

COMPSCI 677 Spring 2022

Lab 3: Caching, Replication and Fault Tolerance

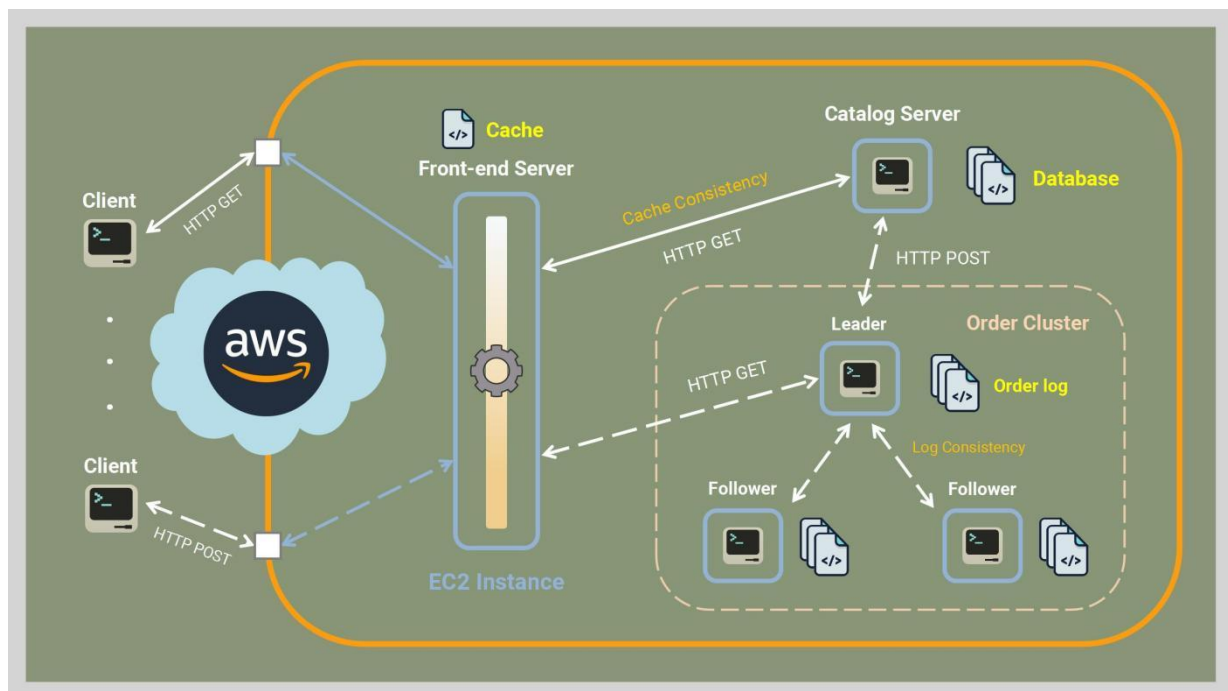
Team Members: Maoqin Zhu, Yixiang Zhang

1. Architecture

In this section, let's focus on the architecture of our online toy store application.

1.1 Overview

This lab assignment is actually based on lab 2. We are further adding caching, replication, and fault tolerance to the micro-service toy store application that we have implemented in the previous labs.



Like in lab 2, the toy store consists of three micro services: the front-end service, the catalog service, and the order service. What's different is that this time we are implementing order service using cluster mode, which can help us realize fault tolerance and high availability. Furthermore, the consistency of cache and log will be taken into consideration as shown above. Finally, this application are being deployed on the well-known cloud platform AWS.

1.2 Flask Web Framework

Recall that we have implemented our own hand-written web framework in lab 2. Since in this lab we will focus on higher level concepts, we are using the modern web framework like Flask to implement our front-end service.

Flask is a micro web framework written in Python. It is classified as a micro framework because it does not require particular tools or libraries.

