

Design Document

Class Design

In this lab, we use object-oriented design to implement the network. In this network, two alive peers are set as two traders and each of other peers will perform as a buyer or a seller. Trader is responsible for handling all buy requests, buyer is responsible for buying products and seller is responsible for selling products. And also there is a peer performing as a warehouse which is responsible for storing all products. Only trader knows the address of warehouse and all buying request will trigger the decrement of products in warehouse and all selling request will trigger the increment of products in the warehouse. For the trader, it can choose whether to use cache. Trader without cache will forward all buying requests and selling requests to the warehouse, while trader with cache will handle buying requests and selling requests with cache and trader flush cached requests to the warehouse under certain conditions.

We use two classes to define the peers in the network: peer and warehouse

Warehouse

address: The address of this peer. This attribute is a tuple which contains IP address and port number, such as `('127.0.0.1', 8000)`

server: A socket server used to receive messages. In each peer, there is a non-stop server that receive messages.

item_list: A dictionary to store the stock information.

over_selling_count: A counter to count the over selling cases.

Peer

address: The address of this peer. This attribute is a tuple which contains IP address and port number, such as `('127.0.0.1', 8000)`

peer_id: The ID of this peer. It is a integer from 1 to N which is the number of peers in a network.

server: A socket server used to receive messages. In each peer, there is a non-stop server that receive messages.

productID: The category of the products for this peer buying or selling.

productNum: The number of the products for this peer buy or selling.

trader_list: A list to store all traders' addresses.

istrader: This is a Boolean variable. True means that this peer is trader, False means that this peer is not trader.

clock: A logic clock, Lamport Clock.

is_electing: This is a Boolean variable. True means that this peer is in the election process, False means that this peer is not in the election process. This variable is used to avoid repeat election.

isBuyer: This is a Boolean variable. True means that this peer is a buyer, False means that this peer is not a buyer.

isSeller: This is a Boolean variable. True means that this peer is a seller, False means that this peer is not a seller.

cache: This is a dictionary to store stock information of warehouse.

warehouse: This is the address of warehouse.

alive_peers: A list to store all alive peers' addresses.

use_cache: This is a Boolean variable. True means that this trader uses cache, False means that this peer does not use cache.

request_list: This is a list to cache all requests received by this trader. If the length of this list is 5, this trader will send it to the warehouse to increment or decrement products.

Process Design

There are 10 different messages in this network. The definition is shown below:

0,1 election.

2 trader to warehouse, buy request. Reply the number of products buyer can get.

3 trader to warehouse, sell request.

4 trader to warehouse, request for stock information.

5 seller to trader, sell products.

6 buyer to trader, buy products. Trader will reply the number of products buyer can get.

7 trader to trader: heartbeat.

8 trader to peers: one trader has dead. Update traders.

9 trader to warehouse: cache flush

Warehouse

The warehouse may receive 4 categories of messages: 2, 3, 4, 9

If it receives a message with 2, which means a buying request. It will calculate the number of products the buyer can get and decrement from the stock. Then it will reply this number to the trader.

If it receives a message with 3, which means a selling request. It will add the number of selling to the stock.

If it receives a message with 4, which means a request for looking up stock information. It will send its stock information to the trader.

If it receives a message with 9, which means a cache flush request. It will flush the received request list to the stock information. To implement the eventual consistency, it need to sort the received request list by logic clock at first. We use Lamport Clock as the logic clock.

The process defined above is implemented by `process()` function in the Warehouse class.

Peer

Peers can be divided into three categories: Buyer, Seller and Trader. Each peer will call the corresponding functions according to its own category. The main process is defined in the `process()` function in the Peer class. The trader election logic is pretty similar to lab2.

Buyer

For a buyer, this peer will send buying requests with 6 continuously. Before each sending, it will select a trader from the trader list randomly. After sending, it will wait for the reply.

The process defined above is implemented by `buyer_process()` function in the Peer class.

Seller

For a seller, this peer will send selling requests with 5 continuously. Before each sending, it will select a trader from the trader list randomly.

The process defined above is implemented by `seller_update_stock()` function in the Peer class.

Additionally, each buyer and seller need a listening process.

In the listening process, the peer will receive messages continuously. If it receives a message with 8, which means that a trader has dead, it will read the alive traders' addresses from a file to update trader list. This listening process is implemented by `peer_listening()` function in the Peer class.

