Peer-to-peer networking

By **Tuan Ho** - *(github.com/ttuanho)*

Introduction

There are five services within this P2P network:

1. Data insertion: An external entity can request any peer to store a new data record into the distributed database implemented as the DHT.
2. Data retrieval: An external entity can request any peer to retrieve a data record from the DHT.
3. Peer joining: A new peer can approach any of the existing peers to join the DHT.
4. Peer departure (graceful): A peer can gracefully leave the DHT by announcing its departure to other relevant peers before shutting down.
5. Peer departure (abrupt): Peers can depart abruptly, e.g., by "killing" a peer process using CTRL-C command.

Other info:

- See [*specifications*] for more.
- Source code: see the main file [`p2p.java`]
- Testing cases
- Video of testing demo: [https://youtu.be/CUpft2ffg-s]
- Video demo of testing against rubric fully : [https://youtu.be/VgxeSM-Tl5w]
- This is a part of COMP3331 assignment and published under those premises
  - the Lecturer In Charge advises, and in fact, encourage to showcase this project
  - the course assignment is ***renewed every term***
  - this is published for the purpose of showcasing only

Running the files

1. Compile the java code to bytcodes via command `javac p2p.java`.
2. Run the server
   - `java p2p [command] [option]....`
   - The running scripts for initial ping successors can be found in `demo.sh` with `pingInterval` of 50 seconds.
   - For quick testing of periodical ping and abrupt exit (`pingInterval` of 10 seconds), the script `demo2.sh` can be used.

- If getting error `Address already in use`, the scripts `kill.sh` can be used.

General description of the implementation

There're 3 main co-current threads / tasks / future (which is always newly created whenever the task / future is done):

- `UDPServerProcessor` class acts as UDPServer, handling Ping messages from other client(s).
- `TCPServerProcessor` class acts as TCPServer, handling messages from other client when a new peer joins the network as well as transfering files or joining messages.
- `TerminalProcessor` class handling IO from terminal. Each time we hit [Enter] or new line, the thread processes the command and terminate.

Ping requests are sent to successors periodically. The time interval to check if the successors are still alive is set `pingInterval` $\pm$ 1 seconds that is given in the commands:

- `java p2p init [peerId] [firsSuccId] [secondSuccId] [pingInterval]`

or

- `java p2p join [peerId] [knownId] [pingInterval]`

Handling messages

A big part of communication between the peers is handling messsages based on fixed protocols and formats. These are expected request & reponse between the peers.

Function of the message	Format of the message	Protocol	Note(s)
Sending ping request	PeerId [`senderId`] : ping request	UDP	
Response to ping request	PeerId [`receiverId`] : ping response [`firstSuccId`] [`secondSuccId`]	UDP	
Sending store request to the suitable successor (as described below)	PeerId [`senderId`] : ping fileStore [`filename`]	UDP	
Sending file request to the suitable successor (as described below)	PeerId [`originalSenderId`] : ping fileStore [`filename`]	UDP	
Reponse to the original file requestor	PeerId [`senderId`] : ping fileRequestAccept	UDP	

Function of the message	Format of the message	Protocol	Note(s)
Broadcasting the peer leaving (to all suitable successing and precedent nodes)	Peer Id [senderId] : ping leave [`leavingPeerId`] [`leavingPeerFirstSuccId`] [`leavingPeerSecondSuccId`] [`leavingPeerThirdSuccId`]	UDP	Successing Id can be filled by successing node(s)
Broadcasting the peer joining (to all suitable successing and precedent nodes)	Sender [`senderId`] : [`joiningPeerId`] join [`joiningPeerFirstSuccId`] [`joiningPeerSecondSuccId`]	TCP	The fields [`joiningPeerFirstSuccId`] [`joiningPeerSecondSuccId`] are filled by successing nodes
Sending the list of file(s) that will be transfered	Sender [`senderId`] : [`hostingFilePeerId`] fileRequestAccept [`number of files`] [filename] [filename]...	TCP	
Transferring the file(s) requested		TCP	

Generalized description of the scenario when a peer is joining

Let $firstSucc$ and *secondSucc* be the functions that return the first and second successors of a node respectively. Suppose have a peer joining with id $joiningNodeId$ and it only knows the node $knownNodeId$.

Initially, peer $joiningNodeId$ will notify its known peer $knownNodeId$ that it's joining. The message would be "$joiningNodeId$ `join`". Then

Let $currNodeId$ be Id of the node that receives the messsage. Hence, initially, $currNodeId = knownNodeId$, then there're following cases:

- (1) If $joiningNodeId > secondSucc(currNodeId) > firstSucc(currNodeId)$,
  - if the node $currNodeId$ receives the message "$joiningNodeId$ `join`" it will pass this message to the node with id of $secondSucc(currNodeId)$
  - if the node $currNodeId$ receives the message "$joiningNodeId$ `join` $firstSucc$ $secondSuccId$" it will pass this message to the node with id of $firstSucc(currNodeId)$
- (2) If $secondSucc(currNodeId) > joiningNodeId > firstSucc(currNodeId) > currNodeId$ (the case where joinning peer should be in between the first and second successors), the node $currNodeId$ will

