Distributed Transaction Settlement System

Jianbang Yang 515030910223 Wuwei Lin 515030910268
Tianrui Chen 515030910289

Table of Contents

1. Overview of System Environment

We deploy our **containerized(dockerized)** applications on a **Docker Swarm** cluster consisting of three Aliyun ECS instances. Here are some details:

- Every Aliyun ECS instance has 2 vCPU, 4 GiB memory and 40 GiB storage which is enough to meet this lab's hardware requirements.
- All the system components are deployed in docker containers so that their dependent environments are well isolated. The docker images we made are reusable so the system can deployed again easily.
- Docker swarm is a native cluster framework for docker. We can manage our docker cluster easily with docker swarm.

There are quite a few components, such as zookeeper and kafka, which are well organized in our system. Each component may be consisted of one or more docker services. All the service in a same component are organized as a **docker stack**.

Service stacks of our system:

- A Hadoop cluster consisting of one namenode and three datanodes
- One Yarn resource manager and three slaves
- A zookeeper cluster of three nodes
- A replicated Kafka cluster of three nodes
- A replicated HTTP server of three nodes
- One container controlling the exchange rate table
- A three-replicated sharded Mongo cluster
- One container running the Spark streaming application

2. Deployment of System

With the help of docker swarm, we can just simply write a configuration file and deploy a complicated component with a simple command - `` `Docker stack deploy -c filename <stack_name>` ``

2.1 Docker Swarm Cluster

Firstly, we have to install docker and docker swarm since our system is fully dockeried.

Then, we set up docker swarm cluster. To setup a Docker Swarm cluster, run `` `docker swarm init` `` on one node to initialize it as cluster manager, and run `` `docker swarm join` `` on other nodes.

Thirdly, we create a **dedicated** overlay **network** for our system by running `` `docker network create -d overlay --attachable ds-cluster` `` and attach containers in our system to the network such that services are visible to each other. Besides, we create a NFS among three nodes to provide persistence by assigning volumes to containers. This prevents data loss when containers are transferred to different hosts.

2.2 HDFS and Yarn

We still have to deploy a Hadoop cluster because we will run Spark application on Yarn cluster. The Hadoop cluster' deployment is much same as we did in lab 4.

The Hadoop cluster contains a namenode and a resource manager running on two different containers, three slave containers running both data node and node manager processes. The cluster is started up by running `start-dfs.sh` and `start-yarn.sh` scripts from the entrypoint commands from the master.

The relevant file for hadoop cluster deployment is *<u>/docker-swarm/hadoop/docker-compose.yml</u>* of the project. Run `` `Docker stack deploy -c /docker-swarm/hadoop/docker-compose.yml zk` `` to deploy hadoop cluster.

2.3 Spark

Spark can be runned in different modes. One of those modes is Yarn cluster mode. In this mode, the Spark driver runs inside an application master process which is managed by Yarn on the cluster.

In our system, Spark is deployed in **Yarn cluster mode**. There is no need to start a standalone spark cluster since we already have a Yarn cluster. So in the spark image, we just download the corresponding spark binary package, copy all the hadoop configuration files into the image and ensure that the environment variable `HADOOP_CONF_DIR` points to the directory which contains the (client side) configuration files for the Hadoop cluster.

We have already added spark image in hadoop services stack that we deploy before so we don't need to deploy spark alone here.

2.4 Zookeeper

Zookeeper cluster contains **three** nodes running in different containers. To setup the cluster, we add three services in the docker-stack file, assign different hostnames and zookeeper **myids** and configure their network as ds-cluster, the overlay network we created before.

The relevant file for zookeeper deployment is *<u>/docker-swarm/zk.yml</u>* of the project. Run `` `Docker stack deploy -c /docker-swarm/zk.yml zk` `` to deploy zookeeper.

2.5 Kafka

Kafka cluster is setup a service deployed in **global mode** in docker swarm such that it run one replica on each node. So we deploy **three** kafka service in total.

