JSON file.

2.1.5 Time Parser

The file time_parser.py is used to get the average response times by taking the mean of the times written to the files *(server)_(method)_time.txt* in *times* directory. Run *python3 time_parser.py* after running *test_response_times* with *mode = 'search', 'lookup', 'buy'* in *client.py* to see the ARTs (in seconds). All files in the *times* directory must be deleted before running *test_response_times* to ensure proper ARTs are being recorded.

2.2 Design Features

1. Minimal configuration (Only server host and port) required.
2. Concurrency built-in in Flask.
3. Edge cases like wrong product/missing product handled well
4. Clear print messages for readability at client. Debug messages at all 3 servers
5. End-to-end testing as well as Unit testing done.

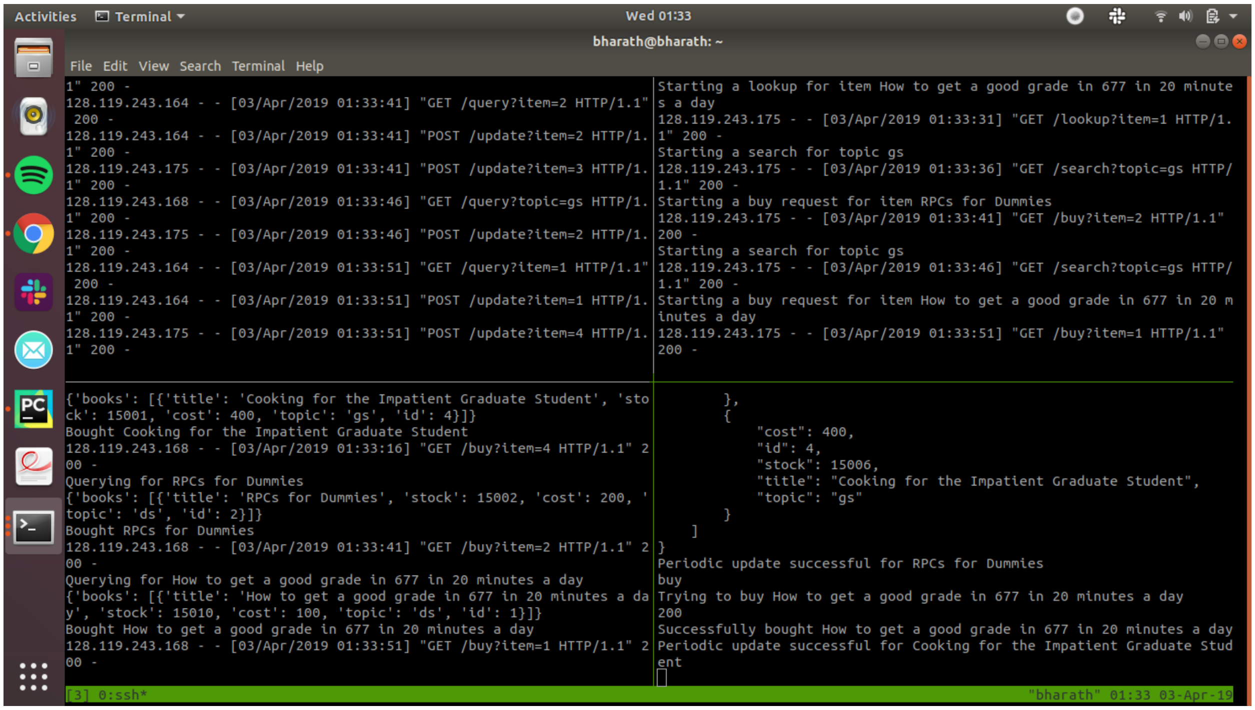
3 Github

The source code can be found at https://github.com/umass-cs677-spring19/lab-2-dosboys. It is divided into 3 folders *docs*, *src*, and *test* for documentation, code and tests respectively.

