Design Document

Class Design

In this lab, we use object-oriented design to implement the network. The peers in the network can be divided into two categories, buyer and seller. We define a Seller class and a Buyer class to represent these two types of peers and we use socket to implement the communication among peers.

Seller

For Seller class, it has six attributes: address, connected_address, peer_id, product_id, num_items and server:

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)`

connected_address: A list of address tuples. This attribute contains all neighbors this peer can connect directly, such as `[('127.0.0.1', 8001), ('127.0.0.1', 8004)]`

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

product_id: The category of the products this seller sells. There are three different categories of products. We use number 0, 1, 2 to represent them.

num_items: The number of items this peer can sell. When this seller complete one transaction, this attribute will decrement one. If the value of this attribute reach zero, this seller will pick one product_id randomly and set this attribute to the original value and continue to sell new products.

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

Buyer

For Buyer class, it has ten attributes: address, connected_address, peer_id, request_items, hop_count, server, success, request_buffer, request_times and reply_times:

address: Same as the attribute in Seller class.

connected_address: Same as the attribute in Seller class.

peer_id: Same as the attribute in Seller class.

request_items: A list of all request product_id. Each element of this list is the product ID this buyer want to buy which is from 0 to 1. This list is generated randomly, such as `[1, 2, 1, 1, 0, 0, 2, 0, 2]`

hop_count: The maximum distance one request can be propagated in the network. It is a value from 1 to N.

server: Same as the attribute in Seller class.

success: A list of flags with the same length of request_items. Each flag of this list has two values 0 and 1. 0 means the request at the corresponding position in the request_items has not been completed and 1 means the request at the corresponding position in the request_items has been completed. All initial values of this list is 0, such as `[0, 0, 0, 0, 0, 0, 0, 0, 0]`. If one request is completed, the corresponding position will be change to 1.

request_buffer: A list of all messages this buyer has send. Each element has a one-to-one correspondence with the element in request_items.

request_times: A list of all message request timestamps. Each element has a one-to-one correspondence with the element in request_items.

reply_times: A list of all message reply timestamps. Each element has a one-to-one correspondence with the element in request_items.

Function Design

The format of the message propagating in the network is `request_category|product_id-index|path|hop_count|(seller_address)|(peer_id)` . The `request_category` has three possible value:0, 1 and 2. 0 means lookup request, 1 means reply request and 2 means buy request.

There are three functions to implement the action of peers: `lookup`, `reply` and `buy` .

`lookup(peer, fields)` : `peer` is the object of the peer. `fields` is a list generate by `split('-')` function from the original request. In this function, a peer decrement the `hop_count` by 1 and append its `address` to `path`. Then it transform `fields` to a string and send the request to its neighbors.

`reply(fields)` : a peer get the last address of `path` to get the next address it should connect. Then the last address is popped from `path` and `fields` will be transformed to string. The peer send the request to the next peer.

`buy(fields)` : a buyer get the `seller_address` from `fields`. The buyer send a buy request to the seller and waits for a reply.

There is a function designed just for buyer:

`init_lookup(self)` : Each buyer should call this function at the beginning of its process. This function is used to construct all lookup requests from `request_items` and send them to the direct neighbors of each buyer.

Process Design

Once a peer receives a lookup request:

For a seller, it will check the `product_id` of this request whether is equal to its selling `product_id`.

If the result of the check is true, it will change the `request_category` to 1. Then it appends its `address` and `peer_id` to the message and call `reply` function to propagate reply request.

If the result is false, it will check whether the `hop_count` is equal to 1. If `hop_count` is 1, it just drop this request. If not, it will call `lookup` function to propagate the request.

For a buyer, it just check the `hop_count` and call `lookup` function to propagate the request.

Once a peer receives a reply request:

For a seller, it just call `reply` function to propagate the request.