Since Kafka nodes are required to register their hostname to zookeeper, we use hostname of the host instead of that of the containers to make them globally accessible from both hosts and the docker network as otherwise binding to a container's hostname does not work well with the service mesh (see troubles we describe below). Note that the hostname of the host can be obtained from `` `docker info` ``.

To configure Kafka, we need to set zookeeper addresses (`zoo1:2181,zoo2:2181,zoo3:2181`), addresses and ports of listeners and advertised listeners in `$KAFKA_HOME/config/server.properties`. The Kafka server can be started by running `` `$KAFKA_HOME/bin/kafka-server-start.sh $KAFKA_HOME/config/server.properties` ``.

The relevant file for kafka deployment is *<u>/docker-swarm/kafka.yml</u>* of the project. Run `` `Docker stack deploy -c /docker-swarm/kafka.yml kafka` `` to deploy kafka.

2.6 Mongo

We aim to deploy a **distributed** mongo cluster with **high-availability**.

The Mongo cluster contain a three-member config server replica set, a three-member data server replica set, two sharding router and a bootstrap container, each running on different containers. Besides, we set a Mongo Express container that provides WebUI for accessing MongoDB.

The config server is started up by:

`mongod --configsvr --replSet cfgrs --smallfiles --port 27017`

On each replica node, and the data server is by

`mongod --shardsvr --replSet datars --smallfiles --port 27017`.

The sharding router is connected to config servers:

`mongos --configdb cfgrs/cfg1:27017,cfg2:27017,cfg3:27017`.

The bootstrap server adds each replica node to the replica set after they have started.

To connect to the Mongo cluster, we can use the connection string `mongodb://mongos1:27017,mongos2:27017`, which will route requests to sharding servers.

The relevant file for mongo-cluster deployment is *<u>/docker-swarm/mongo-cluster.yml</u>* of the project. Run `Docker stack deploy -c /docker-swarm/mongo-cluster.yml mongo` to deploy mongo cluster.

2.7 Screenshots

Docker swarm cluster:

```
➜ ~ docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS      ENGINE VERSION
uutfk0k8qjvuj8ktt2lx39mis *   node1               Ready               Active              Leader              18.03.1-ce
oiivdy6l6kn763w5h4n4q80rl     node2               Ready               Active                                  18.03.1-ce
gbupgurdnouh3jsqtq16j8t9a     node3               Ready               Active                                  18.03.1-ce
```