4 Evaluation and Measurement

4.1 Evaluation on EdLab machines

The catalog, order and frontend servers were setup on *elnux1*, *elnux2* and *elnux3* respectively. (These are the default values and can be changed in *sv_info.csv*). The client was started from *elnux7* and the results are shown in figure **??**



```
1" 200 -
128.119.243.164 - - [03/Apr/2019 01:33:41] "GET /query?item=2 HTTP/1.1"
 200 -
128.119.243.164 - - [03/Apr/2019 01:33:41] "POST /update?item=2 HTTP/1.
1" 200 -
128.119.243.175 - - [03/Apr/2019 01:33:41] "POST /update?item=3 HTTP/1.
1" 200 -
128.119.243.168 - - [03/Apr/2019 01:33:46] "GET /query?topic=gs HTTP/1.
1" 200 -
128.119.243.175 - - [03/Apr/2019 01:33:46] "POST /update?item=2 HTTP/1.
1" 200 -
128.119.243.164 - - [03/Apr/2019 01:33:51] "GET /query?item=1 HTTP/1.1"
 200 -
128.119.243.164 - - [03/Apr/2019 01:33:51] "POST /update?item=1 HTTP/1.
1" 200 -
128.119.243.175 - - [03/Apr/2019 01:33:51] "POST /update?item=4 HTTP/1.
1" 200 -
```

```
Starting a lookup for item How to get a good grade in 677 in 20 minute
s a day
128.119.243.175 - - [03/Apr/2019 01:33:31] "GET /lookup?item=1 HTTP/1.
1" 200 -
Starting a search for topic gs
128.119.243.175 - - [03/Apr/2019 01:33:36] "GET /search?topic=gs HTTP/
1.1" 200 -
Starting a buy request for item RPCs for Dummies
128.119.243.175 - - [03/Apr/2019 01:33:41] "GET /buy?item=2 HTTP/1.1"
200 -
Starting a search for topic gs
128.119.243.175 - - [03/Apr/2019 01:33:46] "GET /search?topic=gs HTTP/
1.1" 200 -
Starting a buy request for item How to get a good grade in 677 in 20 m
inutes a day
128.119.243.175 - - [03/Apr/2019 01:33:51] "GET /buy?item=1 HTTP/1.1"
200 -
```

```
{'books': [{'title': 'Cooking for the Impatient Graduate Student', 'sto
ck': 15001, 'cost': 400, 'topic': 'gs', 'id': 4}]}
Bought Cooking for the Impatient Graduate Student
128.119.243.168 - - [03/Apr/2019 01:33:16] "GET /buy?item=4 HTTP/1.1" 2
00 -
Querying for RPCs for Dummies
{'books': [{'title': 'RPCs for Dummies', 'stock': 15002, 'cost': 200, '
topic': 'ds', 'id': 2}]}
Bought RPCs for Dummies
128.119.243.168 - - [03/Apr/2019 01:33:41] "GET /buy?item=2 HTTP/1.1" 2
00 -
Querying for How to get a good grade in 677 in 20 minutes a day
{'books': [{'title': 'How to get a good grade in 677 in 20 minutes a da
y', 'stock': 15010, 'cost': 100, 'topic': 'ds', 'id': 1}]}
Bought How to get a good grade in 677 in 20 minutes a day
128.119.243.168 - - [03/Apr/2019 01:33:51] "GET /buy?item=1 HTTP/1.1" 2
00 -
```

```
        },
        {
            "cost": 400,
            "id": 4,
            "stock": 15006,
            "title": "Cooking for the Impatient Graduate Student",
            "topic": "gs"
        }
    ]
}
Periodic update successful for RPCs for Dummies
buy
Trying to buy How to get a good grade in 677 in 20 minutes a day
200
Successfully bought How to get a good grade in 677 in 20 minutes a day
Periodic update successful for Cooking for the Impatient Graduate Stud
ent
```

```
[3] 0:ssh*                                          "bharath" 01:33 03-Apr-19
```

Figure 1: Screenshot of evaluations on EdLab

4.2 Average Response time

4.2.1 Average Response time per client request

The results of running 1000 sequential requests to search(), lookup() and buy() are shown in Table 4.2.1 and visualized in Figure 2. We see that the *search* and *lookup* calls take the same amount of time as expected because they are similar GET requests. *buy* takes more time than the two as the order server has to first query the catalog server with a GET request, and then update the catalog server with a POST request.

Method	Response time per client request (ms)
search()	14.245
lookup()	14.316
buy()	29.895

Table 2: Average response time per client request (averaged for 1000 sequential requests each) in milliseconds.

4.2.2 Average Response time per client request with multiple clients

The results of running 1000 sequential requests to search(), lookup() and buy() are shown in Table 4.2.2 and visualized in Figure 3. The number of clients were varied as $n = 1, 2, 5, 10$ and the variation in the response times observed. A general increase in the ART is observed as the number of clients is incresed. This is reasonable because of Flask having to ensure

