# Design Doc

## 1. Introduction

The codes required to run the front-end service, the catalog service, the order service, and the client are in the 'src/front-end', 'src/catalog', 'src/order', and the 'src/client' directory respectively.

As a summary, the connection between the client and the Front-end Service is made using an HTTP-based REST API implemented using the **requests** package and the **http.server** package in Python. The internal connections between each service component on the server-side were implemented using the **gRPC** package in Python. Every service component uses multi-thread to handle RPC requests. Features such as Cache, Leader selection, log propagation, crash failure data recovery are implemented to improve on latency, data consistency, and fault tolerance.

## 2. Front-end Component

### 2.1. Structure

The **ThreadedHTTPServer** server is implemented using the **ThreadingMixIn** class from the **socketserver** package and the **HTTPServer** class from the **http.server** package. The role of **ThreadedHTTPServer** is to wait for new socket connections and initiate a new **RequestHandler** instance in a new thread when a new socket connection is made. The **RequestHandler** class inherits the **BaseHTTPRequestHandler** class from the **http.server** package. A **RequestHandler** instance handles requests by 1) parsing each message, 2) calling the **serve** method of a **NotFlask** instance to get the status code and payload, and 3) sending a response back to the client. While the value of the 'Connection' header is 'keep-alive', the **RequestHandler** instance will be blocked until the next request arrives after handling previous requests with a timeout of 10 seconds.

The **NotFlask** class uses a decorator to simplify codes. A **NotFlask** instance called **app** is initiated at the beginning of the program and it registers the **query** function and the **buy** function that are used to handle the query and buy requests respectively. The **serve** method of the **app** instance is called from **RequestHandler** instances. The role of the **serve** method is to 1) parse the path part of the request, 2) call an appropriate function if the path is valid, and 3) return a status code and payload that will be sent back to the client.

The **CatalogStub** class is used to make a stub instance that sends Query RPC calls to the Catalog Service. The **OrderStub** class is used to make a stub instance that sends Buy RPC calls to the Order Service. Both of these classes are written using gRPC in Python.

### 2.2 REST API error handling

The **RequestHandler** deals with errors with status codes of 400, 404, 414, 431, 500, 501, 505, etc. The query function and the buy function deals with "product not found(404)" errors, "wrong content type(415)" errors, "content-length header required(411)" errors, internal server errors(500),

and many types of "bad request(400)" errors. The **CatalogStub** instance and **OrderStub** instance use a timeout of 3 seconds and an "internal server error(500)" will be sent back to the client if the timeout occurs. When the **buy** request fails due to being out of stock, -1 is returned as the order number to the client with status code 200.

### 2.3. Leader Selection

Upon initiation, the **orderstub_leader_selection** function is used to select a leader order component. The component IDs of three order components and their addresses are given through environment variables. It first sends a **Ping** RPC call with **ping_number** of 0 to check if order components are online in the sequential order using the component ids. The first order component that is found will be selected as the leader component. Then, a **Ping** RPC call is sent with the leader's component id as the **ping_number** to all order components to let them know the leader selection result. However, if all order components are down, this program will exit using the **sys.exit(1)** command. A **Ping** RPC call is sent to the leader component every one second by a separate thread to check if the leader order component is active. This leader selection will be performed again if the RPC call fails because the component is inactive. A lock is used to ensure that at most one leader selection is being held at a time.

### 2.4. Cache

If the required information for a query requested parsed by **ThreadedHTTPServer** is found in **cache**, an in-memory cache initialized empty, then queries with product names are handled without using the **CatalogStub** to send a **Query** call. Whenever new information about products are obtained through the **Query** call, the information is saved in **cache**. A **FrontServicer** instance is used to receive **Invalidate** RPC calls from the catalog component which remove deprecated information from the cache. Using Cache reduced approximately 17% in latency for querying products in when each component is run in separate instances of AWS and 21% when only the front-end service ran in the AWS.