Docker stacks and docker services:

```
➜ ~ docker stack ls
NAME                SERVICES
express             1
hadoop              6
kafka               1
mongo               9
rater               1
server              1
zk                  3
➜ ~ docker service ls
ID                  NAME                        MODE                REPLICAS            IMAGE                                                                  PORTS
ofijfxv0r43o        express_mongo-express       replicated          1/1                 mongo-express:latest                                                   *:8081->8081/tcp
l29p1r3zc229        hadoop_master               replicated          1/1                 registry-vpc.cn-hongkong.aliyuncs.com/vinx13/hadoop-namenode:latest    *:50070->50070/tcp, *:
50090->50090/tcp
00rue97e2752        hadoop_resourcemanager      replicated          1/1                 registry-vpc.cn-hongkong.aliyuncs.com/vinx13/resource-manager:latest   *:8032->8032/tcp, *:80
88->8088/tcp, *:8188->8188/tcp, *:10200->10200/tcp, *:19888->19888/tcp
so52xhwz6zuu        hadoop_slave1               replicated          1/1                 registry-vpc.cn-hongkong.aliyuncs.com/vinx13/hadoop-datanode:latest    *:8042->8042/tcp, *:50
075->50075/tcp
mfl4qgo8qabh        hadoop_slave2               replicated          1/1                 registry-vpc.cn-hongkong.aliyuncs.com/vinx13/hadoop-datanode:latest    *:8043->8042/tcp
yg3nfl2rhi00        hadoop_slave3               replicated          1/1                 registry-vpc.cn-hongkong.aliyuncs.com/vinx13/hadoop-datanode:latest    *:8044->8042/tcp
j9xg0a5tse90        hadoop_spark                replicated          1/1                 registry-vpc.cn-hongkong.aliyuncs.com/vinx13/spark:latest
v7yeq5s99r9v        kafka_kafka                 global              3/3                 wurstmeister/kafka:latest
y07z5wjugryt        mongo_bootstrap             replicated          1/1                 stefanprodan/mongo-bootstrap:latest
sd98q12naw80        mongo_cfg1                  replicated          1/1                 mongo:3.4
goebjve0a1r4        mongo_cfg2                  replicated          1/1                 mongo:3.4
n6d1p506iwm2        mongo_cfg3                  replicated          1/1                 mongo:3.4
9z12hg5e59vn        mongo_data1                 replicated          1/1                 mongo:3.4
96xaevgjxnpn        mongo_data2                 replicated          1/1                 mongo:3.4
yaxqv7ksmvph        mongo_data3                 replicated          1/1                 mongo:3.4
tnxswiuku1e6        mongo_mongos1               replicated          1/1                 mongo:3.4
zd474tzts122        mongo_mongos2               replicated          1/1                 mongo:3.4
yqiu4r0mbbmg        rater_rater                 replicated          1/1                 registry-vpc.cn-hongkong.aliyuncs.com/dynamicheart/rater:latest
uo2f0zrowq1g        server_server               replicated          3/3                 registry-vpc.cn-hongkong.aliyuncs.com/vinx13/server:latest             *:20080->20080/tcp
9acbnzimp7gs        zk_zoo1                     replicated          1/1                 zookeeper:latest                                                       *:2181->2181/tcp
bav75yqfkas3        zk_zoo2                     replicated          1/1                 zookeeper:latest                                                       *:2182->2181/tcp
0umc0e49s8a5        zk_zoo3                     replicated          1/1                 zookeeper:latest                                                       *:2183->2181/tcp
```