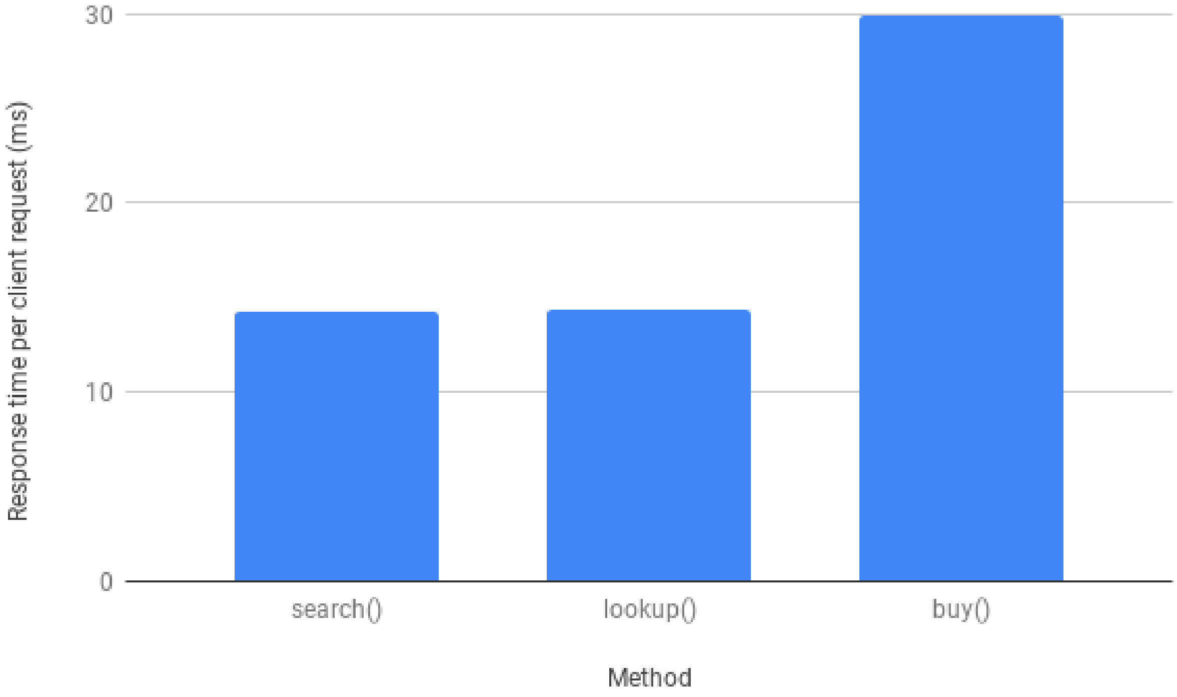


Figure 2: Plot of ART per client request for different methods in milliseconds

consistency of the concurrent requests. The graphs of *search* and *lookup* are similar as is to be expected. The *buy* call has a steeper slope than the former two because of Flask having to block for 2 requests instead of 1.

Method \ART(ms)	$n = 1$	$n = 2$	$n = 5$	$n = 10$
search()	14.245	16.381	18.663	27.908
lookup()	14.316	17.355	18.285	27.201
buy()	29.895	37.711	35.854	60.432

Table 3: Variation of ART in milliseconds with number of clients n for different methods

4.3 Per-tier Response time

The results of running 1000 sequential requests to search(), lookup() and buy() are shown in Table 4.3 and visualized in Figure 4 We observe that time spent in the catalog tier is the least because it is just calculations with no network involved. The time spent increases from Frontend to Client largely due to network latency. The difference between two successive bars would give us twice the latency between the two machines.

