Alok Kumar
Saurabh Bajaj

Lab 1: Asterix and the Bazaar design doc

Table of Contents

Problem Statement

Construct a p2p network such that all N peers form a connected network. You can use either a structured or unstructured P2P topology to construct the network. All peers should have no more than three direct neighbors, and should only communicate directly (make RPCs, RMIs, or use the sockets of) direct neighbors during the simulation. You should also ensure that the network is fully connected:

Once the network is formed, assign each peer a random role: fish seller, salt seller, boar seller, or buyer. Once the roles have been assigned, each buyer randomly picks an item and attempts to purchase it using certain interfaces specified below; it then waits a random amount of time, then picks another thing to buy, and so on. Each seller starts with n items (e.g., n boars) to sell; upon selling all n items, the seller picks another thing at random and becomes a seller of that item.

Our implementation

Our peer-to-peer network is basically an undirected connected graph as shown in Fig 1. Here each node of the graph represents a peer in our network (buyer, seller). At each run, our network is randomly generated as a connected undirected graph with a maximum of three neighbors for each node. Once generated the graph is saved as an adjacency list in a JSON file. To establish the connection among peers and allow remote procedure calls in our network we used **Pyro4.** Each peer in our network is basically a python object registered as a Pyro4 which allows each peer to have remote access to other peers. To manage the peers registered in the network we are using Pyro's **nameserver** which is a tool to help keep track of the objects in our network. Each object is first registered with Pyro daemon which returns a URI and then a mapping of peer_id and URI is registered onto the nameserver. This implementation provides a very fast (O(1)) and easy way to lookup for a peer's URI given a peer id. To simulate different peers as a different process in our local machine. Each peer is spawned as a sub-process from the main process in a separate directory. Once started each peer spawns

multiple worker threads and starts listening for any incoming requests or starts sending new requests. For sending the requests all peers first do a neighbor discovery. Neighbor discovery in our implementation is done once per run and neighbors of each peer are stored in a class variable for subsequent calls. This method of discovery is fast and removes the need for repetitive neighbor discovery at each subsequent call. Once each peer discovers their neighbors the Bazar starts to run (seller peers looking for any incoming request to sell their assigned item and buyer processes flooding the network with the requests for items they want to buy). Once a match is found, the buyer enters into a direct transaction with the seller and completes the purchase.