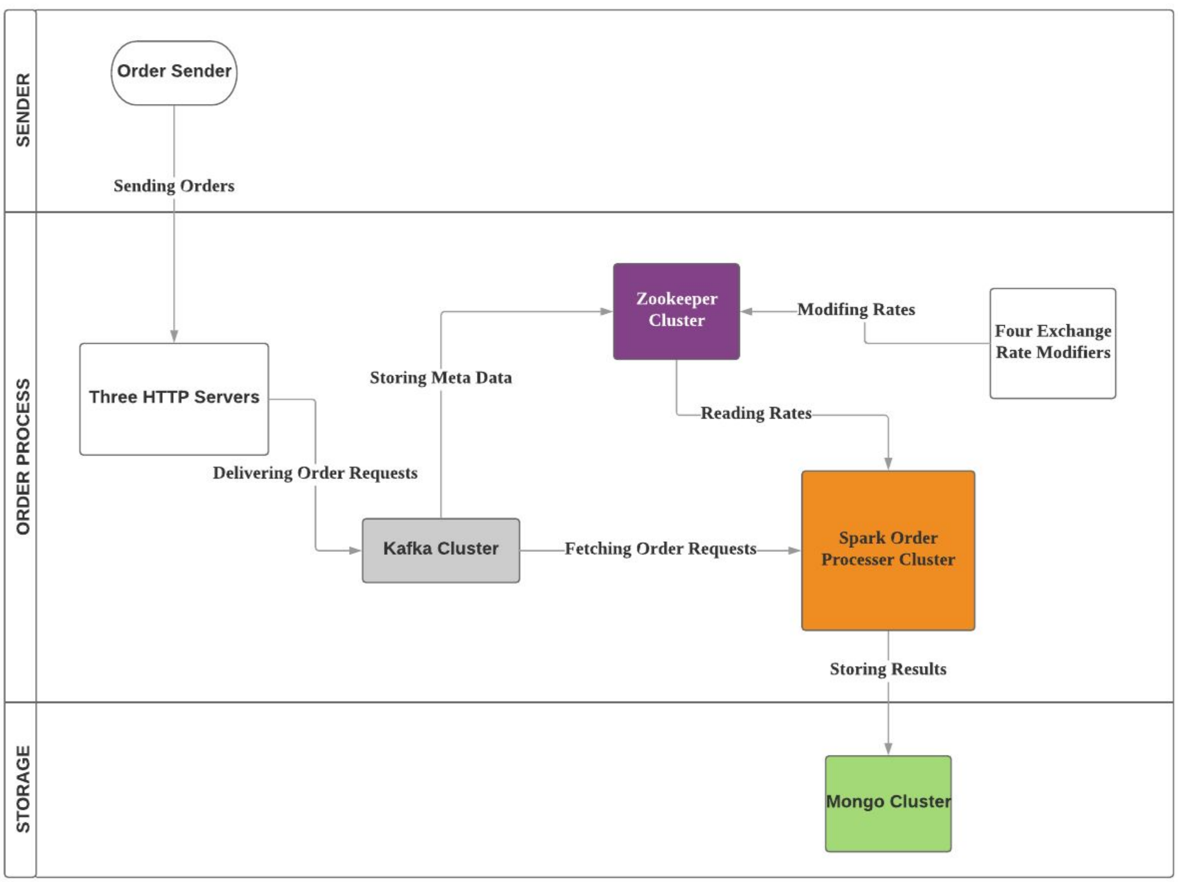
3. Program Design

3.1 Order Settlement Process Description

Here are the whole process:

- The order senders send out all the orders to the HTTP servers at a time.
- Then HTTP server simply delivering order requests to kafka.
- Four exchange rate modifiers contend to modify rate table stored in zookeeper.
- At the same time, spark order processor reads order requests from kafka.
- Once spark read the valid rate table from zookeeper, it do the map-reduce operations and store the result into mongo cluster.

The illustration below shows the whole process:



3.2 HTTP Server

HTTP server is responsible for receiving order requests and delivers these orders to kafka. We implement it in Go. The server is replicated on three containers such that requests to the server is load-balanced by the service mesh of Docker Swarm. The services provides an ingress port, which is accessible from any host nodes and is routed to some containers in round-robin manner.

The servers maintain long connection to three Kafka brokers. On requests, the servers read the request body, which is a JSON object string, and synchronously send the request body as message to the Kafka topic. After receiving responses from Kafka brokers, servers send responses to the client with status code 200 or 500. In this way, we can either ensure that requests are sent to Kafka or explicitly ask clients to retry.

3.3 Requests Sender

To simulate transaction requests, we implement a HTTP requests sender in Python, which reads the requests data saved in a JSON array in a file, sequentially sending each element in the array representing one transaction request via HTTP POST to the server. The sender create one subprocess for each file. Since there are three test data files, there are three subprocesses. Senders are running on the host instead of docker containers. The sender sends requests to the ingress port of the server so that requests are load balanced to server running on different containers.