For a buyer, it need to check `path` whether is equal to empty string. If the result is true, it means that the request has arrived at the target peer and then this buyer can call `buy` function and use the `seller_address` to send buy request and make transaction. If the result is false, it will call `reply` function to propagate this request.

Once a peer receives a buy request:

Only the seller can receive buy request. A seller need to check the requiring `product_id` whether is equal to the current `product_id` it sells. If the result is true, it will decrement the `num_items` by 1 and reply a success message to the buyer. If the result is false, it will reply a failure message to the buyer.

In a buying process, the buyer will wait for the seller's reply. If the buyer get a success message. it will record the reply time and compute the response time and change the corresponding position of `success` list to 1 to drop duplicated reply request. If the buyer get a failure message, it will send lookup request again.

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 one product in one transaction to simplify the transaction process.

We only use one thread in each process to simplify the transaction process.

Improvements & Extensions

We can use asynchronous module to reduce the blocking time.

We can use multi threads to improve the concurrent performance.

How to run

You need PyCharm to run the program.

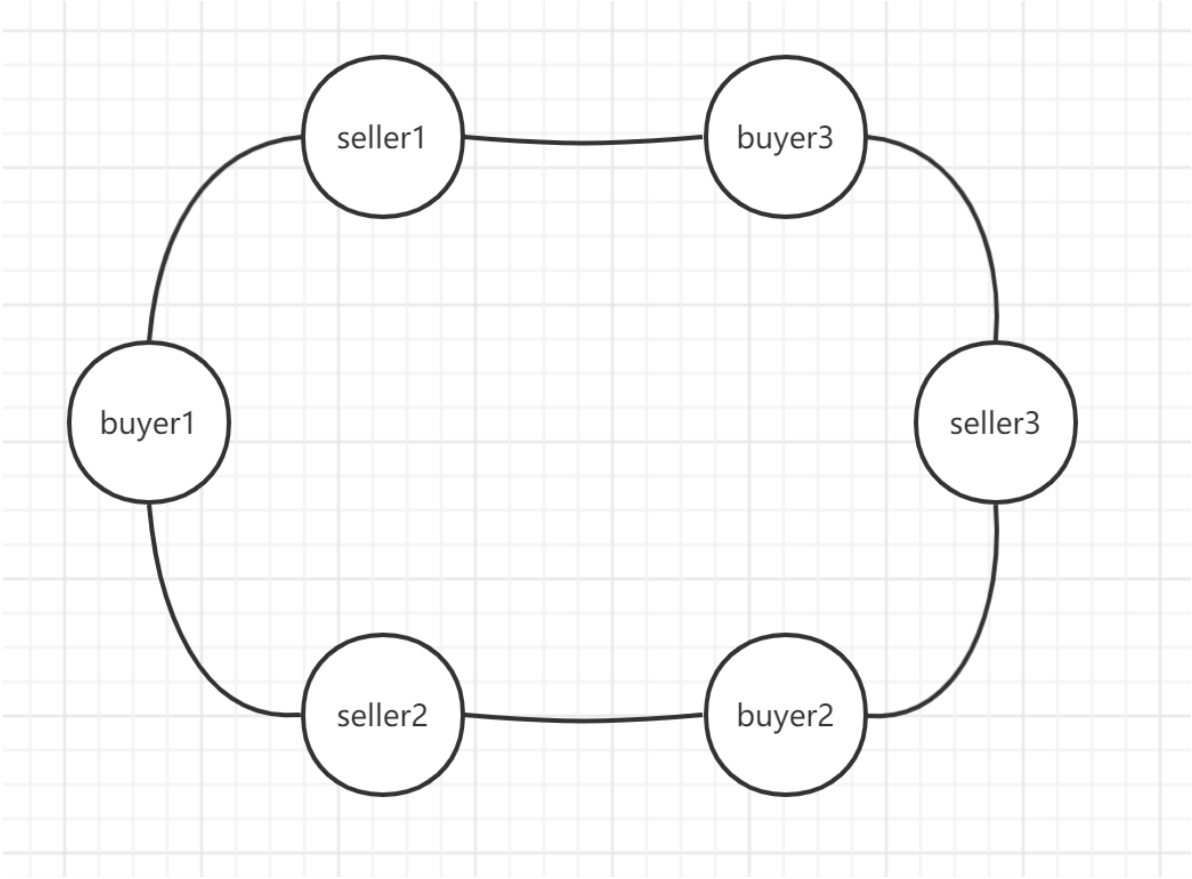
Firstly, you should run Seller1.py, Seller2.py and Seller3.py