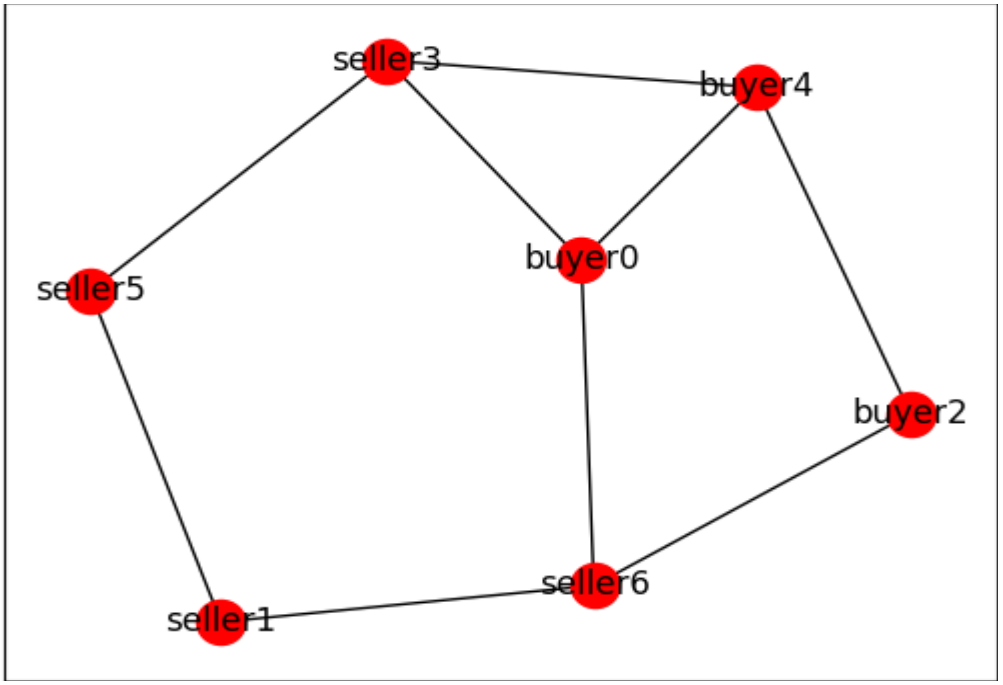

Fig 1: A sample-generated peer-to-peer network with 7 nodes

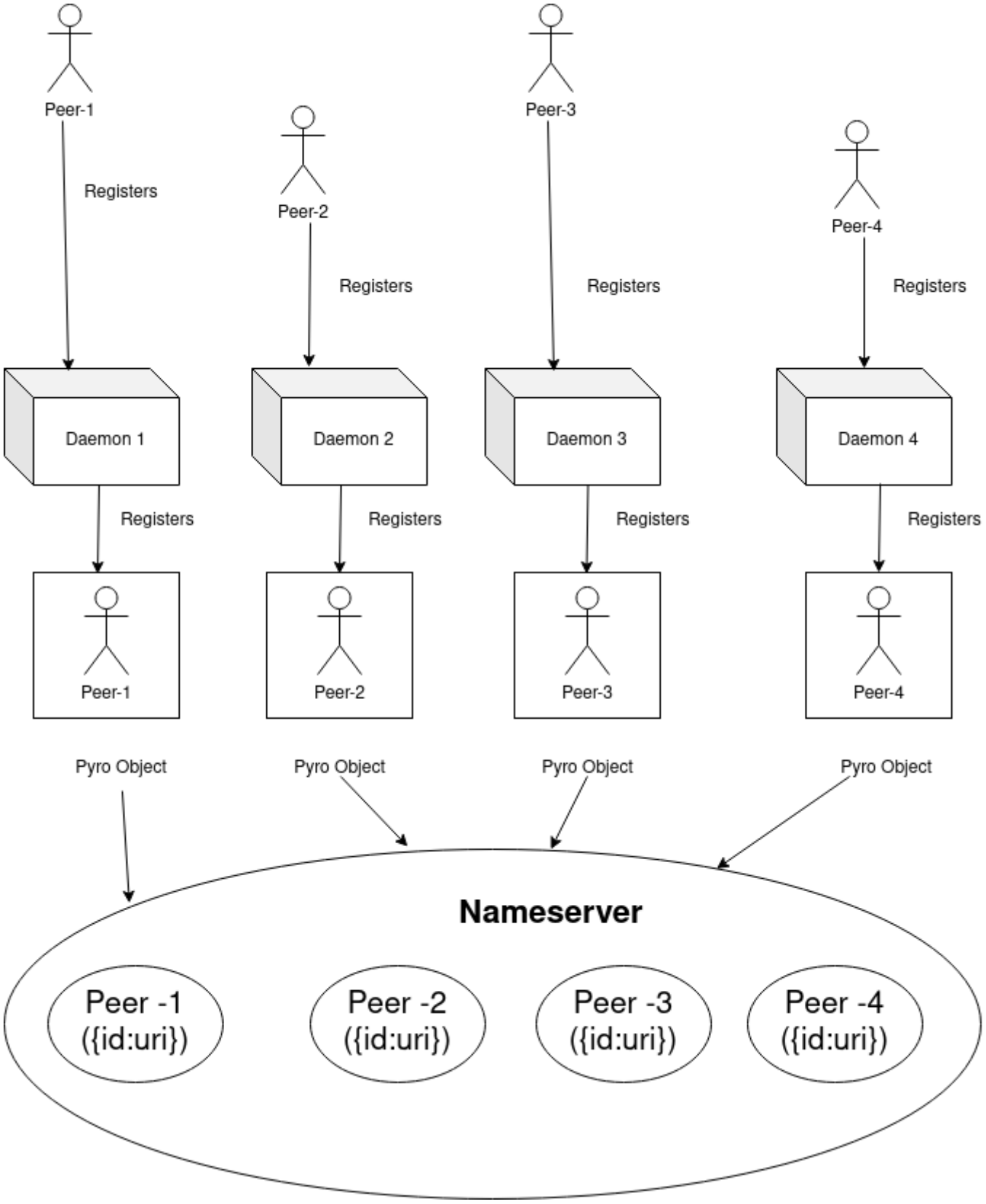

Fig 2: Registration process of each peer as a Pyro object in the nameserver

