COMPSCI 677
Lab 1 - Design Document
Peer-to-peer Market

Overview of the system architecture and supplementary classes:

The peer to peer network is comprised of a certain number of peers pre-determined by us within the network topology. Each peer node is its own java process, containing a set of client and server threads. The number of client threads is equal to the number of neighbours for that node. Each node also spins one server thread that continuously listens to the assigned ports, and spins new threads to handle each incoming request. Each node runs on one of the 4 available EDLAB machines, which is specified earlier, and connects to its neighbours by retrieving the ports and hostNames for its neighbors through its config object.
Each peer has an associate Config object which stores information including:

1) Peer ID
2) Server port - the port that the server listens to
3) List of ports that its client threads will communicate with
4) List of Peer IDs for all its neighbours
5) PortMap - This is a hashmap that stores key value pairs for Peer IDs and their corresponding ports
6) LocationMap - This is a hashmap that stores which server a particular node is placed in, so that the appropriate replacement for “localhost” can be found

The Utilities.java file contains the code for building the topologies for the different test cases. This class creates a list of config objects (one for each node in the network) for each test case by specifying each field of the config object for each node manually. This information is then serialized and stored in a txt file, which is later deserialized by each Peer process in order to find its own config object (through getConfig() and getHostName()) and receive its place in the network and store its configuration information. This object allows a peer to know how many neighbours it has, which ones are buyers/sellers/both and set its server port and client ports accordingly.

There is a separate Message class that details the structure of a message that is passed around in the network. Which includes information such as:

1) Type - whether the message is a lookup request, reply or direct connection request
   The message object employs a custom hashcode and equals method and uses the requestId, sourcePeerId and type to make uniquely identifiable hashes for each message
2) RequestId - unique request that this message is associated with regardless of its type
3) SourcePeerId - Id of the node that this request originated from
4) destinationSellerId - the peer node identified as the seller for a particular lookup request
5) hopCount - number of hops the message has remaining before expiration

6) messagePath - the list of nodes a particular message has been handled by before a seller is identified
7) ProductId - Id of the product being sold/bought
8) Destination seller location - stores the location of the seller

The responseTimeTracker object is responsible for storing the timestamp at which a particular lookup request was generated as well as a list of timestamps for each response that was received by the peer for that particular lookup request. The purpose of this is to analyze response time metrics across the network.

Shared Data structures used within each node:

1) SharedRequestBuffer - A queue for received lookup requests that need to be forwarded further
2) sharedReplyBuffer - A queue for received reply messages that need to be forwarded to a specific node determined from the message path
3) requestHistory - A hashmap that keeps track of unique messages that have been encountered earlier in order to prevent flooding the same request multiple times through the same routes, and to prevent executing the same transaction twice
4) sharedTransactionBuffer - A queue for messages that need to be sent directly to the respective non-neighbour nodes in order to confirm the respective transactions
5) servicedRequests - A hashmap to keep a track of completed transactions
6) ResponseTimeMap - A hashmap to keep track of timestamps for request generation and responses received for each unique lookup request

How it works:

Main thread in each node:

1) A process is spun for a node
2) The main thread retrieves its config object and sets its ports accordingly
3) The main thread then decides its role randomly (buyer/seller/both)
4) The main thread then spins up 1 server thread and binds it to its server port
5) For the buyer nodes, A new thread is spun that is responsible for generating new look up requests for a random product every 5-7 seconds. This thread increments the request id for the peer and inputs the newly generated lookup request into the shared request buffer as well as the request history map. At the same time it also generates a timestamp for each new request and creates an entry in the ResponseTimeTracker HashMap. This thread generates 5 requests per buyer.
6) For the seller nodes, the main thread establishes what the product to sell will be
7) The main thread also periodically checks if the quantity ever underflows, it calls the getSellDetails() method to select a new product to sell.

8) The main thread periodically checks the sharedRequestBuffer, and if it finds it to be non-empty, it polls the queue, decrements the hop count and appends its own id into the message path. It then generates the necessary number of client threads (= no. of neighbors), each configured to the specific server port for a neighbor, and the respective client thread performs the lookup function (floods the request) and is destroyed immediately.
9) Similarly, the main thread periodically checks the sharedReplyBuffer, and if it finds it to be non-empty, it polls the queue, identifies the last entry in its message path as the immediate destination, removes this id from the path and spins a client thread just to propagate this request to the immediate neighbor and is destroyed
10) Similarly, the main thread periodically checks the sharedTransactionBuffer, and if it finds it to be non-empty, it polls the queue, identifies the id of the peer that the direct transaction connection needs to be made with, and finds the non-neighbor port number for that id and spins a client thread to connect to that port in order to complete the transaction. Before doing this, it checks the servicedRequests map to ensure that this transaction has not been conducted before
11) At the end of execution, just before exiting the program, the main thread prints the average response time per lookup request for that node by dumping the contents of the responseTImeMap onto the file and also calculates the average response time across all requests

Client thread in each node:

