Distributed Operating Systems
Asterix and the Bazaar

Sainyam Galhotra & Haresh Chudgar

April 11, 2016

1 Design

We use an unstructured Peer 2 Peer network and RPCs to implement the system. The following subsections present in detail about the various aspects of the system.

1.1 Pyro

We use Python Remote Object (PyRO) library to facilitate communication between peers. PyROs clean set of functions, tutorials and good documentation led us to choosing it over other libraries. Using PyRO, one of the machines is set up as a nameserver. Whenever a peer starts, it registers itself to the nameserver; this enables querying the peers URI just by name and linking to it.

1.2 Peer

We are maintaining a stateful peer which knows about all the other node ids in the network.

1.3 Leader Election

We have implemented the bully algorithm of leader election which floods the network with lookup and ok messages. We also implemented the ring algorithm over the overlay network as we earlier felt that we need to use the overlay network for leader election. In case of bully algorithm the node with highest id becomes the leader. Every peer waits for an ok message. If it receives an ok message then it cannot be a leader. In case a node resigns, it messages to one of the neighbor which starts the election all over again. The resigned node acts dormant during the election. It then passes the stock information to the new leader. The old leader acts as normal buyer, seller peer after the new leader is elected.

1.4 Message Passing:

We implemented the protocols as per the specifications where each seller registers the items with the trader and each buyer directly requests the trader for an item. The trader keeps the commission

and gives the earnings to the seller on each buy.

1.5 Synchronization

We used vector clocks for clock synchronization and total ordering. The choice was mainly due to easy debugging and testing. Each peer increments its clock component and sends a clock update message to all its peers except the trader. The peer then immediately calls the trader's buy function with the updated clock.

2 How it works?

2.1 Start

The network is started by deploying peers individually. Each peer on start registers itself on the nameserver. The last peer that joins the network starts leader election. Every peer is both a buyer and a seller until it becomes a trader.

2.2 Buying

The buyer contacts the trader with the item and its vector clock. The trader checks its clock with the buyer's vector clock for any messages it has missed. There are two cases to be considered here:

- If there are no missed message, the trader immediately responds to the message by first incrementing its clock component, multi casting clock update message to all peers except the buyer, and then responding to the buyer. The clock update is handled in a thread to decrease delay in response time. The prototype for buy function is given below for reference.
- If the vector clock indicates a missed message, it queues the request. When the trader finally receives the delayed message, it processes it as per the above step, and processes the queue either till it is empty or till it encounters a message with vector clock value which is out of order. At this point it has to wait for the correct message to arrive before it can process the rest of the queue.

```
def buy(self, item, peerID, timeStamp, requestID):
```

- item: Item that the buyer wants to buy
- timestamp: Value of vector clock at buyer's end
- requestID: requestID for buyer's reference

2.3 Selling

The trader maintains a dictionary with items as keys and list of (seller id, count) tuples as values, Upon requesting a buy request from a buyer, the trader looks up its item dictionary and randomly selects a seller from the list. The trader then increments its earnings, calls seller's function for its share of payment and finally calls buyer's sell function to confirm the sale.

```
def commissionForItem(self,item,commission):
```

- item: ID of the item that was sold
- commission: The seller's part of the sale.

```
def sell(self, isSold, item, timestamp, requestID):
```

- isSold: Boolean which specifies whether the item was sold or not
- item: The item which was sold
- timestamp: The value of vector clock when buy request was received
- requestID: The request id which was sent along with the buy request.

2.4 Leader resignation

When a leader resigns, a new election is started to choose a new leader. As soon as the new leader is elected, it needs to get the state of the market to resume trading. For this to work in harmony, we have implemented two different ways to capture the state of the market.

- File based: The trader maintains a file in a common location which is known to all peers. Whenever a trader resigns, the new leader reads the file and continues operation of the market.
- Distributed: Since the trader has to contact the seller for its commission, the seller is able to keep track of its inventory. The seller therefore has the complete state of its inventory. Whenever a trader is elected, the seller registers itself with the trader by sending the list of items in its inventory. So even if the old trader resigns, and a new trader is elected, the new trader will automatically receive updates from all sellers, and will be able to recreate the item inventory of the market.