Control Flow logic

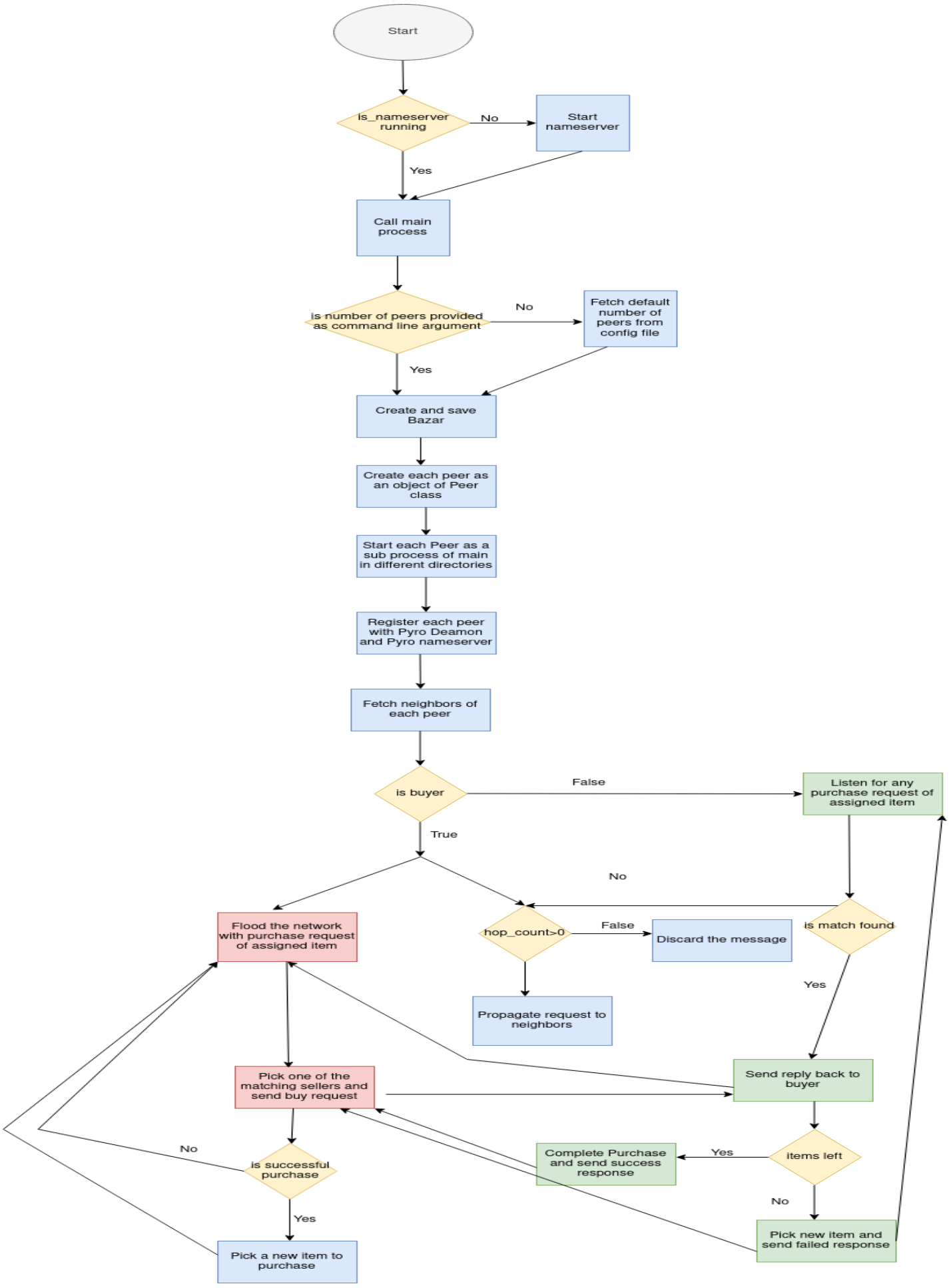

Fig 3: Control flow logic of the application

How the system works

The overall control flow of the system is represented in Fig 3 and is described below.