Secondly, you should run Buyer1.py, Buyer2.py and Buyer3.py

After waiting about 5 seconds, you can see that the console starts to appear running information.

Test

We build a simple network. The network has total 6 nodes: 3 seller and 3 buyer. The 6 nodes connects together to build a simple loop: seller1--buyer1--seller2--buyer2--seller3--buyer3--seller1.



Here is the purchase logs of each buyer.

The peer id of buyer1, buyer2 and buyer3 is 2, 3 and 4.

We initialize that 3 sellers sell different items. The request items list of each buyer is a randomized integer list of a given length 30. The hop_count is set to 5.

Each seller have 15 items in stock. For buyer 1, he can not buy any product2 because all the product2 are sold out and each seller does not sell product2 again.

```
2022-10-19 02:32:24, 2 purchase product1 from 6!
2022-10-19 02:32:26, 2 purchase product0 from 1!
2022-10-19 02:32:27, 2 purchase product0 from 1!
2022-10-19 02:32:27, 2 purchase product1 from 6!
2022-10-19 02:32:27, 2 purchase product0 from 1!
2022-10-19 02:32:28, 2 purchase product0 from 1!
2022-10-19 02:32:28, 2 purchase product1 from 6!
2022-10-19 02:32:28, 2 purchase product1 from 6!
2022-10-19 02:32:29, 2 purchase product0 from 1!
2022-10-19 02:32:29, 2 purchase product0 from 1!
2022-10-19 02:32:29, 2 purchase product1 from 6!
2022-10-19 02:32:29, 2 purchase product1 from 6!
2022-10-19 02:32:30, 2 purchase product1 from 6!
2022-10-19 02:32:30, 2 purchase product1 from 6!
2022-10-19 02:32:30, 2 purchase product1 from 6!
2022-10-19 02:32:30, 2 purchase product0 from 1!
2022-10-19 02:32:30, 2 purchase product0 from 1!
2022-10-19 02:32:31, 2 purchase product0 from 1!
2022-10-19 02:32:31, 2 purchase product0 from 1!
2022-10-19 02:32:31, 2 purchase product1 from 6!
```

```
2022-10-19 02:32:28, 3 purchase product2 from 5!
2022-10-19 02:32:30, 3 purchase product2 from 5!
2022-10-19 02:32:31, 3 purchase product2 from 5!
2022-10-19 02:32:31, 3 purchase product2 from 5!
2022-10-19 02:32:32, 3 purchase product2 from 5!
2022-10-19 02:32:32, 3 purchase product2 from 5!
2022-10-19 02:32:33, 3 purchase product2 from 5!
2022-10-19 02:32:34, 3 purchase product1 from 6!
2022-10-19 02:32:35, 3 purchase product1 from 6!
2022-10-19 02:32:35, 3 purchase product1 from 6!
2022-10-19 02:32:37, 3 purchase product1 from 6!
2022-10-19 02:32:38, 3 purchase product1 from 6!
2022-10-19 02:32:40, 3 purchase product1 from 6!
2022-10-19 02:32:42, 3 purchase product1 from 6!
2022-10-19 02:32:43, 3 purchase product1 from 6!
2022-10-19 02:32:56, 3 purchase product0 from 1!
2022-10-19 02:32:58, 3 purchase product0 from 1!
2022-10-19 02:33:00, 3 purchase product0 from 1!
2022-10-19 02:33:01, 3 purchase product0 from 1!
2022-10-19 02:33:02, 3 purchase product0 from 1!
2022-10-19 02:33:04, 3 purchase product0 from 1!
2022-10-19 02:33:05, 3 purchase product0 from 1!
2022-10-19 02:33:07, 3 purchase product0 from 1!
```

```
2022-10-19 02:32:27, 4 purchase product0 from 1!
2022-10-19 02:32:28, 4 purchase product0 from 1!
2022-10-19 02:32:28, 4 purchase product0 from 1!
2022-10-19 02:32:28, 4 purchase product0 from 1!
2022-10-19 02:32:29, 4 purchase product0 from 1!
2022-10-19 02:32:29, 4 purchase product0 from 1!
2022-10-19 02:32:30, 4 purchase product0 from 1!
2022-10-19 02:32:30, 4 purchase product0 from 1!
2022-10-19 02:32:30, 4 purchase product2 from 5!
2022-10-19 02:32:31, 4 purchase product2 from 5!
2022-10-19 02:32:31, 4 purchase product0 from 1!
2022-10-19 02:32:31, 4 purchase product2 from 5!
2022-10-19 02:32:31, 4 purchase product0 from 1!
2022-10-19 02:32:32, 4 purchase product2 from 5!
2022-10-19 02:32:33, 4 purchase product2 from 5!
2022-10-19 02:32:33, 4 purchase product2 from 5!
2022-10-19 02:32:33, 4 purchase product2 from 5!
2022-10-19 02:32:55, 4 purchase product1 from 6!
2022-10-19 02:32:56, 4 purchase product1 from 6!
2022-10-19 02:32:56, 4 purchase product1 from 6!
2022-10-19 02:32:57, 4 purchase product1 from 6!
2022-10-19 02:32:58, 4 purchase product1 from 6!
2022-10-19 02:32:59, 4 purchase product1 from 6!
2022-10-19 02:32:59, 4 purchase product1 from 5!
2022-10-19 02:33:00, 4 purchase product1 from 6!
```

