UNSW COMP3331 20T1 DHT Assignment Report

Author: Andrew Wong (z5206677) | GitHub: featherbear/UNSW-COMP3331-assign

Overview

The assignment involved designing a networked P2P program that implements a basic Distributed Hash Table.
This involves Peer Initialisation, Peer Pinging, Peer Management, Data Insertion and Data Retrieval.

To facilitate the development of my approach to the assignment, I have created several helper files and tools, which have been explained below. The assignment specifications were unclear and vague in areas - at times contradicting itself. This made the implementation of the assignment confusing.

Breakdown

`runner.py`

The `runner.py` file is a wrapper around the Peer initialisation functionality, spawning peers in a programmatic manner.

Example when Peer 15 joins

```
Launching nodes:
  (Peer[2]->4->5)
  (Peer[4]->5->8)
  (Peer[5]->8->9)
  (Peer[8]->9->14)
  (Peer[9]->14->19)
  (Peer[14]->19->2)
  (Peer[19]->2->4)

[4] Peer 15 Join request forwarded to my successor
[5] Peer 15 Join request forwarded to my successor
[8] Peer 15 Join request forwarded to my successor
[9] Peer 15 Join request forwarded to my successor
[14] > Peer 15 Join request received
[14] > My first successor is Peer 15
[9] > Successor Change request received
[14] > My second successor is Peer 19
[9] > My new first successor is Peer 14
[9] > My new second successor is Peer 15
```

Its 'Reporter mode' (when launched with `runner.py -r`) periodically checks for the predecessors and successors of each peer, to make it easy to identify the relationships between peers; especially as peers come online and offline.

Example when Peer 15 joins

```
==================================
Peer  2 - 15 > 19 > [ 2] >  4 >  5
Peer  4 - 19 >  2 > [ 4] >  5 >  8
Peer  5 -  2 >  4 > [ 5] >  8 >  9
Peer  8 -  4 >  5 > [ 8] >  9 > 14
Peer  9 -  5 >  8 > [ 9] > 14 > 15
Peer 14 -  8 >  9 > [14] > 15 > 19
Peer 19 - 14 > 15 > [19] >  2 >  4

Unmonitored Peers (1): 15
==================================
```

`lib/argParser.py`

The `argParser.py` file contains functionality to validate the syntax and format of the arguments used to launch the peer.

`lib/Peer.py`

The `Peer.py` file contains the DHT networking logic inside a class `Peer`.

It was designed as a class, with the Runner script in mind - So that nodes could be created and initialised programmatically rather than using the suggested bash script.

To facilitate concurrency, each Peer spawns three task threads that run in the background

TCP Server (Entrypoint: `__serverFn`)

The TCP server handles incoming connections as well as commands from other peers.
The majority of network command are handled in this thread.

As advised from the tutor, each command has its own connection, rather than sharing the connection. This could be a possible extension to the implementation, by implementing persistent TCP connections

UDP Ping Server (Entrypoint: `__pingServerFn`)

The ping server listens for ping requests, and replies to its clients with a listing of its own successors.

UDP Ping Client (Entrypoint: `__pingClientFn`)

The ping client regularly sends UDP packets to ping its two successors, and checks for the time it last received a reply.

The `__sendPing` method is exposed to the `Peer` class instance when this thread is spawned.

Peer Management

- When a peer wants to join, the request is sent to the known peer
- The known peer will pass the request around the network until the appropriate peer (immediate predecessor to the joining peer) accepts the request
- The appropriate peer will send an offer to the joining peer
- When a peer wants to leave, a quit message will be sent to both of the peer's predecessors
- The predecessors will update their successors from the data in the quit message
- When a peer abruptly leaves, after a timeout period (20-60 seconds), the reference to the exited peer is updated, and new successors are requested

Transmitting and Receiving Files

To send/receive files, the hash of the file (*modulo 256*) is determined, and the closest successor for that hash is located by passing the requestor and the filename around the network. In the case of requesting for a file, the sender will send the length of the file, a well as the contents; so the receiver knows how many bytes of incoming stream is for the file.

During data transfer, a structure is created in `_connectionsMetadata` to pass data between reads.

`lib/portUtils.py`

The `portUtils.py` file contains helper/utility functions to perform port-related operations

`lib/Reporter.py`

The `Reporter.py` file contains the `Reporter` class, which is used in the `runner.py` Reporter mode to inspect the state of each peer

Improvements

Reporter Mode

Read above at `runner.py`

Ready callback

A callback is available to be set during peer setup, which will be called when the peer is ready for activity. This allows external functions to asynchronously wait for the peer to connect before proceeding.

Verbose messages

Three flags `SHOW_PING_REQUEST`, `SHOW_PING_RESPONSE`, and `SHOW_CUSTOM_DEBUG` are used, which when enabled output extra messages to the console.

Step 6 - Data Insertion

To emulate the storage of a file; the peer which accepts the file, will create a file `XXXX.pdf` in its directory.

DHT Shortcuts

When DHT shortcuts are enabled, peers will be able to pass data around the network more efficiently, as they can skip transmitting data to unnecessary peers. For example, Peer 5 sends data around the network intended for Peer 7 - if Peer 5 is aware of Peer 6, but also of Peer 7 - it can skip passing the data to Peer 6.

The below code was written but **disabled in the code**, as it would generate an output that *does not match the assignment specifications*.

```
if hash > self.first_successor:
    self.___sendTCP(self.second_successor, f"store|{_filename}|{requestor}".encode())
else:
    self.___sendTCP(self.first_successor, f"store|{_filename}|{requestor}".encode())
```

Extensions

Persistent TCP Connections

Due to the nature of TCP connections, there is overhead in setting up and tearing down TCP connections. It may be worthwhile to implement persistent TCP connections between peers. However due to the spontaneous nature of peer to peer activity, it may be a waste of resources (sockets and file descriptors) if there is no activity over most connections.

File data length

If a file was 22222 bytes long, the currently implemented protocol would require 5 bytes to transmit this metadata. However in reality the number 22222 is a 2-byte number; and 3 bytes are wasted due to transmitting the length as ASCII characters.
We could use Python's struct packing and unpacking functions to encode a number into its bytes, as I have previously used in another project.

Security

The implementation currently does not implement any sort of security measures, such as the encryption of data sent over the network.

This will allow anyone to intercept data (such as files transmitted with the send command) that may be confidential in nature. Furthermore, join requests could be forged, allowing any peer to join the network.

A good extension would to implement encrypted communication between each node.