3.4 Exchange Rate Modifier

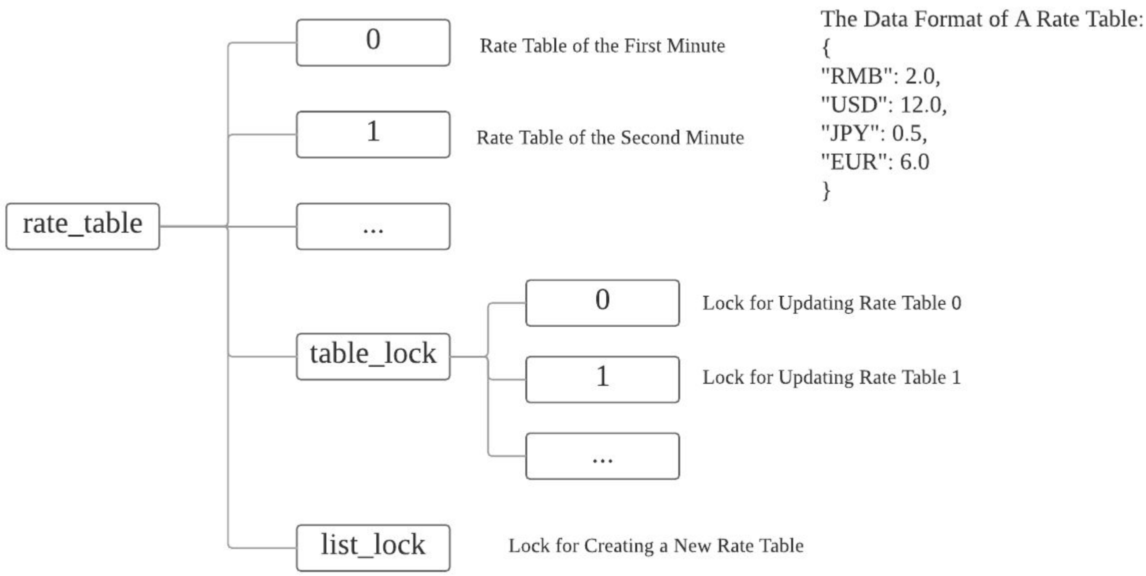
We implement a Python program to control the exchange rate at each moment. The program creates a node in zookeeper as root, and create four processes using fork. Each process controls the exchange rate for one currency and writes to zookeeper every minute.

We store every version of rate table in zookeeper so rate tables can be seemed a versioned link-list. The exchange rate of all currencies in every minute are stored in one node, i.e. `` `/path/time/` ``. Data of each node is a string of JSON object containing `<currency name, rate>` pairs.

Concurrency control becomes a tough problem since we have four processes to modify the same rate table. Fortunately, zookeeper already provides us a **distributed lock** interface. We have to deal with two situations:

- First is that a process have to get the list_lock before it create a new version of rate table and release lock after it create it.
- Second is that a process have to get the table lock corresponding version before it modify the rate table.

The data hierarchy in zookeeper shows as below:



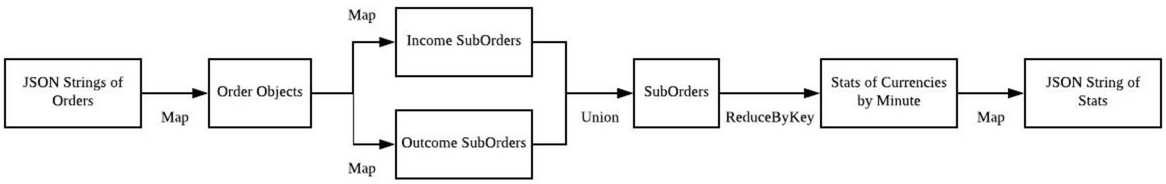
3.5 Spark Order Processor

The Spark Order Processor is implemented by Spark Streaming technology. The processor will fetch orders from Kafka cluster every specified period of time, process the data through Map-Reduce operations, and write the results into Mongo Cluster.

During the data processing period, the processor should read exchange rate from Zookeeper Cluster. If it failed to read the rates, the processor would hang and keep retrying until it got the correct data.

The detailed processing steps are as follows.

- Parse JSON strings into Order objects.
- Divide each order into two suborders, one for income and another for outcome (including reading rates from zookeeper).
- Reduce the suborders by the key of a combination of minute and currency.
- Convert the statistics back into JSON strings, ready to be written into MongoDB.