1. The application is started by running a single bash script file.
2. The first step is to start a nameserver if one is not already running.
3. Remove any dead peers from previous runs from the nameserver
4. Start the main process
5. Check if the number of peers is provided as a command line argument otherwise pick the number of peers from the config file.
6. Generate a connected graph network with a number of nodes equal to the number of peers.
7. Create peer objects and initialize them with mandatory parameters like role, items, etc
8. Start each peer as a separate sub-process of the main in different directories.
9. Register each peer as a pyro object over the pyro demon
10. Register each pyro object to the nameserver in form of {“id”: URI}
11. Start the daemon loop on multiple channels and listen to any incoming requests
12. Perform neighbor discovery and store all connected neighbors in a variable
13. If buyer flood the network with purchase requests of assigned item
14. If the seller listens to any incoming purchase request
15. Each buyer will forward any incoming request from a neighbor to its neighbors if the hop count is not zero else discard the message
16. Each seller will forward any incoming request from a neighbor to its neighbors if the request is not for the item it is selling
17. If the request is for the item a particular seller is selling then respond back to the buyer over the look-up path.
18. If a buyer receives a request for an item from the seller they will pick one of the sellers at random and enter into a transaction with the seller.
19. If the transaction is successful buyer will pick another item to buy or else will issue a new lookup for the same item.
20. The seller will pick a new item if stock of the previous item exhaust.

The design choice to handle concurrency

To make sure that each peer can listen and produce multiple concurrent requests we have spawned multiple worker threads inside each peer process. To spawn the thread we have used **threadPoolExecutor** of python with maximum workers set to 20. To make sure that any write operation is thread-safe we have added a thread lock for every write operation. This makes sure that at any moment only one thread has the access to the critical section.

Functional Documentation

1. **main() -** Entry point of the program. Responsible for spawning sub-processes (peers).
2. **create_bazar(peer_list, show_bazar=true)** - Takes a list of peer ids as an argument and calls the generate_graph function to create the network. It also has the capability to plot the network if the show_plot argument is set to true.
3. **generate_graph(node, edge) -** Takes a number of nodes and number of edges to connect the nodes as argument and generates a random undirected graph using **Networkx** graph generator. To guarantee that the graph is connected and has a maximum of three neighbors for each node we have added some wrapper code around the generated graph
4. **get_hop_count(edges) -** This function returns the hop count for the network. It finds the maximum path length among all possible paths. **hop_count** is set as a value lower than the maximum path length
5. **class Peer(Process) -** A child class of the process class that acts as the base template for generating peers. Each peer is associated as a buyer on seller based on **self.role.**
6. **get_all_neighors(self,peer_id) -** Returns all neighbors of a particular peer in the form of a dictionary of {“peer_id”:URI}.
7. **run(self)** - Implements the run method of the Process class. It is the entry point of each peer. The run method is responsible for registering each peer to the pyro daemon and pyro nameserver. Neighbor fetch and lookup are also done in the run itself.
8. **lookup(self, product_name, hop_count, search_path) -** This method is responsible for flooding the lookup request of a peer. If a match is found then the seller sends a reply back to the buyer on search_path otherwise request is propagated further and the search path is extended accordingly.
9. **reply(self, item,id_list) -** Once a match is found reply method is propagated seller_id to the buyer along the reply path.
10. **buy(self,buyer_id) -** Out of all the matches that buyer gets it picks one of the sellers and calls the buy method to enter into a direct transaction with the seller. If a seller still has the item left then the transaction is completed and a success acknowledgment is sent back to the buyer. Otherwise, the transaction has been deemed a failure, and acknowledgment for the same is received by the buyer.

Challenges

In the course of designing and implementing the problem statement, we faced a number of challenges. A few worth mentioning are :

1. Initially, we picked Pyro5 as our library of choice as it is the newer version of Pyro although we ran into the issue of threads not being able to take control of the proxy of another thread as proxies are not being shared across threads in Pyro5 implementation. The official resolution for this is to transfer control of the proxy from one thread to another. However, we failed to make it work even after multiple trials.
2. For finding the hop count we were required to find the longest path in the graph among all possible nodes. This problem in itself is a problem of NP-hard class and hence unsolvable in polynomial time. Due to this, our algorithm was taking a lot of time to calculate the longest path length if the number of nodes in the network become large say 20. To resolve this we set a fixed hop_count if a number of nodes exceed a certain threshold.

Future Scope

For now, our network is working fine if peers are deployed on a local server(localhost) machine. It will also work fine on remote servers as long as all the machines on which peers are deployed are under the same

remote server. However to make the system capable of running even when the peers are deployed on different remote servers we need to make some changes in our current implementation of neighbor discovery.