## 3. Order Component

### 3.1. Structure (Leader component)

A **CatalogStub** instance is used to send Order RPC calls to the Catalog service. An **OrderServicer** instance is used to reply to the **Buy** RPC calls made from the Front-end Service. Both the **OrderServicer** class and the **CatalogStub** class are implemented using gRPC in Python. The Order and the Buy RPC calls have 3 seconds of timeout from the caller side.

The **OrderServicer** instance saves the order number, product name, and quantity of successful orders in a log file on the disk. Upon initialization, **OrderServicer** reads the last order number from the log file. If there is no existing log file, a new log file is made and the first order number to use is 0. If the buy was successful, it uses the next order number by increasing one from the previous order number. Otherwise, the negative number received from the Catalog service is returned to the

front-end service as the order number. The order number and the order log file are synchronized using a lock called **order_number_lock** and a lock called **log_file_lock** respectively.

The **OrderServicer** component uses a separate thread called **writer_thread** to write logs in memory to disk. This thread runs the **_write_log_in_file** method where the thread writes logs in the sequential order of the order number to disk periodically with 1 second interval.

### 3.2. Log propagation

For successful purchases, the new log that is created in the leader component is propagated to other order components by making a **Propagate RPC** call using an **OrderStub** instance. In particular, a threadpool of the **OrderServicer** instance is used to send **Propagate** RPC calls concurrently. Received **Propagate** RPC calls are handled through the **OrderServicer** instance of each order component by saving the received log information in memory. The log information in memory is eventually saved into disk as in the leader component.

### 3.3 Crash Failure Data Recovery

When a order component is initiated, it first reads all logs that are saved in disk. Then using a **RecoveryStub** instance, it sends a **BackOnline** RPC call to other order components and receive their next order numbers. If there are gap between the component's order number and the received next order number, the component requests and receives the missing information through the **RequestMissingLogs** RPC call, which is a bidirectional call. The **RequestMissingLogs** RPC call can also be called by **writer_thread** if missing order numbers are found while trying to write logs into disk.

## 4. Catalog Component

### 4.1. Structure

A **CatalogServicer** instance that is implemented using gRPC takes care of the Query RPC call and the Order RPC call through its **Query** method and **Order** method respectively. The catalog data are read from "catalog.csv" into memory during the initialization of a **CatalogServicer** instance.

The **Query** method returns the price and the quantity of the requested product when the product was found in the catalog and returns -1, -1 for price and quantity otherwise. The **Order** call returns -3, -2, -1, and 1 for the order number when the product name was not found in the catalog, when the quantity is not bigger than 0, when the product is out of stock, and when the buy was successful respectively. When the buy was successful, the quantity of the product in the catalog is modified.

A read-write lock is used to synchronize the catalog data in memory. In particular, a separate thread saves the catalog data to the "catalog.csv" file every one second using a read lock if the catalog data has been modified since the last period.

### 4.2. Data management and restock

A separate thread is used to write catalog data into disk periodically with a 1 second interval if the catalog has been modified. Another separate thread is used to restock products with zero quantity to 100 periodically with 10 second intervals. There are 162 types of toys the Catalog Component and are set to 100 by running the 'src/catalog/make_initial_csv.py' file before initiating the catalog component.

### 4.3. Invalidate Cache

The **Invalidate** RPC calls are send to the front-end component using a **FrontStub** instance. Whenever the information about a product due to a successful purchase or restock, a **Invalidate** RPC call is made by using a threadpool.

## 5. Client Component

The **query**, **buy**, and **check** functions are used to query product information, make a purchase order, and query log information respectively. The **run_sessions** function first queries a product's information and if there are at least one quantity of the queried product, it sends a purchase order with probability **p** where **p** is given as an argument. After repeating running the aforementioned session for a pre-defined number of times, if the **check_order** argument is set to True, then it queries log information of all successful purchases made during the sessions and matches the result to the purchase information that the client has. Product name is selected randomly among toys that exist in catalog and the quantity for a purchase request is always 1. For experiments, the **run_session** function can be called in multi-thread.