Trader

For a trader, this peer may receive 3 categories of messages: 6, 5, 7

If it receives a message with 6, which means a buying request, the trader without cache or the trader with an expired cache will send a message with 2 to the warehouse to buy products and wait for a reply from the warehouse so that it can reply to the buyer, while the trader with a valid cache will update its cache and reply to the buyer.

If it receives a message with 5, which means a selling request, the trader without cache or the trader with an expired cache will send a message with 3 to the warehouse to sell products, which the trader with a valid cache will update its cache.

In addition, the trader with a valid cache will flush cached requests to the warehouse if the number of cached requests reach 5 after handling buying requests and selling requests. This flush logic is implemented by `cache_flush()` function.

If it receives a message with 7, which means a heartbeat message, it will print 'alive' in the console.

In the beginning of the process described above, the trader with an expired cache need to look up the stock information from the warehouse. This look up logic is implemented by `lookup()` function.

The whole process shown above is implemented by `trader_listening()` function.

Each trader needs an extra heartbeat process. Each trader will send heartbeat messages with 7 to all other traders continuously. If there is an exception, which means a trader has dead, the trader will update the trader address file and send messages with 8 to all alive peers to tell them to update trader addresses.

This heartbeat process is implemented by `trader_process()` function.

Trade Off

To simplify the design we use socket module. The process designed by socket is a synchronized process. If we use an asynchronous module, the concurrency performance will be better.

We assume that each buyer only need one kind of products in one transaction to simplify the transaction process.

We use not-strict consistency to simplify the design.

Improvements & Extensions

We can use asynchronous module to reduce the blocking time.

The network may lose messages in some cases. Communication quality and efficiency between peers need to be improved.

How to Run

Modify peers in Run.py as you want. If `state=True` then this peer is a seller, else this peer is a buyer.

```
1 | peer = Peer(address, peer_id, state, product_ID, product_num, warehouse_addr,  
    | use_cache=False)
```

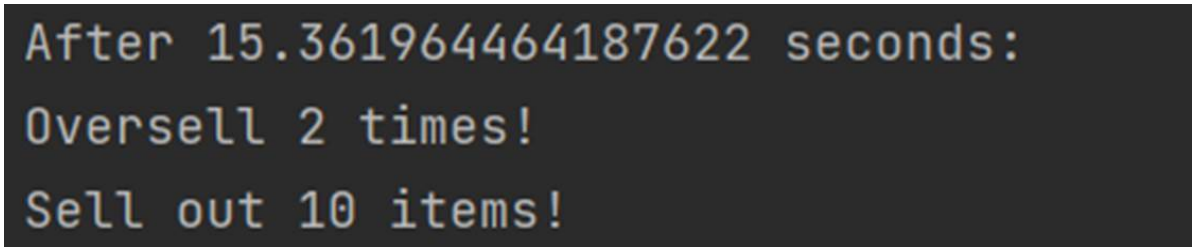
Just run this script. Then all the peers and warehouse will be run concurrently.

Test

cache-based approaches:

In the experiment, all of peers sell or buy product 0. Seller sells 5 product 0 every 5 seconds. Buyer buy a random quantity of product 0. The random quantity is between 1 and 5.

First sell happens about 15 seconds after the program start.

A terminal window with a dark background and light-colored text. The text displays the results of a simulation after 15.36 seconds.

```
After 15.361964464187622 seconds:  
Oversell 2 times!  
Sell out 10 items!
```

When there are 2 sellers and 2 buyers, after about 71 seconds, there are 75 oversellings and 111 products has been shipped out.

```
After 71.35876536369324 seconds:  
Oversell 75 times!  
Sell out 111 items!
```

