CSE434 Socket Project Design

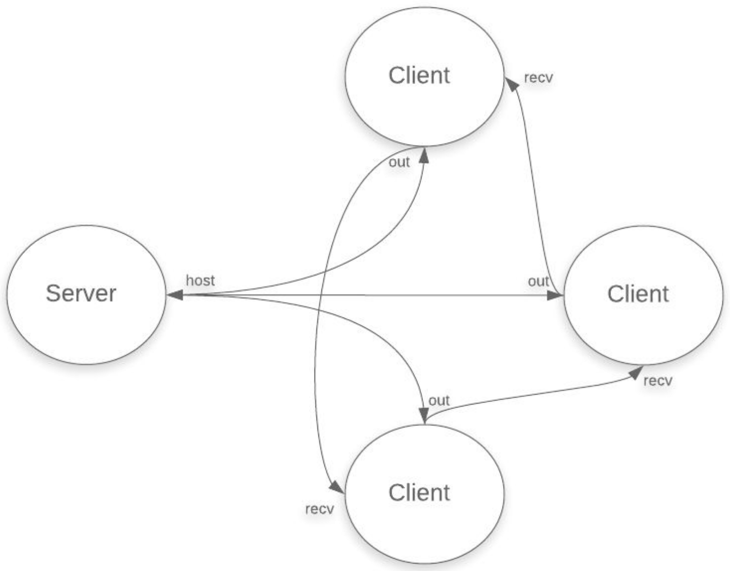
Kevin Shannon - 1213551857

Preface

The goal starting this project was to implement an application in which processes communicate using sockets to build a distributed hash table and then process queries using it. With the flexibility of languages available, I chose to write my application in python. There are two main scripts, ‘server.py’ and ‘client.py’. As their respective names imply one is for running server processes and the other for client processes. Each has its own argparser, for more information on running these scripts refer to ‘--help’ option or the USAGE file. The main goals in my design choices were to maximize ease of implementation, readability of the code, and user experience.

DHT Structure

The Structure of DHT is the foundation of the application and will dictate how the processes will be able to communicate with each other. Too many ports is unruly and can get confusing. For simplicity each client will ever only have two ports and the server only uses one port. One port from each client is used for sending data. Whether it is to the server or another client the client will use this port to send it. This port is also responsible for receiving the responses from requests it sends on this port. The other port is opened when the user has registered to that port. A new thread is started and the client will listen on that port for the rest of its duration. This port is used for receiving data from other clients.



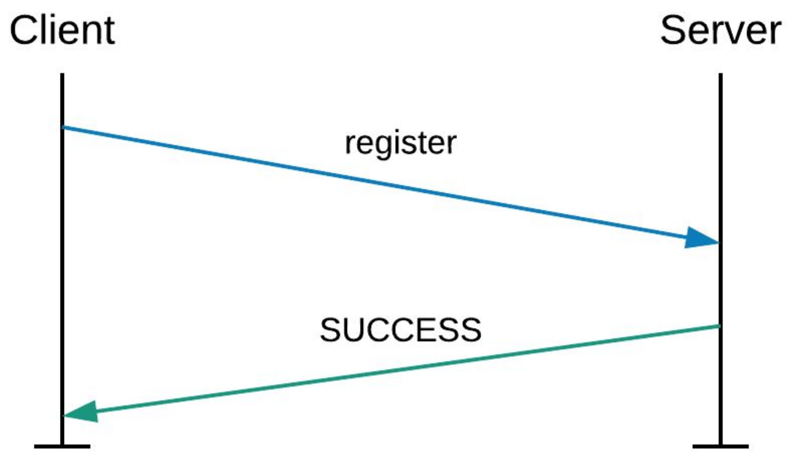
Above is a basic illustration showing the connectivity of a DHT of size 3. However, the network is not limited to this configuration, an address can be carried over the network. In the case of query-dht, the user's address who issued the query is passed along the ring and when the entry is found the client that found it will directly send the response directly back to the user.

Command Design

For each command I wanted the user to only have to specify what is absolutely necessary. Subsequently there are less chances for the user to make a mistake and the user can quickly interface with the application.

