CS 677 Lab 3 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. There are five servers in operation: 2 catalog servers, 2 order servers and 1 frontend server. In terms of replication, we used the primary-write scheme. The frontend uses a random load balancing scheme while we fixed server `order1` to talk with `catalog1` and `order2` talks with `catalog2`. 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
"How to finish Project 3 on time"	5	gradschool	100	1
"Why theory classes are so hard"	6	systems	20	2
"Spring in the Pioneer Valley"	7	gradschool	50	3

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. Caching is implemented by storing the previously queried result in a dictionary, using the item numbers or topic name as key and `Json` as value.

If the catalog server is ever updated, the catalog server sends an invalidation request and deletes the key from the local cache. For load balancing, the frontend server randomly picks a

server from the list of available servers and sends a request. If the request fails to go through, the frontend server just tries to pick another random server and hopes that the requests go through.

Order Server:

There are two replicas of the order server: one primary and one secondary. Each server is assigned an integer ID and the primary is always the alive server with the highest ID. 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 update RESTful request to the catalog server.

When both replicas are running, replica 1 is assigned as the primary. If a server crash occurs, the surviving replica is designated at the primary and as soon as the second replica comes back online, it multicasts to both replicas that the primary is now assigned to replica 1, since now replica 1 must be online.

If the server is a secondary server, it forwards the request to the primary server for the primary to execute locally. Once the request is executed locally, the primary forwards the request to the secondary server, telling the secondary to sync the same update as the primary using the `/sync` endpoint. Both servers write logs of orders into a local database. If the server crashes, it tries to replicate the database of the other server to stay in sync by using the `/download` endpoint. An example buy request and json result is as below:

```
/buy/1

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

Catalog Server:

There are two replicas of the catalog server: one primary and one secondary. Each server is assigned an integer ID and the primary is always the alive server with the highest ID. 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 update API call sent by the order server, locks the item database and catalog log, and returns a success message if the update function is performed correctly.

Replication and fault tolerance is implemented in a similar manner as the order server. The only difference is that when the crashed server wakes up from the crash, it downloads from the alive server both the database and the internal data structure. 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/<item_number>/<attribute>/<operation>/<int:number>
{"status": True}
```

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.

- One 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.
- We could have made the system more expandable by introducing a leader election algorithm that would work with 3 or more replicas. The current system does not need a leader election algorithm since only two servers are involved.

- Another possible extension would be implementing some kind of cache replacement algorithms for the frontend cache like LRU so that for a limited amount of cache memory, we can optimize the cache hit frequency.

How to Run

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

```
# for remote testing
$ bash run_tests.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 without docker
$ bash run.sh

# setting up the server with docker
# bash run_remote.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.