We design 3 performance test. Test1 is a test that The buyer have sufficient items and each seller can easily get what they want.

Here is the request times from sending request to successfully purchasing the item of each buyer.

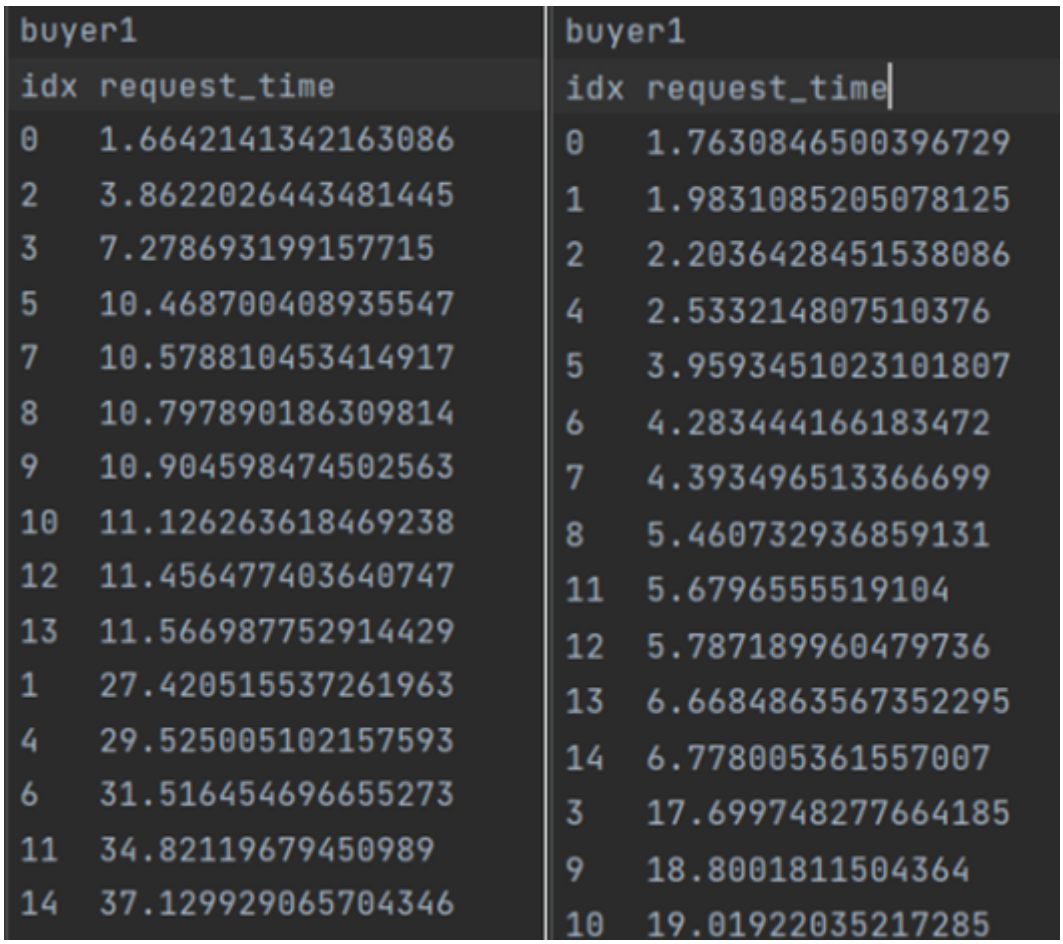
buyer1	
idx	request_time
1	3.3861355781555176
2	6.770829200744629
5	9.605874061584473
6	9.824414491653442
7	9.933767318725586
9	10.368786334991455
10	10.913642168045044
12	11.240863800048828
13	11.45942735671997
15	11.897641897201538
16	12.006340265274048
17	12.11272406578064
18	12.332063436508179
20	12.770294904708862
22	13.424484491348267
25	13.752885341644287
27	13.862383127212524
28	14.18947982788086
0	30.1452214717865
3	30.940474033355713
4	31.15783429145813
8	31.484673976898193
11	31.70483899116516
14	32.25098633766174
19	33.019267559051514
21	33.24109148979187
23	33.4589147567749
24	33.78887414932251
26	34.55511403083801
29	35.09450006484985

buyer2	
idx	request_time
0	4.3734118938446045
3	10.921722412109375
4	12.668751955032349
5	13.324651956558228
13	14.743064165115356
2	16.525227308273315
23	17.033252716064453
6	18.270363807678223
7	19.363686323165894
26	19.544656991958618
9	21.005881309509277