Once we create the app instance, we can use it to handle incoming web requests and send responses to the client. `@app.route` is a decorator that turns a regular Python function into a Flask view function, which converts the function's return value into an HTTP response to be displayed by an HTTP client, such as a web browser. And we pass the value `'/...'` to `@app.route()` to signify that this function will respond to web requests for the URL `'/...'`, which is the main URL.

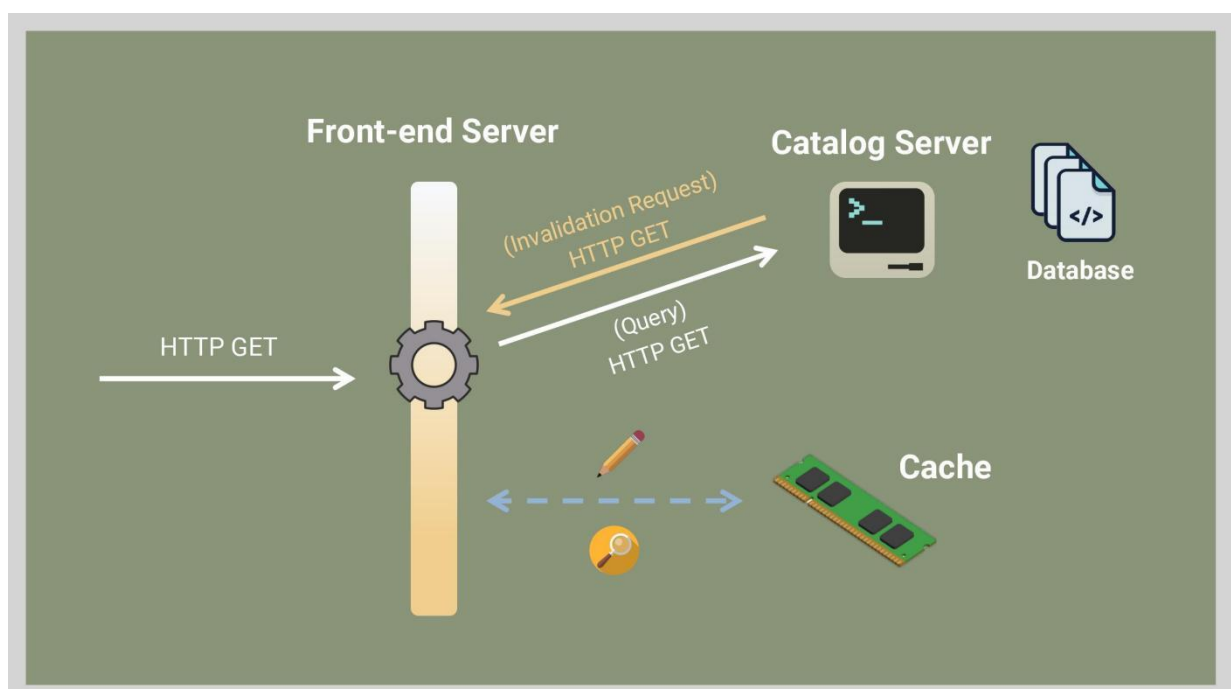
How to Install Flask?

```
$ pip3 install flask
```

2. Caching

In this section, let's see how we add caching to the front-end server.

2.1 Overview



In order to reduce the latency of the toy query requests. The front-end server start with an empty in-memory cache. Upon receiving a toy query request, there are two cases in decorator `@app.route('/products', methods=['GET'])`:

Scenario 1: Cache contains the current querying data

- Read and fetch out the data from in-memory cache, and directly put it into our HTTP GET response.

Scenario 2: Cache does not contain the current querying data

- Forward the request to catalog server.
- Respond the client using the returned result from catalog server.
- Write the valid returned result from catalog server into cache.

2.2 Consistency

In this lab assignment, the catalog server should check remaining quantity of every toy every 10 seconds, if a toy is out of stock the catalog service will restock it to 100. Cache consistency needs to be addressed whenever a toy is purchased or restocked.