3.6 Baseline Program

In order to verify our whole program, we wrote a baseline program, which just simply reads orders from test_data files and calculates the correct results without interacting with other modules. The correct results as shown below:

```
➜ validator python3 baseline.py
name   income       expend     time
RMB    372662.50    132965.00  2018-01-01 00:00
USD    32624.00     123136.00  2018-01-01 00:00
JPY    1984716.00   134131.00  2018-01-01 00:00
EUR    97804.59     150895.00  2018-01-01 00:00
RMB    361208.20    132965.00  2018-01-01 00:01
USD    33472.53     123136.00  2018-01-01 00:01
JPY    1678388.92   134131.00  2018-01-01 00:01
EUR    98282.42     150895.00  2018-01-01 00:01
RMB    350795.10    132965.00  2018-01-01 00:02
USD    34307.07     123136.00  2018-01-01 00:02
JPY    1459583.81   134131.00  2018-01-01 00:02
EUR    98744.80     150895.00  2018-01-01 00:02
RMB    341287.49    132965.00  2018-01-01 00:03
USD    35128.16     123136.00  2018-01-01 00:03
JPY    1295479.63   134131.00  2018-01-01 00:03
EUR    99192.55     150895.00  2018-01-01 00:03
RMB    332572.39    132965.00  2018-01-01 00:04
USD    35936.12     123136.00  2018-01-01 00:04
JPY    1167843.54   134131.00  2018-01-01 00:04
EUR    99625.92     150895.00  2018-01-01 00:04
RMB    324554.20    132965.00  2018-01-01 00:05
USD    36731.04     123136.00  2018-01-01 00:05
JPY    1065734.50   134131.00  2018-01-01 00:05
EUR    100046.58    150895.00  2018-01-01 00:05
```

3.7 Result Viewer

We found that there may existing multiple results in mongo, which need to be combined. This is a problem that we will discuss in 4.5. We believe that our practice is the best among all the potential solutions. We wrote a viewer program, which reads data from mongo cluster, combines some results and displays the final results. Our final results as shown below:

