CS 677 Lab 1 Design Report

Dan Cline, Kyle Stevens, Tian Xia *UMass Amherst, Amherst College*

This document provides information of how our peer system was designed, the trade-offs involved with our implementation and future imporvements that may be made on the system.

Overview of the System

Our implementation of the system made use of `golang` and the default go `rpc` package to allow remote procedure calls between peers. We choose Go mainly for two reasons. In the first place, Go is more efficient than Java since it is directly compiled to machine code whereas Java has to be reduced to bytecode by JVM before it can be compiled to machine code, and it also has a a better garbage collection mechanism. More importantly, Go is designed for simplicity and scalability, especially in its multithreading abilities which smoothly handles concurrent processes with its `goroutines`. The amount of concurrency involved in this project makes Go, in our opinion, the most suitable language for this project.

Here we give a brief bird eye view of the system - the node network could either be randomly generated or specified through a network configuration file (templates can be found in the `test` directory). When it is randomly generated, the system takes in an arbitrary `N` and `k` in a configuration file where `N` is the number of nodes and `k` is the maximum number of neighbors each peer can have. It then forms a structured network and spawns a configuration file for each node to run on. Peers are randomly initialized to be one of the four roles: buyer, seller, both or none. The desired good and inventory, and the amount for each is also randomly initialized for each peer. Buyers then continuously send look-up requests to their neighbors for their targeted product, while sellers listens for requests, responds and sells products.

In terms of specific implementation, we followed the component structure and interfaces as given in the lab specification. Below, we describe in details each component and how each component accomplishes its intended functions, as well as additional features.

- Peers: Each peer runs on its own binary and takes in an initialization configuration file to start the process. The initialization configuration file for each node is generated by a go program `cmd/generatenodes`. Neighbor assignment in our system is static. Once the network is formed, the peers' neighbors and their addresses cannot change. Each peer configuration file is then copied to the host address specified in its configuration file along with the source code using `scp` and then booted. On start, peers would wait 5 seconds for all the peers to be on the network before making any RPC requests.

- Network Generation: `cmd/generatenodes` generates a network using `includeEdges` and `excludeEdges` in a network configuration YAML. The `includeEdges` lists the edges to be connected and the `excludeEdges` lists the edges to be excluded. In case of random generation, the network randomly assigns edges to be connected based on `N` and `k` but assures that each node has at least one neighbor.

- `CreateNodeFromConfigFile(configFile)`: The function reads in YAML configuration file, unmarshalls it and provides the peer details about it self (host address, port, roles, hopcount etc.) and also about the network (neighbor node IDs and their addresses). If the current node is a seller, this function is responsible for randomly picking an available item from the inventory.

- `init()`: The function sends the buyer into a buyer loop and does nothing for the seller. The buyer loop initiates look-up requests continuously and waits for all requests to come back before invoking a buy request.
- `lookupProduct(routeList, productName, hopcount)`: The look-up function is initiated by buyers to flood its neighbors and ask about the desired product. To allow the reply interface to backtrack a response through the network once a seller is found, the function also keeps track of a growing list of peers that receives the look-up request. Any node who is not a seller with the desired product adds its `id` and `hostname:port` on the route list, decrements the hop count and forwards the request to neighbors with a `goroutine rpc` call provided that the neighbor is not on the route list. If the hop count is 0, the message is discarded. If a matching seller receives the request, it calls the `reply` interface.
- `reply(routeList, sellerID)` Replies are initiated when one of the nodes on the look-up path happens to be a seller and has the desired product. The seller then uses the passed-along route list to send a `goroutine rpc` call to backtrack to where the request came from. If the length of the route list shrinks to 0 we know that we have reached the buyer and the propagation stops. The buyer then adds the `sellerID` to a go `channel` for it to pick after it has waited a specified amount of time to collect all the potential sellers.
- `buy(sellerID)`: The `buy` interface is only invoked when the buyer has a list of sellers that has the desired good. If there are multiple sellers who are able to provide the same good, then the buyer picks randomly a seller to buy from by calling the `sell` interface on the seller end. The seller can receive multiple buy requests at the same time so we implemented a lock `Mutex` on the amount of a good a seller has, so that the item inventory can only be decremented one at a time. If the flag `unlimited` is set to true in the seller node configuration, then the seller automatically restocks the supply of a certain good after the inventory hits 0. If the current item is sold old and the `unlimited` flag is set to false, the seller randomly picks an item from the inventory to sell. After the transaction is complete, the list is cleared and buyer spawns another random request.

Design Trade-offs

We used a structured peer network that does not change throughout. Advantages of this system is that we do not have to implement some kind of handshake interface or server daemon for peers to randomly pick its neighbors on joining the network. We can also set the network topology as we wish to test the system. One disadvantage of this system, however, is that if one of the nodes leaves the network, the whole network could be no longer connected if the node is a cut point.

To allow the replies to backtrack the look-up path, we chose to pass along a route list to keep track of the path. The upside of this implementation is its simplicity. However, this implementation can become very expensive if the network becomes very large since the route list that each RPC call passes can expand dramatically. This could cause a great deal of traffic in the system as well as slowing down the processing speed of each node. A better way to do this is send along instead a `uuid` with each buy request while letting each node keep track of where each `uuid` buy request came from. When the seller sends a reply, the node simply looks for the node associated with the `uuid`.

Future Improvements

Because of time constraints, there are several features that we did not implement but would like to have included:

- The current deployment system uses a bash script to `scp` the configuration files and source code to each target server. The bash script is also responsible for installing `golang` dependencies on each server. The process is rather cumbersome and the system will be cleaner and more efficient if in the future we wrapped everything in a docker container.
- We can implement a way for the network to handle nodes leaving the network. A way to do this is to have each node keep the adjacency information of the entire network. Each node also keeps track the heartbeat of its neighbors. If one of the neighbors cannot be connected, the node connects itself the the neighbors of that neighbor, so to prevent the network from becoming disjointed.

How to Run

To run all fo the designed tests, navigate to the `test` folder and

```
# for local testing
$ bash runall.sh

# for remote resting
$ bash testallremote.sh
```

To run specified tests locally, navigate to the `test` folder and use the bash script `runtest.sh` to run the desired network configuration. The user can also create their own `.yml` network configuration files based on the template give in the `test` directory.

```
$ bash runtest.sh <config>.yml
```

To run specified tests remotely, navigate to the `run` folder and use the bash script `run.sh` to run the desired network configuration.

```
$ bash run.sh <config>.yml
```

To run the system with arbitrary `N` and `k` parameters, create a `.yml` file like the one below and use `run.sh` to run it remotely or `runtest.sh` to run it locally.

```
K: 7
N: 50
maxHops: 7
outputDir: test8nodes
```

To terminate the process on all machines, send `ctrl+c` to the script and it will terminate all running pids of node instances on each remote machine.

If the program is run locally output logs for each node will be dumped in the working directory. If the program is run remotely, logs for each node is dumped on the machine it is running on. Each time a new tests run the old logs are cleared. `ssh` into different machines to view the output logs.