Specifically, catalog server sends invalidation requests to the front-end server after each purchase and restock.



front_end.py

The invalidation requests is able to cause the front-end service to remove the corresponding item from the cache. Hence our front-end server provides an API for the cache consistency as follows:

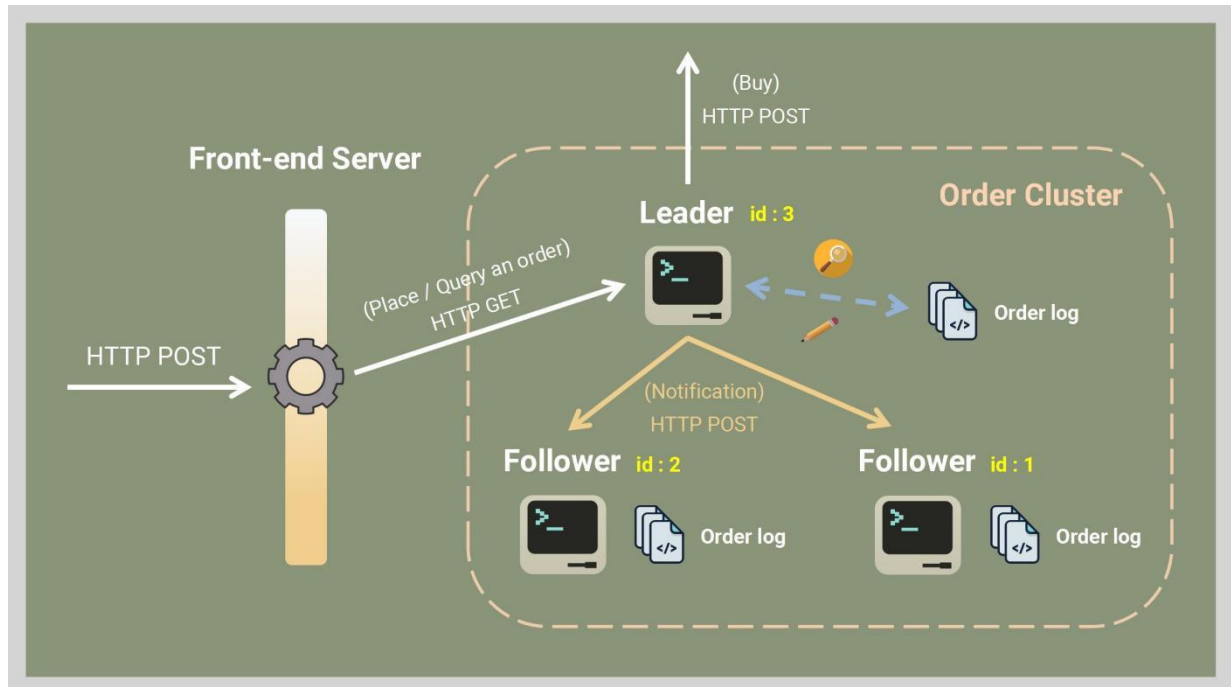
```
@app.route('/rmcache', methods=['GET'])
def rm():
    name = request.args.get('name')
    rmCache(name)

def rmCache(name):
    .....
    ..... # checkout the source code for details
```

3. Replication

In order to make our services more robust, we are replicating the order servers.

3.1 Cluster



Like in previous lab, when we start the application, we first start the catalog service, but what's different in this lab is we start 3 replicas of the order service, each with a **unique id number** and its own order log. In the cluster, there should always exist one node called **leader**, and the rest are called **follower** nodes.