1) The client thread implements the P2PInterface - which employs the lookup(), reply() and buy() methods
2) On initialization, the client thread establishes connection with the server port and corresponding host name provided to it
3) Depending on the type of the message(lookup/ reply/ buy) the client serializes the message and executes the appropriate method and stops the thread. For the buy requests it also inserts the message into servicedRequests

Server thread in each node:

1) On initialization, the server thread starts listening on the port it has been assigned.
2) When the server thread accepts a connection on its port, it spins a new thread to handle that request and continues listening on the port. This allows for handling of concurrent requests
3) The server first checks if the message is present in the request history. If yes, and it is a “reply” meant for this peer itself, then the server notes the timestamp for this reply to one of its earlier lookup requests, and adds it to the list of response times for this request, and then the message is discarded
4) If the message is a “lookup request” with an expired hop count, the message is discarded

5) If the message has not been encountered before, and it is a “lookup request”, and the server thread happens to be a seller, it checks if it is selling the product mentioned in the request. If yes, it creates a new reply message and pushes it into the sharedReplyBuffer
6) If it is not a seller or does not possess the product, it pushes the message into the sharedRequestBuffer to be forwarded
7) If the message is of type “reply” the server thread first checks if the reply is meant for this peer or some other peer. If the reply’s sourcePeerId matches this peer, then the server creates a new message meant for the direct transaction and pushes it into the sharedTransactionBuffer. If meant for someone else, it pushes it into the sharedReplyBuffer, also adding its timestamp into the response time map
8) If the message is of type ”transaction” the server checks if it has sufficient quantity of the product to be sold. If yes, it decrements the quantity and returns an acknowledgement to the client, at which point the transaction is complete and product is considered sold
9) The server then adds this message to the requestHistory

The system is designed to asynchronously generate new Buy requests independent of previously completed or executed requests. As such, the system is robust to failed transactions as well as messages lost in transmission and execution is never stopped for those reasons.

Design tradeoffs:

One of the design tradeoffs we implemented is that we are maintaining a single server thread for each node that constantly listens to a port, and spins up threads to handle each request. As a result, we do achieve a degree of concurrent execution where each thread could be in a different stage of its execution simultaneously, but at a given instance we can have one connection being made to the server port. We had initially designed it such that each node would have a dedicated server thread for each of its neighbors as well as one for direct connections with non-neighbors. This allowed for a great degree of parallelism in terms of servicing requests, but we ran into issues executing this on EDLAB since it appears that EDLAB placed a cap on the number of threads that can be spun on each node (manifesting as an OutOfMemory error). As a result of this limitation, we could not have multiple server threads actively listening to ports all the time, and decided to spin and destroy server threads (which happens almost instantaneously) as required to reduce the number of threads being generated per node

Secondly, due to the same EDLAB memory issue, we had to limit the number of nodes running on each EDLAB machine to 2. As a result, we had to design the test cases such that no machine is handling more than 2 nodes at a given time, and all nodes are distributed across 4 machines. This allows for a more distributed approach to the system but limits the number of nodes we can have in our network to 8 for the test cases

Another trade off we are implementing is that we are creating a new set of threads for each flood operation and destroying them immediately after. This is more efficient than implementing a client thread pool as the workload is handled dynamically and threads aren't running when not required (when the sharedRequestBuffer is empty) However, in this scenario the number of threads spun are not bound, as a result we could have a very large number of threads being spun simultaneously if the incoming traffic ever increases dramatically

Potential improvements/extensions:

One potential improvement/extension that we could make to the system would be to allow peers to dynamically change their locations from one machine to another potentially for load balancing purposes. This could be done by implementing ways for peers to query other peers to find out how many other nodes are running on their respective locations (since each node has access to this information through the config file). Depending on these responses, the peer could send a connection request to a node on a machine that it discovers to be underutilized, and call a script to spin a new node on that machine with information about its existing configurations, and relay this information to all the other peers so they can update their config files as well. In this way a node could successfully migrate between locations with appropriate updates being flooded to all the nodes in the network so as not to break the workflow and achieve load balancing on different locations

How to run:

This system requires sufficient resources in terms of available memory, and cpu power. Please ensure these requirements are met on the testing platform. This program should ideally be run in isolation to ensure availability of specified ports for each node on all the machines in use

1) Ssh into EDLAB machines 1,2, 3 and 7
2) Clone the git repository: https://github.com/ds-umass/lab-1-rao-gupta.git
3) Navigate to the /test directory to find the test scripts.
4) The README file within the /test directory would include the details around the specific scripts that correspond to each test case and how they are to be executed
5) Each test script can be executed by running ./file-name.sh for example. This would compile and run the entire program for the given test.
6) The program is set to run for a period of 2 minutes, after which all the generated processes are automatically terminated, and a set of output files should be generated within the same directory. Depending on the test that is executed, a different number of output files will be generated (one for each node in the topology that was executed)
7) The output files show the sequence of events that occurred with respect to a particular node and also include the time-based performance metrics for each node