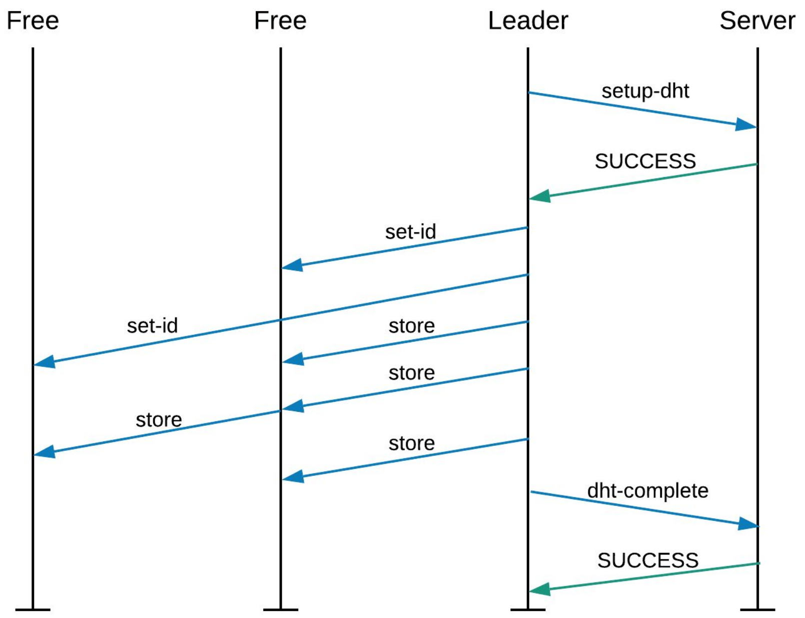
- ### register <user-name> <port>

  Since the client is told the IP address of the server when the process is spawned and the server knows where the message is coming from, the only needed parameters for register are user-name and port to listen on.



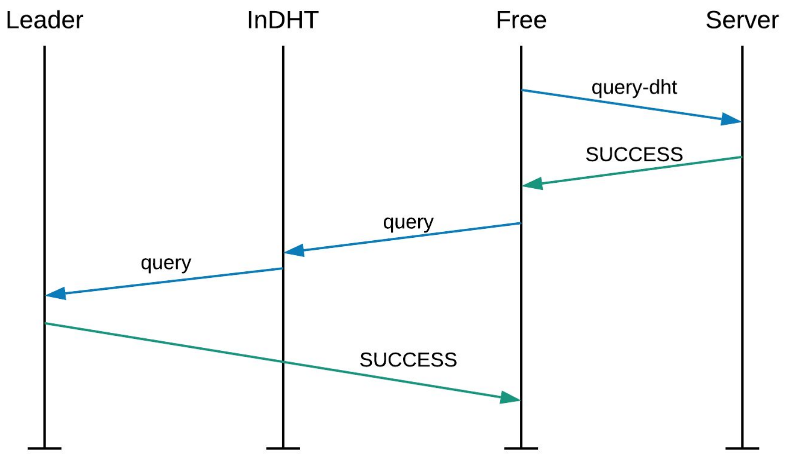
- setup-dht <n>

  Similarly, the server can tell which user is sending the request so the user needs only to specify the size of DHT to construct. On completion a complete-dht message is automatically sent to the server so it can start accepting new commands.



- query-dht <long-name>

  The command parser is smart so when it sees a query-dht it will take all subsequent input as part of the long-name regardless of whitespace.



Data Structures

For both the client and server programs, many liberties were given on what data structures could be used to accomplish our goal of creating a DHT. I went with an object oriented approach, choosing to make the server and Client their own classes. This simplified keeping track of persistent state variables. Although this could've been accomplished by just using global variables, my approach is cleaner and more scalable.

State information must be kept on the server, my implementation of how a user is represented is slightly different from a 3-tuple of user-name ip and port. A user in my application is namedtuple of user-name out-addr and recv-addr, each addr being a 2-tuple of IPv4 and port itself. Although the IP address is being duplicated, it makes it easier to set up this way because now each attribute is unique to each user and we can directly address each of a client's two ports by one name.

The hash table is a module that I wrote myself. I wanted a basic hash table data structure that uses open addressing linear probing and allowed the size to be fixed. The table part of the hash table is actually just a list of table entry objects. Each entry has a potential record, key, and tombstone. Tombstones are markers that signify where there have been entries removed from the hash table so that lookups can continue.

Message Format

After many iterations the final message format I landed on was a pickled 'types.SimpleNamespace'. By no means the most efficient format but it filled all of my personal criteria. First, pickling our message allows us to send almost anything over the line with no extra work, and there is no hassle decoding just a simple dump will bring it back to a python object. The reason for using a SimpleNamespace is again flexibility and the readability of dot notation. The standard format for a request message is a SimpleNamespace with two attributes: command and args. command is a string of just the name of the command. The server can always rely on being able to parse what command is being sent. args is another SimpleNamespace that contains the arguments pertinent to that command, these are dynamic so all commands can use this same format but with their command specific args.

After our request has been sent there will come a response. The format for a response message is similar to that of the request, again a pickled SimpleNamespace but now the two attributes are: status and body. The status will be a string either 'SUCCESS' or 'FAILURE'. body is whatever data needs to be returned, it can be almost anything or None.

Some messages like store and set-id and query don’t always have a response. Besides dropped packets or a disconnect there are no clear ways these requests can fail. This also helps avoid the problem of sending the user 300 unhelpful store message SUCCESSes.

Notes

StatsCountry.csv must be utf-8 encoded for the application to correctly parse it. Doing so seemed to improve what looked like some special characters getting corrupted but there are still a few cases of some strange looking characters in the csv. If you would like to change the stats file used in the application you can change it with an --stat_file in client.py.