## 6. To run this program

### 6.1. Catalog Service

- Example in bash

```
cd src/catalog
python3 catalog.py
```

- Environment Variables (catalog.py)
  FRONT_HOST: name or ip address of the front-end component (default: '127.0.0.1')
  FRONT_PORT: port number of the front-end component (default: 1111)
  CATALOG_FILE: path to the catalog file (default: "data/catalog.csv")
  CATALOG_PORT: port number of the catalog component (default: 1130)

- To initialize catalog file in disk

```
cd src/catalog
python3 make\_initial\_csv.py
```

### Order Service

- First component

  ```
  cd src/order
  COMPONENT_ID=1 ORDER_LOG_FILE=data/log1.csv python3 order.py
  ```

- Second component

  ```
  cd src/order
  COMPONENT_ID=2 ORDER_LOG_FILE=data/log2.csv python3 order.py
  ```

- Third component

  ```
  cd src/order
  COMPONENT_ID=3 ORDER_LOG_FILE=data/log3.csv python3 order.py
  ```

- Environment Variables
  COMPONENT_ID: The component ID of the instance. (default: 1)
  ORDER_LOG_FILE: path to the log file (default: "data/log1.csv")

  ORDER_HOST_1: name or ip address of the first order component (default: '127.0.0.1')
  ORDER_PORT_1: port number of the order service of the first order component (default: 1121)
  ORDER_HOST_2: name or ip address of the second order component (default: '127.0.0.1')
  ORDER_PORT_2: port number of the order service of the second order component (default: 1122)
  ORDER_HOST_3: name or ip address of the third order component (default: '127.0.0.1')
  ORDER_PORT_3: port number of the order service of the third order component (default: 1123)

  ORDER2_PORT_1: port number of the recovery service of the first order component (default: 1124)
  ORDER2_PORT_2: port number of the recovery service of the second order component (default: 1125)
  ORDER2_PORT_3: port number of the recovery service of the thrid order component (default: 1126)

  CATALOG_HOST: name or ip address of the catalog component (default: '127.0.0.1')

### Front-end Service

- Example in bash

  ```
  cd src/front-end
  python3 front_end.py
  ```

- Environment Variables
  REST_API_PORT: port number of the restful API of the front-end component (default: 1110)
  FRONT_PORT: port number of the front servicer of the front-end component (default: 1111)

  ORDER_HOST_1: name or ip address of the first order component (default: '127.0.0.1')
  ORDER_PORT_1: port number of the order service of the first order component (default: 1121)
  ORDER_HOST_2: name or ip address of the second order component (default: '127.0.0.1')
  ORDER_PORT_2: port number of the order service of the second order component (default: 1122)
  ORDER_HOST_3: name or ip address of the third order component (default: '127.0.0.1')
  ORDER_PORT_3: port number of the order service of the third order component (default: 1123)

  CATALOG_HOST: name or ip address of the catalog component (default: '127.0.0.1')
  CATALOG_PORT: port number of the catalog component (default: 1130)

### Client Service

- Example in bash

```
cd src/client
python3 client.py
```

- arguments
  –front_host: name or ip address of the front-end component (default: '127.0.0.1')
  –front_port: port number of the restful API of the front-end component (default: 1110)
  –n_repeats: number of sessions to run(default: 10)
  –n_threads: number of clients to run (each client runs in a different thread)
  –p: probability to send purchase requests after query requests if not out-of-stock. (default: 0.5)
  –run_type: session types to run (default: 'session_check', other choices: 'query', 'session')

## 7. References

- Read-Write lock:
  https://pypi.org/project/readerwriterlock/
  https://en.wikipedia.org/wiki/Readers%e2%80%93writers_problem
- Python requests package
  https://docs.python-requests.org/en/latest/
- Python http.server package
  https://docs.python.org/3/library/http.server.html
  https://github.com/python/cpython/blob/3.10/Lib/http/server.py

- Restful API

  https://help.tableau.com/current/api/rest_api/en-us/REST/rest_api_concepts_example_requests.htm

- HTTP Error Status

  https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/HttpStatus.html

- gRPC bidirectional RPC calls

  https://www.techunits.com/topics/tutorials/grpc/how-to-implement-bi-directional-streaming-using-grpc-with-python-server-part-1/