Specifically, when the client sends a buy request or an order query request, the front-end only forwards the request to the leader node by calling following API, instead of all the nodes in the cluster.




order_server.py

```
# query an order API
@app.route('/query', methods=['GET'])
def query():
    ..... # checkout the source code for details

# place an order API
@app.route('/orders', methods=['GET'])
def orders():
    ..... # checkout the source code for details
```

3.2 Consistency

In case of a successful buy (a new order number is generated), the leader node will propagate the information of the new order to other follower nodes to maintain order log consistency.



```
# place an order API
@app.route('/orders', methods=['GET'])
def orders():
    .....
    def notify_others(data)
    .....
    def notify_others(data):
        ..... # checkout the source code for details

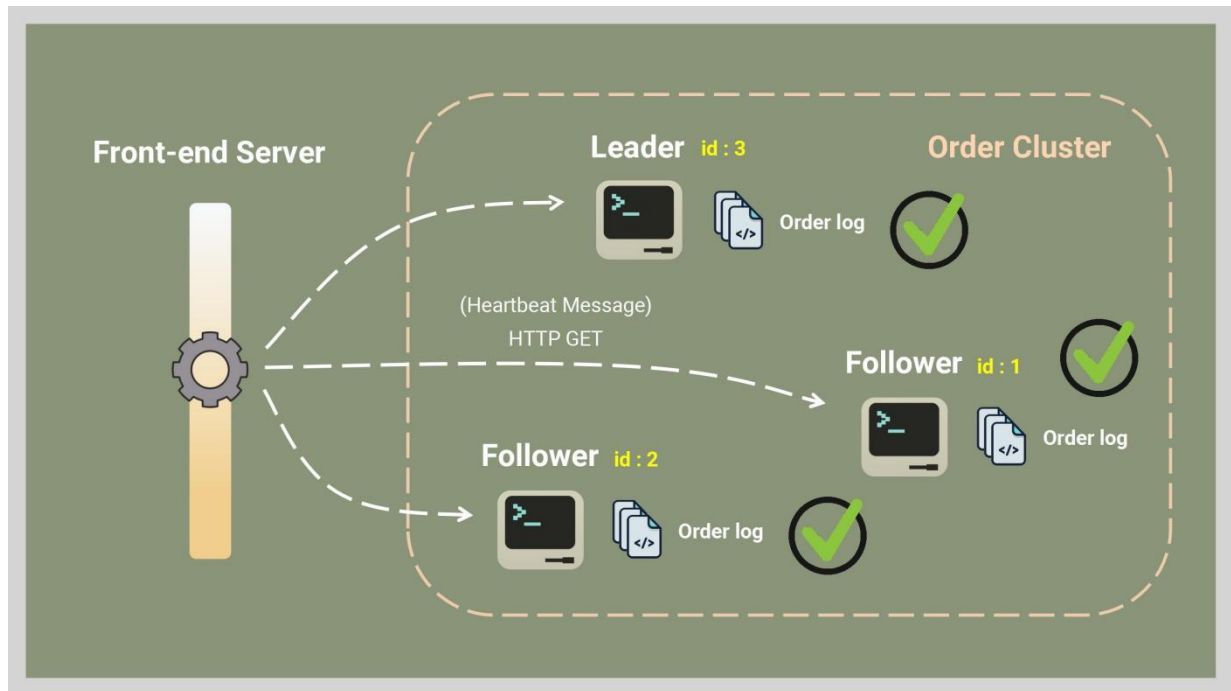
# receive and update log for consistency API
@app.route('/notify', methods=['POST'])
def notify():
    ..... # checkout the source code for details
```

4. Fault Tolerance

In this section, we will make sure that our toy store application does not lose any order information due to crash failures.