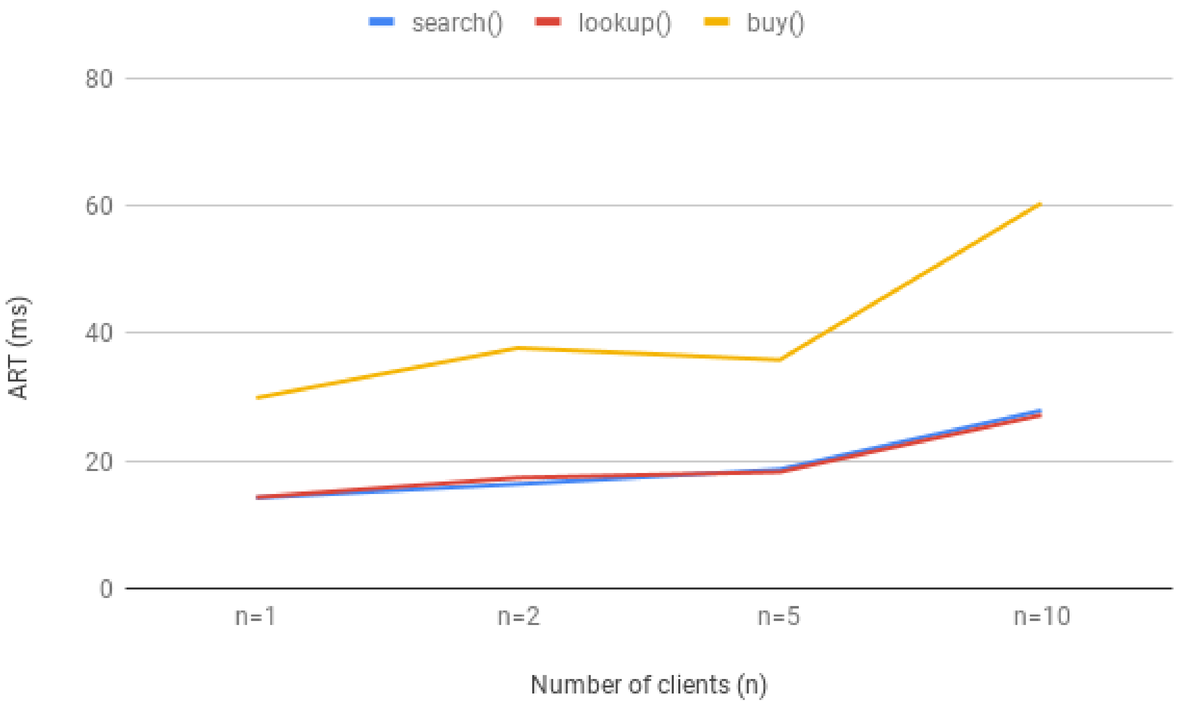


Figure 3: Plot of average response times for each method call versus the number of clients n

Method\Response time (ms)	Catalog tier	Order tier	Frontend tier	Client tier
search()	0.858	-	7.390	14.470
lookup()	0.920	-	7.331	13.856
buy()	0.774	13.823	20.976	27.907

Table 4: Per-tier response time for query and buy requests (averaged for 1000 sequential requests each) in milliseconds.

5 Possible Extensions

It could be possible to use the PUT method for carrying out updates as it is idempotent. This would ensure consistency even if multiple updates are made by mistake.

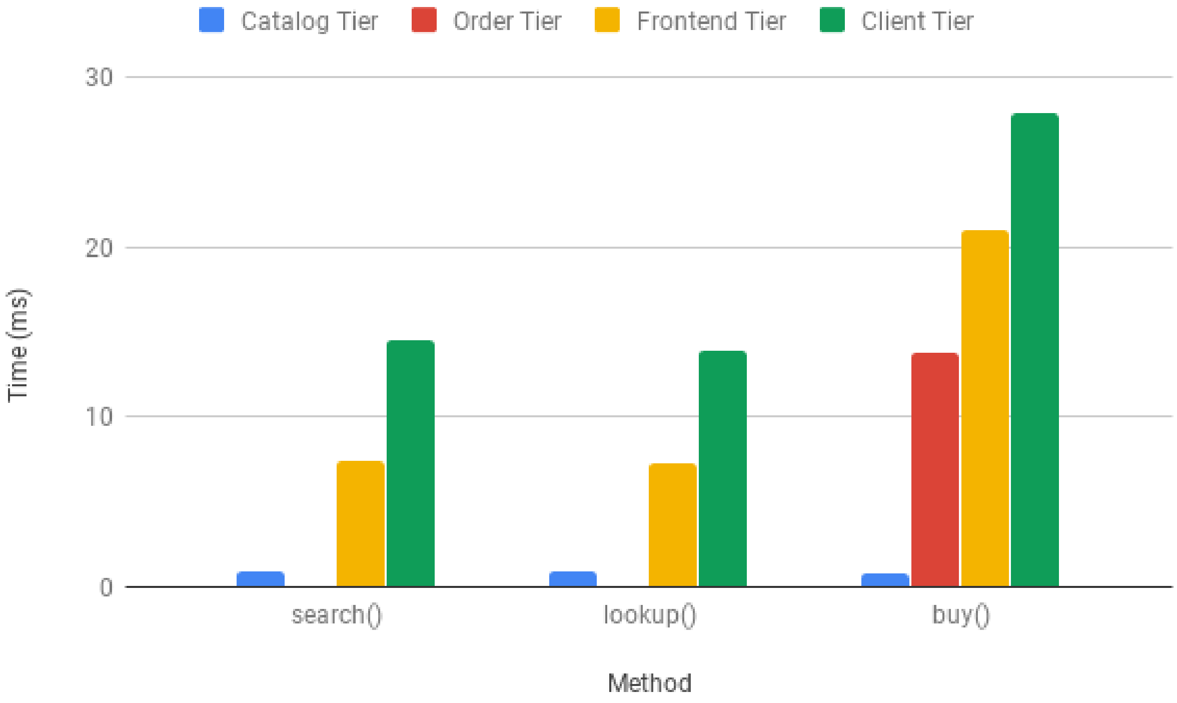


Figure 4: Plot of per-tier response times for each method call