3 Design Tradeoffs

In order to implement the P2P network, there were some tradeoffs which we have taken into account. Firstly in leader election we have done bully algorithm. The clock synchronisation has been implemented using vector clocks along with multicast. Every peer multicasts to the whole network about their buy request. This helps us to maintain total order. Another feature in our code is that the trader which resigns becomes a normal peer and starts buying and selling thereon. Its commission is maintained from the time it had started.

We tried these two different approaches as both of them follow the same procedure and the second method is handy when there is no shared data space.

3.1 Output log of the peer

We have printed the log of the server in the files present in two folders for two different test cases. In case of buyer, it contains the information about the item being bought, lookup calls made and the reply which it receives. In case of seller the information is present about reply and the information that the item has been bought.

4 Code functions

We have implemented a bunch of functions other than the lookup,buy and sell. The helper functions are the following :

- __init__ : Initialzes the object of the peer class. This assigns the IP adress, registers the object to name server and initializes the variables.
- startElection : Starts the election from the peer.
- election_lookup : The set of lookups needed during the election process
- broadcastElectionResult : Broadcast the result of election and inform every peer to start buying/selling
- registerItems : Register the set of items to be sold to the trader
- Vectorclock.py : A library which implements the different functions for vector clock which have been called by Peer.py for synchronisation

5 Testing

In order to test the correctness of our code, we tested on various scenarios which we explain in detail below :

5.1 Leader election

We tested the leader election in case of variable number of nodes. We also tested it in the situations when there are multiple leader elections started by different nodes at the same time. This helps us to verify thr correctness of our leader election algorithm.

5.2 Resignation

In order to test the correctness of re starting a new leader election and passing over control to the new leader, we instrumented the code where a leader resigns after some time. This worked perfectly fine and helps us ensure that the leader election procedure is correct.

5.3 Clock synchronisation

In order to test the correctness of working of clocks, we tested the correct updation of the clocks of buyers and sellers. This helps us ensure the correctness of the implementation of vector clocks to maintain the ordering in which buy requests are generated.

5.4 Clock + leader + resignation

In this test we test the correct working of our clocks even when new leaders get elected i.e. when some leader resigns and a new leader gets elected.

5.5 Clock + leader + resignation + larger delay to reorder

This is one of the msot important test cases as it tests the queue implementation which is used whenever an out of order message is received by the trader.

In this experiment we instrument the code to employ a larger delay on a particular link which helps us ensure that the messages received by the trader are out of order. In that case these messages get queued up and get processed later on in the order of their time stamps.

Testing our code on all these test cases helps us validate the correctness of our implementation.

6 Experimental evaluation

We analyzed the lag we receive in the response from the buyer in order to buy an item. For the exhaustive analysis, we ran the experiment with 6 peers where we tried different variations where the trader was once on one machine and after resigning the new trader is on another machine.

The outcomes have been shown in the table below :

Number of Buyers	Number of sellers	Time lag(ms)
5	1	12.33
4	2	12.2
3	3	12.24

Table 1: Local machine

This table just shows that there is not much difference in time even if we change the proportion of buyers and sellers. The main reason is that the trading process takes place through a single trader. It can be seen that the lag is marginally higher in case of large number of buyers as the trader takes some time to process the requests.

In order to analyze the lag we get over WAN, we ran three instances on our own machine (in University Wifi) and three instances on edLab server. The resutls which we report are an average computation over 1000 buy requests. With this we observed that the lag over WAN was higher as compared to the lag over a local machine. The outcomes were the following :

Number of Buyers	Number of sellers	Time lag(Ms)
5	1	146
4	2	143
3	3	147

Table 2: WAN

Another observation is that the time lag for the buyers which were on different machine was much higher as compared to the one on same machine. This is because of network delay and the message, vector clock being delivered over WAN.

In case of re-election, all the buying/selling processes pause and it takes around 535 milli seconds for the new leader to get elected and then the new leader to come into action. Over WAN this delay is slightly higher 1.45 sec as the current state is sent by resigning trader to the newly elected trader.