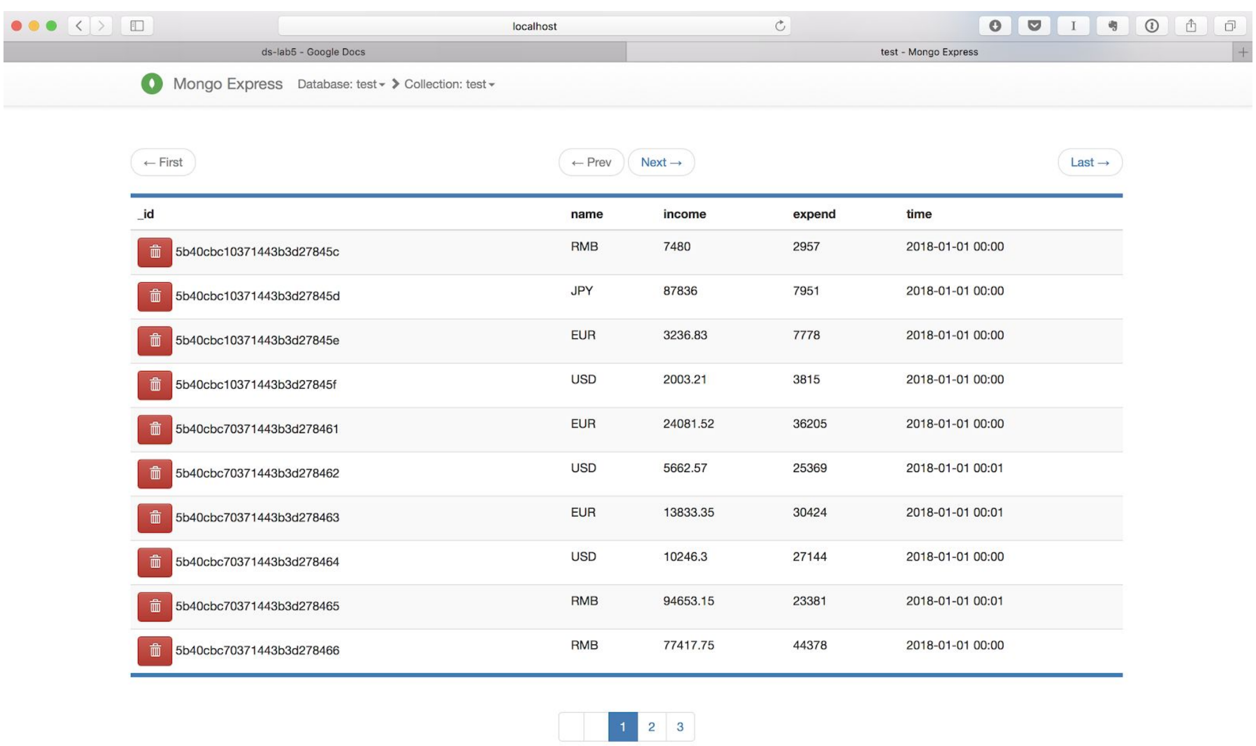
```
➜ result_combiner ./view node1:27017,node2:27018 test test
name   income       expend     time
RMB    372662.50    132965.00  2018-01-01 00:00
USD    32624.00     123136.00  2018-01-01 00:00
JPY    1984716.00   134131.00  2018-01-01 00:00
EUR    97804.59     150895.00  2018-01-01 00:00
RMB    361208.20    132965.00  2018-01-01 00:01
USD    33472.53     123136.00  2018-01-01 00:01
JPY    1678388.92   134131.00  2018-01-01 00:01
EUR    98282.42     150895.00  2018-01-01 00:01
RMB    350795.10    132965.00  2018-01-01 00:02
USD    34307.07     123136.00  2018-01-01 00:02
JPY    1459583.81   134131.00  2018-01-01 00:02
EUR    98744.80     150895.00  2018-01-01 00:02
RMB    341287.49    132965.00  2018-01-01 00:03
USD    35128.16     123136.00  2018-01-01 00:03
JPY    1295480.01   134131.00  2018-01-01 00:03
EUR    99192.55     150895.00  2018-01-01 00:03
RMB    332571.97    132965.00  2018-01-01 00:04
USD    35936.12     123136.00  2018-01-01 00:04
JPY    1167843.54   134131.00  2018-01-01 00:04
EUR    99626.56     150895.00  2018-01-01 00:04
RMB    324554.20    132965.00  2018-01-01 00:05
USD    36731.04     123136.00  2018-01-01 00:05
JPY    1065734.50   134131.00  2018-01-01 00:05
EUR    100046.58    150895.00  2018-01-01 00:05
```

3.8 Handling Precision

We round every result of order up with 2 digits of accuracy before adding to results. We think this approach could meet the accuracy requirement in this lab.

3.9 Screenshots:

After running our process, we get results in mongo. We can browse those results in mongo express web ui:



_id	name	income	expend	time
5b40cbc10371443b3d27845c	RMB	7480	2957	2018-01-01 00:00
5b40cbc10371443b3d27845d	JPY	87836	7951	2018-01-01 00:00
5b40cbc10371443b3d27845e	EUR	3236.83	7778	2018-01-01 00:00
5b40cbc10371443b3d27845f	USD	2003.21	3815	2018-01-01 00:00
5b40cbc70371443b3d278461	EUR	24081.52	36205	2018-01-01 00:00
5b40cbc70371443b3d278462	USD	5662.57	25369	2018-01-01 00:01
5b40cbc70371443b3d278463	EUR	13833.35	30424	2018-01-01 00:01
5b40cbc70371443b3d278464	USD	10246.3	27144	2018-01-01 00:00
5b40cbc70371443b3d278465	RMB	94653.15	23381	2018-01-01 00:01
5b40cbc70371443b3d278466	RMB	77417.75	44378	2018-01-01 00:00

4. Problem Encountered

4.1 Service Name v.s. Hostname

In docker, we can use service name to connect to a node. For example, in docker-compose file, we define a service like

```
services:
  service_name:
    ...
```

Running `nslookup` on another container can resolve correctly. However, the container has a generated hostname different from service name if not provided. In this case, if we set the service address to service_name:port in the server, instead of 0.0.0.0 or localhost, you may encounter `Connection closed` error because routed requests from the service mesh will not use the service name of containers.