- notify $firstSucc(currNodeId)$ that peer $joiningNodeId$ is joining. The message is still "$joiningNodeId$ `join`".
  - update $secondSucc(currNodeId) = joiningNodeId$

- (3) If $firstSucc(currNodeId) > joiningNodeId > currNodeId$, the node $currNodeId$ will
  - notify the peer $joiningNodeId$ that his successors are now my previous successors. So node $currNodeId$ will send to peer $joiningNodeId$ with the message "$joiningNodeId$ `join` $firstSucc(currNodeId)$ $secondSucc(currNodeId)$"
  - send this same message "$joiningNodeId$ `join` $firstSucc(currNodeId)$ $secondSucc(currNodeId)$" to the $senderId$
  - update $secondSucc(currNodeId) = firstSucc(currNodeId)$
  - update $firstSucc(currNodeId) = joiningNodeId$
- (4) If $joiningNodeId > firstSucc(currNodeId) > currNodeId$ and $secondSucc(currNodeId) < firstSucc(currNodeId)$ (the case where $joiningNodeId$ will be the $maxId$ of the network and will be in between the first and second successor), do the same as (2).
- (5) If $firstSucc(currNodeId) > currNodeId > secondSucc(currNodeId)$ and $joiningNodeId < secondSucc(currNodeId)$ (the case where $joiningNodeId$ will be the $minId$ of the network and will be in between the first and second successor), do the same as (2).
- (6) If $joiningNodeId < firstSucc(currNodeId) < currNodeId$ (the case where $joiningNodeId$ will be the $minId$ of the network and will be in between $currNodeId$ and the first successor), do the same as (3).
- (7) If $joiningNodeId > currNodeId > secondSucc(currNodeId) > firstSucc(currNodeId)$ (the case where $joiningNodeId$ will be the $maxId$ of the network and will be in between $currNodeId$ and the first successor), do the same as (3).

Generalized description of the scenario when a peer is (known to be) leaving

In contradiction to joining the network, instead of using TCP, we now use UDP. Instead of trying to figure out the successors of the joining peer and update the successors of the precedents of the joining peer, we now try to update the the successors of the precedents of the leaving peer.

We consider a peer leaves the network if

- it notifies to one or more peer currently in the network that it's leaving
- it does not respond to ping request after `pingInterval`$\times 1.5 \pm 2$ seconds

If a peer with $leavingPeerId$ leaves gracefully,

- it passes its 2 successors' ids to its first successor. The message would look like: "`Sender [senderId] :` $leavingPeerId$ `leave` $firstSucc(leavingPeerId)$ $secondSucc(leavingPeerId)$"
- if a node ($nodeId$) receives a message as formated above and $leavingPeerId \neq firstSucc(nodeId)$ and $leavingPeerId \neq secondSucc(nodeId)$, it continues to pass this same message to $firstSucc(nodeId)$
- if a node ($nodeId$) receives a message as formated above and $leavingPeerId = secondSucc(nodeId)$,
  - update $secondSucc(nodeId) = firstSucc(leavingPeerId)$
  - continues to pass this same message to $firstSucc(nodeId)$
- if a node ($nodeId$) receives a message as formated above and $leavingPeerId = firstSucc(nodeId)$, then
  - update $firstSucc(nodeId) = firstSucc(leavingPeerId)$
  - update $secondSucc(nodeId) = secondSucc(leavingPeerId)$

In the case where a peer with $leavingPeerId$ leaves abruptly, after `pingInterval` $\times 1.5\pm$ 2 seconds, the two preceding nodes will presume $leavingPeerId$ has left the network, then

- the first precedent ($firstPrecedent$) of the $leavingPeerId$ (the one that has the $secondSucc(firstPrecedent) = leavingPeerId$) will wait 2 seconds until the second precedent figures out his successors
- the second precedent ($secondPrecedent$) of the $leavingPeerId$ (the one that has the $firstSucc(secondPrecedent) = leavingPeerId$) will send the UDP message "`Sender` $secondPrecedent$: $leavingPeerId$ leave $secondSucc(secondPrecedent)$" to its first successor
- the node receives the partially known message from the second precedent will send back the full message where it's the second successor of the leaving peer
- the second precedent nows receives and knows the 2 successors of the leaving peer and update its own successors

Generalized description of how to determine which successor to send Store and Request messages

Given a node / peer with and id $nodeId$ and its 2 successors have id of $firstSuccId$ and $secondSuccId$ respectively. If the node receive a UDP message from other node, it will also knows the id of the sender ($senderId$).