28	21.18800377845764
15	23.853692293167114
16	25.06103014945984
17	26.04259705543518
18	26.9217267036438
19	27.798288822174072
24	29.00396227836609
27	29.76781940460205
29	30.206705331802368
1	39.25542116165161
8	40.89521098136902
10	41.14782214164734
11	41.366302251815796
12	41.80303168296814
14	42.01805639266968
20	42.7797966003418
21	42.9993417263031
22	43.2176718711853
25	43.32787823677063

buyer3	
idx	request_time
4	6.957645654678345
10	8.044337749481201
13	8.372920513153076
14	8.919833421707153
16	9.135977268218994
19	9.899433851242065
20	10.556426048278809
0	11.513596296310425
24	11.617435693740845
2	11.840496063232422
5	12.169044971466064
8	12.715587854385376
11	13.256350994110107
15	14.13139271736145
18	14.677155017852783
23	15.332162141799927
28	15.876780033111572
1	28.949415683746338
3	30.046854972839355
6	30.81510090827942
7	31.798563480377197
9	32.66322708129883
12	34.30114006996155
17	36.04254674911499
21	37.90832281112671
22	39.439910888671875
25	40.857826709747314
26	42.388344526290894
27	43.595078468322754
29	44.14331650733948

We can easily found that because of so many requests coming into the same port, some of the requests need to wait for a long time.