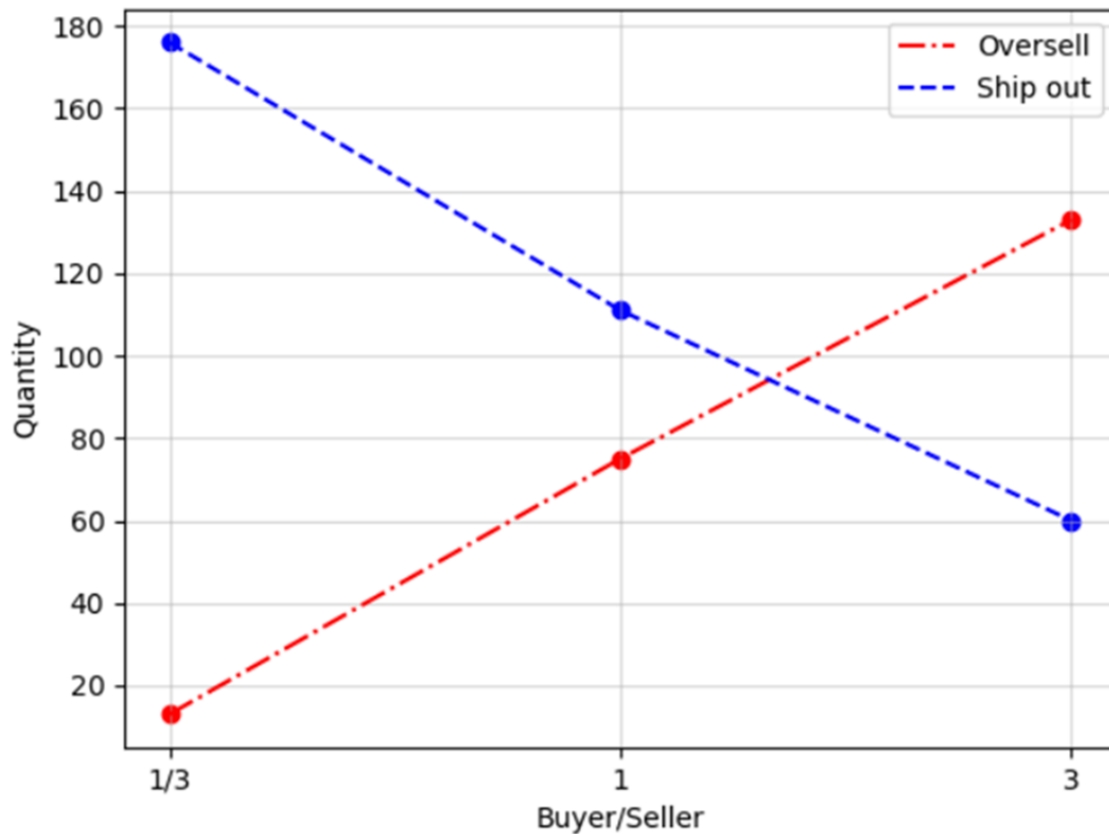
When there are 3 sellers and 1 buyers, after about 71 seconds, there are 13 oversellings and 176 products has been shipped out.

```
After 71.15880012512207 seconds:  
Oversell 13 times!  
Sell out 176 items!
```

When there are 1 sellers and 3 buyers, after about 71 seconds, there are 133 oversellings and 60 products has been shipped out.

```
After 71.46312761306763 seconds:  
Oversell 133 times!  
Sell out 60 items!
```

Here is the plot of the buyer-seller ratio and oversell/ship out.



\

Obviously, when the ratio become larger, there are more overselling and less shipping out. This is because, when there are more seller, there will be more items processed successfully with trader's cache because the selling items is enough at most times. When there are more buyer, the selling items is extremely deficient. So there will be more oversellings. The sum of the overselling and shipping out is almost the same for different ratios.

Cache-less approaches:

When there are 2 sellers and 2 buyers, after about 71 seconds, there are 122 products has been shipped out.

```
After 71.32192802429199 seconds:
Oversell 0 times!
Sell out 122 items!
```

The shipping out number per second for cache-less approach is 2.18 items/s. The shipping out number per second for cache-based approach is 1.98 items/s. But there happens a lot of overselling for the cache-based approaches. If we compared the speed of processing request between 2 approaches, the approach with cache is significant more efficiently. I think if there are more sellers, then the shipping out number per second will become faster easily.

Fault tolerance:

When I shut down 1 trader, the other trader will detect this at about 24 second.

```
After 24.12411403656006 seconds:  
The other trader dead. Sending message to all peer.
```

Then, the trader will tell all peers to only do trade with him.

```
After 24.125115633010864 seconds:  
1 trader dead. Reset trader list.
```

At about 24 second, warehouse has shipped out 30 items with 2 traders.

```
After 24.520503520965576 seconds:  
Oversell 0 times!  
Sell out 30 items!
```

At about 48 second, warehouse has shipped out 75 items with only 1 traders.

```
After 49.84261417388916 seconds:  
Oversell 0 times!  
Sell out 75 items!
```

So, the within the first 24 seconds, 2 traders sold out 30 items. After that, for the next 24 second, the alive trader sold out 45 items. I think this is because I use a thread pool for processing the requests. The threading module in Python is inefficient. Also, with only 1 trader alive, all the peers will no longer need to randomize a trader to trade with. Thus after shut down 1 trader, the speed of process is increased.