4.1 Health Check

The health check of each node in order cluster will be done by front-end server in our design. The notion of heartbeat messages is to track which subset of the replicas are alive at any instant. So the front-end server will send health check request to nodes in order cluster periodically, and tell us the health state of each node on the console. Hence, we can manually restart the crashed node as soon as possible.



```
# heartbeat API
```

```
@app.route('/heartbeat', methods=['GET'])
```

```
def heartbeat():
```

```
    return json.dumps({"idNum": "id"})
```

4.2 Leader Election

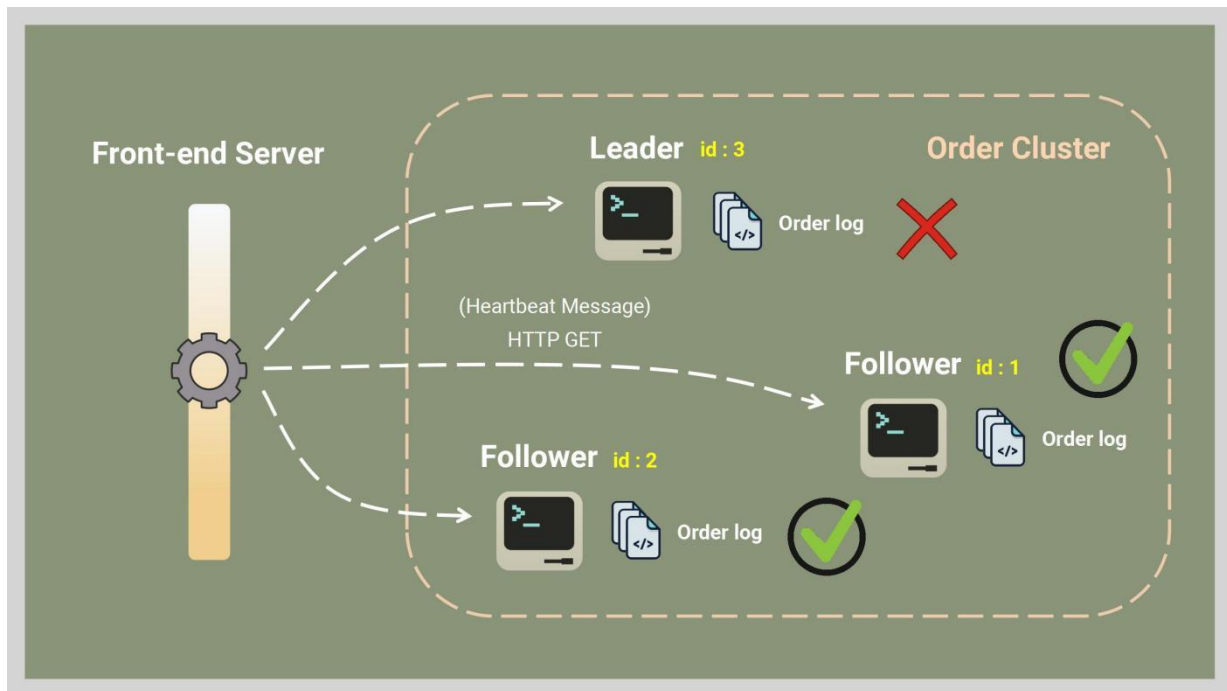
Leader Election Rule:

- The front-end server will always try to pick the node with the highest id number as the leader. Specifically, it will send heartbeat message to the replica with the highest id number to see if it's responsive. If so it will notify all the replicas that a leader has been selected with the id number, otherwise it will try the replica with the second highest id number. The process repeats until a leader has been found.