Solution for this is either specifying hostname in docker-compose file or listening to 0.0.0.0.

4.2 Unresolvable Hostname

We had several hostname added in `/etc/hosts/` of containers but still unresolveable in some programs. We found that this is the desired behavior of Alpine Linux, which bundles musl-libc which provides inconsistent APIs with glibc. Configuration in `/etc/resolv.conf` in not accepted and thus `/etc/hosts` does not work. Configuring to use glibc is a quick fix for this.

4.3 Unable to Receive Message from Kafka in Spark Streaming

Kafka provides two kinds of APIs: the high-level receiver based API and the low-level direct API. The direct API is to use `createDirectStream` to create a Kafka stream which reads the offset directly from Kafka brokers and is more efficient. However we were mysteriously not able to read anything from the Kafka stream and finally we decided to switched to receiver-based API.

4.4 Fail to Reduce in Spark when Using Custom Key

When writing codes of Spark Order Processor, we found the ‘reduceByKey’ method couldn’t work will when the key is an object declared by ourselves, even the class had inherit the ‘Ordered[T]’ trait and implemented the ‘compareTo’ method. Finally we found from StackOverFlow website that ‘reduceByKey’ method uses ‘equals’ method and the hashcode of an object to judge whether the two keys are the same or not. Therefore, to ensure this method works well, we should override ‘equals’ method and ‘hashCode’ method.

4.5 Results Need to Be Combined

Because this is a simulation system, we have to process orders according the timestamp pinned in it. The network delivering is out-of-order. Some orders with newer timestamps may arrived in advance. Then orders within same minute may be processed in difference spark procedures. So there are two solutions:

- In every spark procedure, before storing results into mongo, read from mongo and combine result. **However, this approach will introduce new data races because reading and writing same mongo document at the same time, which is a huge damage to system performance.** So we deny this approach.

- We delay the combination until we want to read results from mongo. This approach is still correct because all the results we need are still keeped in mongo. There is no need to combine result in advance. We think this approach is the better than the approach above. So we finally choose this approach.

5. Project Structure

The whole project directory as shown below:

```
(master)⚡% tree . -L 2
.
├── README.md
├── app
│   ├── exchange_rater
│   ├── order_processor
│   ├── receiver
│   ├── result_combiner
│   ├── sender
│   └── validator
├── doc
│   └── DTSS_doc.pdf
├── docker-swarm
│   ├── express.yml
│   ├── hadoop
│   ├── kafka.yml
│   ├── mongo-cluster.yml
│   ├── restart.sh
│   └── zk.yml
└── test_data
    ├── test_data_0.json
    ├── test_data_1.json
    └── test_data_2.json

11 directories, 10 files
```

- The `/docker-swarm` directory consists of all the configuration files that we need to deploy the whole system, including zookeeper, kafka, hadoop, spark and mongo cluster.
- The `/app/exchange_rater` directory consists of a python program that is responsible for modifying the rate table in zookeeper.
- The `/app/order_processor` directory consists of the main spark program.
- The `/app/receiver` directory consists of the server program.
- The `/app/result_combiner` directory consists of the result_viewer program.
- The `/app/sender` directory consists of the sender program.

6. Student Workload

Jianbang Yang: Environment Setup, Exchange Rate Modifier, Validator and Documentation
Wuwei Lin: Environment Setup, Sender, HTTP Server and Documentation
Tianrui Chen: Environment Setup, Spark Program and Documentation

7. Reference

[1] Hadoop cluster: https://github.com/vinx13/docker-hadoop
[2] Kafka: https://github.com/wurstmeister/kafka-docker
[3] Zookeeper: https://docs.docker.com/samples/library/zookeeper
[4] Mongo cluster: https://github.com/stefanprodan/mongo-swarm