For test 2, we want to compare the request time of different network. However, if we add only 1 edge to the network, the request number in the total network increases so much that our port do not have the capability to deal with them. As a result, we must reduce the request number of each buyer to 15. The left side of the following picture is the network after adding 1 edge. It is obvious that the request time of the new network is greater than the original network.



buyer1

idx	request_time
0	1.6642141342163086
2	3.8622026443481445
3	7.278693199157715
5	10.468700408935547
7	10.578810453414917
8	10.797890186309814
9	10.904598474502563
10	11.126263618469238
12	11.456477403640747
13	11.566987752914429
1	27.420515537261963
4	29.525005102157593
6	31.516454696655273
11	34.82119679450989
14	37.129929065704346

buyer1

idx	request_time
0	1.7630846500396729
1	1.9831085205078125
2	2.2036428451538086
4	2.533214807510376
5	3.9593451023101807
6	4.283444166183472
7	4.393496513366699
8	5.460732936859131
11	5.6796555519104
12	5.787189960479736
13	6.6684863567352295
14	6.778005361557007
3	17.699748277664185
9	18.8001811504364
10	19.01922035217285

Lastly, we want to find out what will happen if there is no sufficient items for each buyer. Here we set the request number of each buyer to 30 and set the stock number of each seller to 15. It is obvious that when there are no sufficient items, the random product after selling out the previous product will not guarantee that all the buyer can buy whatever they want. It is obvious that buyer1 and buyer3 do not successfully purchase all the 30 items.

buyer1		buyer2		buyer3	
idx	request_time	idx	request_time	idx	request_time
0	3.280087947845459	0	8.968790054321289	1	5.363613128662109
2	3.4971954822540283	1	11.911839723587036	2	5.695026874542236
5	3.712289810180664	2	13.654430866241455	4	6.350473642349243
6	3.930507183074951	4	14.524663209915161	14	8.200270414352417
9	7.75736403465271	5	14.959221601486206	27	10.16129732131958
10	8.085963726043701	7	15.396867513656616	6	12.731335639953613
11	8.195242643356323	11	16.378753185272217	8	13.167564630508423
12	9.575984001159668	6	16.7062108516693 1	11	13.71287989616394
13	10.125244855880737	13	17.252723217010498	12	14.041531801223755
14	10.234085321426392	14	17.686835527420044	17	15.567196369171143
15	10.34317922592163	8	17.842975616455078	18	15.787078857421875
17	10.67053771018982	9	18.28453302383423	19	16.219481468200684
19	11.212064504623413	10	18.833690881729126	20	16.330535650253296
20	11.867761135101318	17	19.020131826400757	23	16.659578323364258
21	12.522879123687744	15	20.25246572494507	25	16.7677583694458
24	12.960362195968628	16	21.235922813415527	26	16.987847328186035
25	13.395992994308472	27	24.15513586997986	28	17.333566427230835
26	13.612971544265747	3	36.18036127090454	29	17.55305290222168
29	13.938233613967896	12	38.14633369445801	3	28.839950561523438
1	33.35613775253296	22	39.563236474990845	5	28.731353044509888
3	34.118964195251465	25	39.78279948234558	7	28.61869740486145
4	34.69395732879639	28	39.99863266944885	9	26.65608239173889
7	35.02274775505066	18	11.69110894203186	10	25.1259024143219
8	35.241854190826416	20	10.928327322006226	13	22.830899477005005
16	35.894009590148926	19	44.35384917259216	15	19.87965965270996
		21	44.027981758117676	16	18.786882877349854
		23	44.14133405685425		
		26	44.03178262710571		
		29	43.48714828491211		