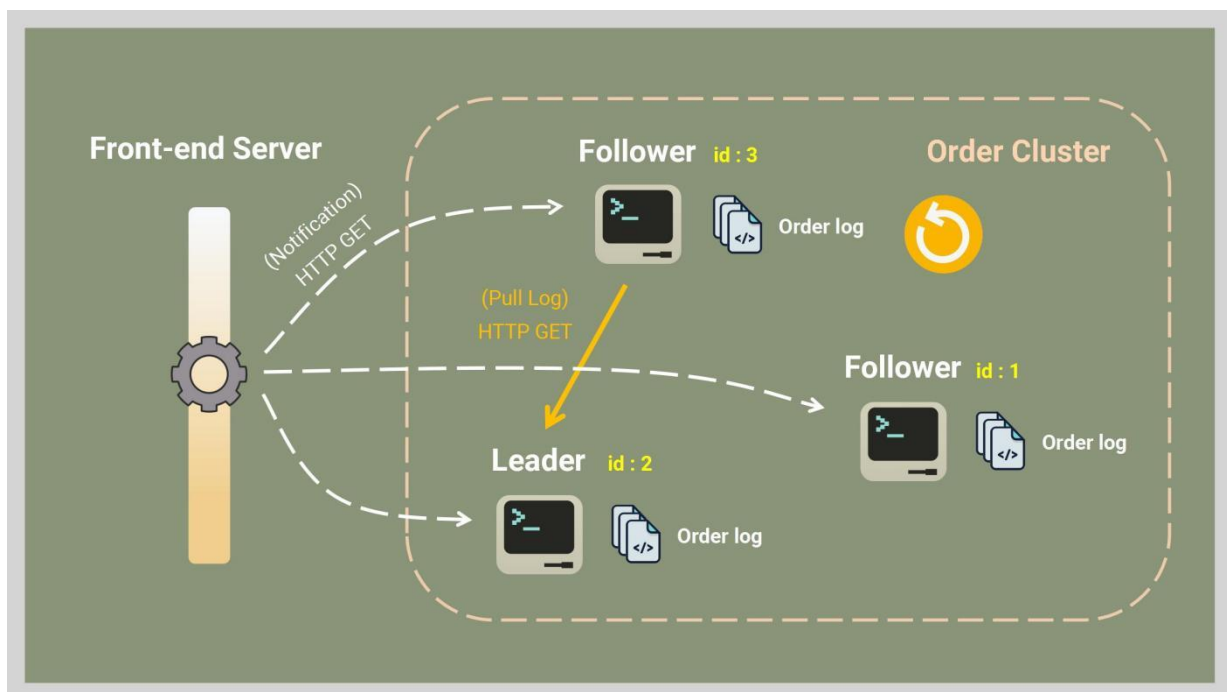
When to Initial an Election?

- The front-end server will send health check request to nodes in order cluster periodically. **Only when** it finds that the leader is unresponsive, it will redo the leader election.


4.3 Crash & Restart



As described in previous section, our goal is that when any replica crashes (including the leader), toy purchase requests and order query requests can still be handled and return the correct result. Therefore, when the front-end server finds that the leader node is unresponsive, it will redo the leader election.



We also want to make sure that when a crashed replica is back online, it can synchronize with the other replicas to retrieve the order information that it has missed during the offline time. So when a server came back online from a crash, it will pull the latest order log from other nodes in the cluster as follows.



```
# pull the latest order log from other nodes

def replica_log():
    ..... # checkout the source code for details

# main function
if __name__ == '__main__':
    .....
    replica_log()
    .....
```

5. Testing & Evaluation with Deployment on AWS

In this section, we will talk about how to deploy our application on the cloud.

5.1 AWS Cloud

Amazon Elastic Compute Cloud (EC2) provides scalable computing capacity in the Amazon Web Services (AWS) Cloud. Using Amazon EC2 eliminates our need to invest in hardware up front, so we can develop and deploy applications faster. We can use Amazon EC2 to launch as many or as few virtual servers as we need, configure security and networking, and manage storage.

Specifically, we are deploying our online application on an **m5a.large instance** in the us-east-1 region on AWS. Actually, we have provided an server **deployment tutorial** in “evaluation” file and “README” file for you. You can checkout the details and follow the steps as described in that tutorial.

5.2 Automated Testing

Like in lab 2, according to each API provided in our source code, we are using **Python unittest module** to perform both the functional test and the load test of our application and micro-services.

In terms of **functional test**, we created a Python testing file called “**test_func.py**”, where our test cases go over all the types of responses by sending a series of valid/invalid requests. In addition, we provide a shell “**test_func.sh**” for you to run multiple testing cases more conveniently. In terms of testing results we get, please check out the “output” document for details.

