# CS 677 Lab 2 Design Report

**Dan Cline, Kyle Stevens, Tian Xia** *UMass Amherst, Amherst College*

This document provides information of how our multi-tier web system was designed, the trade-offs involved with our implementation and future improvements that may be made on the system.

**Overview of the System**

We used the Python Flask library to implement a web framework with REST APIs. Each server would be initialized with their own `ip` and `port` by reading in the configurations from `config.yml`. Each server is also aware of the addresses of other servers through the same config file. The order server and the catalog server dumps their logs in the local `log` directory. The item configurations are agreed on by all components pre-specified in the catalog server in an internal data structure:

| book | item_number | topic | stock | price |
|---|---|---|---|---|
| "How to get a good grade in 677 in 20 minutes a day" | 1 | systems | 1011 | 3 |
| "RPC for Dummies" | 2 | systems | 1007 | 4 |
| "Xen and the Art of Surviving Graduate School" | 3 | gradschool | 1002 | 15 |
| "Cooking for the Impatient Graduate Student" | 4 | gradschool | 1100 | 5 |

In terms of specific implementation, we followed the component structure and interfaces as given in the lab specification. Below, we describe in details each component and how each component accomplishes its intended functions, as well as additional features.

- `Client`: The client connects to the frontend server and uses the functions exposed by the frontend server to perform `buy`, `search` and `lookup` requests.
- `Frontend Server`: This server has three functions: `search(topic)`, `lookup(item_number)` and `buy(item_number)`. Each function redirects the client to the appropriate server, waits for the results and parse the the `json` request in a nicely formatted string to return to the client.
- `Order Server`: The order server exposes the `buy(item_id)` function which is responsible for querying the catalog server for the current stock number of the desired item. If the stock number is 0, the buy request fails and returns a failure message to the client. Otherwise, the order server sends a updated `json` payload based on the queried json to update the catalog server, which in turn decreases the stock accordingly. An example buy request and json result is as below:

```
/buy/1

{
  "status": true,
  "processing_time": 0.01
}
```

- `Catalog Server`: The catalog server exposes two functions: `query()` and `update`, which has root URLs `/query` and `/update`. The query function allows querying by topic string and item number integers, and returns book details. The update function reads the `json` payload sent by the order server, locks the item database and catalog log, and returns a success message if the update function is performed correctly.Examples of querying and updating request are as below:

```
/query/1
{
  "How to get a good grade in 677 in 20 minutes a day": {
    "item_number": 1,
    "price": 3,
    "stock": 11,
    "topic": 'systems'
  }
}

/query/systems
{
  "How to get a good grade in 677 in 20 minutes a day": {
    "item_number": 1,
    "price": 3,
    "stock": 11,
    "topic": 'systems'
  },
  "RPC for Dummies" : {
    "item_number": 2,
    "price": 4,
    "stock": 5,
    "topic": 'systems'
  }
}

/update/<updated json payload>
{"Success": True}
```

Both the order and catalog server keeps logs. The order server writes each buy request information to the order log. The catalog server writes both the initial stock of the servers and each update requests to the catalog log.

**Design Trade-offs**

For the ease of implementation we hardcoded the address and port of each server on startup. This could be a problem if we want to move the resource of one of the server around, since the other components would lose track of the server address. We also only used one lock to handle updates in the catalog database. Although this is easier to code, it can be inefficient since even buying a different item would require the lock to be released on the item the catalog serve is currently updating.

**Future Improvements**

Because of time constraints, there are several features that we did not implement but would like to have included.

- The current deployment system uses a bash script to `scp` the configuration files and source code to each target server. The process is rather cumbersome and the system will be cleaner and more efficient if in the future we wrapped everything in a docker container for shipment and deployment.
- Another possible extension would be to able to buy multiple items at the same time. This makes more intuitive sense from an online store perspective. The update function in the order server can already handle multiple item buy requests and only the frontend need to be updated to expose the function.

**How to Run**

To run all the designed tests, navigate to the `tests` folder and

```
# for remote testing
$ bash run_all.sh
```

To terminate the process on all machines, send `ctrl+c` to the script and it will terminate all running pids of node instances on each remote machine.

To run the program in general, navigate to the `tests` folder, specify the desired ip address and port number in `config.yml` and run

```
# setting up the server
$ bash run.sh
```

To run the local client to perform functions on the remote servers, run `python3 client.py`

```
# Normal client functions
$ python3 client.py buy <item_id>
$ python3 client.py lookup <item_id>
$ python3 client.py search <topic>

# Sequential requests
$ python3 client.py lookup <item_id> <iterations>

# Write average time to file
$ python3 client.py lookup <item_id> <iterations> <client_id>
```

If the program is run locally output logs for each server will be dumped in the working directory. If the program is run remotely, logs for each server is dumped on the machine it is running on. `ssh` into different machines to view the output logs.