CS3331 Assignment 1

Implementation of Peer-to-Peer Network using Distributed Hash Table

By Aniket Chavan (z5265106)

A look at my code

The code is simplified into 1 file (main.py) implementing a class for the Peer and using classes within that to handle all necessary operations that a peer needs to be able to complete. All information is stored as an attribute of the main class and most of this is handled within the `__init__` function for the class.

Commands

All UDP and TCP commands begin after a `/` signifier which lets the code know that it's a command. In addition to this, all commands have a set structure where when split by spaces, the last element of the resulting array is the id of which peer sent the message while the second last element is the id of which peer the message is intended for.

E.g. /command [--optional info1] [--optional info2] [destination] [source]

Functions

`__init__`

If the peer is part of the network initialisation, it is assigned a second successor, otherwise it is a joining peer and the second successor argument is instead used as the interval value.

`updateSucc` & `updatePred`

Both functions updated the respective attributes of the Peer class to the given values, only `updatePred` had to figure out which was first and second predecessor.
E.g. Second Predecessor -> First Predecessor -> Current Peer -> First Successor -> Second Successor

`serverSetup`

This function creates both the TCP and UDP server instances which are needed for both peers that are initialising the network and peers that are joining. It also starts the servers as they are initialised thread processes.

`activatePeer`

This creates the PingClient and InputInfo instances and starts both of those Thread processes as well.

`joinPeer`

When a peer is joining, its PingClient and InputInfo can't be established until it is accepted into the network and has successors set up. This function sends through the join request to find a place for the new peer.

`fileStorer` & `fileFinder`

These functions handle where a file to be stored should go and finding if the file is in the current peer, else forwarding the request to store/retrieve the specified files.

`sendUDPmessage` & `sendTCPmessage`

These functions create sockets for `SOCK_DGRAM` and `SOCK_STREAM` respectively and use them to send messages.

Classes

All classes are initialised as Thread instances that allows them to run on separate processes. They also all have the Peer class passed in as `self` in order to access all values of the Peer.

`UDPServer`

Within its `run` function it handles receiving UDP messages, responding to interval pings and also maintaining who the predecessors of the peer are.

Commands it deals with are: `/ping`, `/response`

`TCPServer`

This class handles receiving all TCP messages for which there are a lot of commands. It parses the received message and has a switch case implementation for dealing with the many commands.

Commands it deals with are: `/join`, `/decline`, `/accept`, `/successor`, `/leave`, `/store`, `/request` and `/deliver`

`PingClient`

This class handles sending the interval UDP ping requests to the peer's successors.

`InputInfo`

This class finally handles user input for things such as `Quit`, `Store` and `Request`.

`Other`

The signal libraries are used to allow `ctr+C` exit as well as for the `Quit` input and closing the terminal.

Improvements

This code does not have abrupt leaving handled or any actual handling of files for data retrieval. I had the pressure of my 1531 project alongside which gave me significantly less time on this assignment. These assignments was both painful but also insightful as I found 3331 and 1531 touch two different sides of the same things: networks and server handling.

Abrupt leaving could be handled by setting up a timer, possibly another Thread class which has the sole purpose of checking every interval+x seconds for whether there has been a new ping from a predecessor.

File handling I imagine would have been a simple copy of the file to signify that it was properly identified and renaming it to `_filenum` to show the file was correctly identified. I completed all other aspects of retrieval involving identifying which peer stores the file and creating a direct connection between the requesting peer and the peer who stores it.

I included a few catches for things like a new Peer joining with an id that already exists and a Peer requesting a file that it itself stores, however given more time I would also create more of these catches. One specifically being for the data insertion where the specific Peer given the hash does not exist. Currently it only stores the file if a peer with id equal to the hash exists else it will just forward forever. I would implement it checking with preds and succs to make a more informed choice and storing it in the closest successor.

Code Use

https://github.com/WWEISONG/Projects/blob/master/20T1ass_3331/simple_example.py

https://webcms3.cse.unsw.edu.au/static/uploads/course/COMP3331/19T3/dd412842f430a4f58dc524f00a75b6d5c3b3e21d6a339374c5ffaf5f2fa55868/UDPServer3.py

https://webcms3.cse.unsw.edu.au/static/uploads/course/COMP3331/19T3/186b270da2af3f60002398e12e664620428592f910cd7f543a749c20aebc0e49/UDPClient3.py

I only borrowed the base of my tutor's starting code which helped me understand one implementation of the overall structure. While I took his ideas for implementing the classes within classes and class methods, I used the code in a way that I personally found more intuitive including passing the Peer instance as `self` through to each of the subclass instances. The largest help in this code was his use of creating Thread initialised classes. I otherwise used the course given code for multi-threaded servers to understand how they work.