Execute Shell File to Run Multiple Test Cases



`$ sh test_func.sh` or `$ bash test_func.sh`

In terms of each command in the shell, it follows:

`$ FRONT=<IP Address> CATALOG=<IP Address> ORDER=<IP Address> python3
-m unittest -v <Module>.<Class>.<TestCase>`

Specifically, module name is “test_func”, class name is “TestFunctionality”, and test case is the method name defined in “test_func.py” such as “test_app_client_query_valid”.

Note: Be careful when you do local test. When starting a server, we will print IP address on your screen. Type that IP address as environment variable rather than “127.0.0.1”

Notice that our test cases are effective only when database is in initial state, because expected response is configured statically in testing codes. Of course, you can also run your own test case simply by configuring request parameters and expected responses in the method. The initial state of database should be:

```
1 Tux 25.99 100
2 Whale 34.99 100
3 Elephant 29.99 100
4 Bird 39.99 100
5 Risk 15.99 100
6 Sand 19.99 100
7 Jenga 21.99 100
8 Uno 35.99 100
9 Pinball 49.99 100
10 Clue 9.99 100
```

In order to conduct an automatic performance test, we created a Python testing file called “**test_load.py**”. We are opening multiple client terminals, and each of them will send a series of “Query”, “Buy” or “orderQuery” requests simultaneously using our testing code.

Multiple Terminals



`$ FRONT=<IP Address> python3 -m unittest -v <Module>.<Class>.<TestCase>`

Specifically, module name is “test_load”, class name is “TestLoadPerformance”, and test case is the method name defined in “test_load.py” such as “test_load_query”.

Note: Be careful when you do local test. When starting a server, we will print IP address on your screen. Type that IP address as environment variable rather than “127.0.0.1”.

5.3 Caching, Replication and Fault Tolerance Testing

As described in previous sections, we have added the caching, replication and fault tolerance to our online application. We are providing several test cases for each of them. The details of **testing result** and **evaluation** can be found in “output” file and “evaluation” file.

Caching Test: we are measuring the latency seen by each client for different type requests. Change the probability of a follow up purchase request from 0 to 80%, with an increment of 20%, and record the result for each probability setting. Then do the same experiments but with caching turned off, estimate how much benefits does caching provide by comparing the results.

Cluster Test: we are checking out if the leader election performed correctly, and if followers’ order log is synchronized with the leader’s.

Fault Tolerance Test: we are simulating crash failures by killing a random order service replica while the clients is running, and then bring it back online after some time. Check out if the front end can detect the crash in time, if the whole application breaks down, if the leader election performed correctly, and if the restarted node pull the latest order log successfully.

6. Configurations

6.1 Static Parameters List

1) Initial products info in “database.txt”

- *(toyName, price, quantity)*

2) Ports in “front_end.py”, “catalog_server.py”, “order_server.py”

- *port number*

3) Socket port in “client.py”

- *port number*

4) Client Terminal

- *Type in: which mode, toy name, quantity, number of requests*

(check out Section1.2 for details of each mode)

5) Caching switch in “front_end.py”

- *variable “use_cach_flag”*



6.2 Environment Variables



1) IP address in “test_func.sh”

- IP address of FRONT, CATALOG and ORDER

2) Unittest command at terminal

- IP address, module name, class name, method name

3) Order server startup

- \$ ID=<id> PORT=<port> python3 order_server.py

4) Client terminal

- \$ FRONT=<IP address> p=<probability> python3 client.py

7. Reference

[1] Wikipedia Flask (web framework)

([https://en.wikipedia.org/wiki/Flask_\(web_framework\)](https://en.wikipedia.org/wiki/Flask_(web_framework)))

[2] Community Tutorials

(<https://www.digitalocean.com/community/tutorials/how-to-make-a-web-application-using-flask-in-python-3>)

[3] AWS Userguide

(<https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html>)