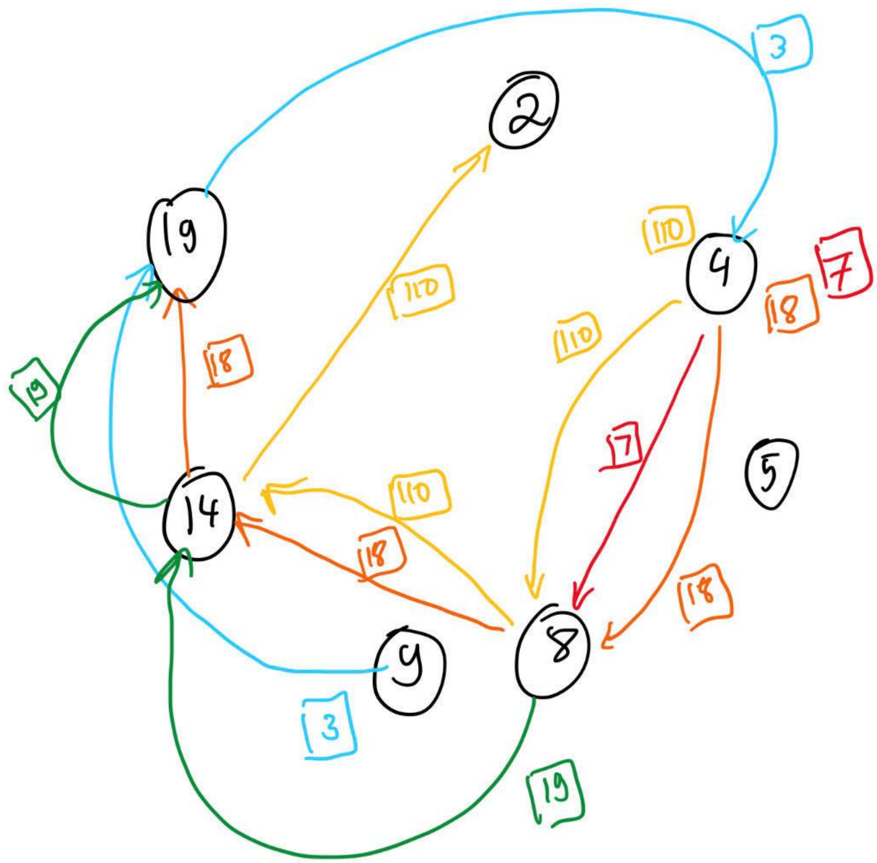
Let $fileName$ be the name of the files requested or to be stored and $fileId \equiv fileName \bmod 256$ and $receiverId$ be the id of the node that will receive the messages. So the node will store the file

- if $fileId = nodeId$
- $fileId < myId$ and $fileId > senderId$ (the case it's going back to the start of the circle)
- $fileId < myId$ and $myId > firstSuccId$ and $myId > secSuccId$ (the case the file is assigned at the end of the cirlce)

Otherwise,

- If $nodeId < fileId < firstSuccId < secondSuccId$, then $receiverId = firstSuccId$.
- If $firstSuccId < secondSuccId < nodeId < fileId$, then $receiverId = firstSuccId$ (the case 2 of the nodes' successors are at the beginning of the circle).
- If $secondSuccId < nodeId < fileId < firstSuccId$, then $receiverId = firstSuccId$.
- If $nodeId < firstSuccId < fileId < secondSuccId$, then $receiverId = secondSuccId$.
- If $secondSuccId < nodeId < firstSuccId < fileId$, then $receiverId = secondSuccId$.

This idea can be illustrated by the image below where the circles represent the peers or $nodeId$, the squares are $fileId$ (the modulus of the filename to 256) and the arrows are the corresponding routes that the messsages should be forwared:



From the implementation above, if Store $fileId$ request is enter from the terminal of $nodeId$, it's hard to know to if this node is responsible for the file and so it will send the message around the cirle. Based off the logic sequence above, if it receives the message again then the node will store the file. Otherwise,

- other node with smaller id would store the file if $fileId > nodeId$
- other node with higher id would store the file if $fileId < nodeId$

Other notes

- Some expected delays:
  - To avoid multiple UDP messages flooded to the one thread, some delays between sending ping messages of peers will occur; however, it should not be longer than 2 seconds. Co-current arrival of a TCP and UDP messages should cause no delay.
  - When a peer leavs gracefully, it takes a few seconds for the precedents

    to update their successors correctly

- File extention will be ingored so the peer must consider all files have the given 4 numbers (https://webcms3.cse.unsw.edu.au/COMP3331/20T1/forums/2758303) and (https://webcms3.cse.unsw.edu.au/COMP3331/20T1/forums/2760440)
  - The starting scripts `demo.sh` also creates 2 valid file `1234` and `1234.txt` with different contents. These 2 files will be transfered to the peer makes "`request 1234`"".
- **The file requested must be assigned to one peer**. Otherwise, the peer that is responsible for the stored file will send the message *"The file is not store here"* back to the original peer that sends the request.
- For easier testing of quiting case, look for the lines with keywords "`>> My succ`", uncomment out the debugging line `logger.info(...)`.

Assumptions

- There's no concurent request for the same file
- When departing (gracefully), a node will ***not*** send all its files to its immediate successor (https://webcms3.cse.unsw.edu.au/COMP3331/20T1/forums/2760482)
- There will be no updates for data insertion storage for when a peer joins or leaves (https://webcms3.cse.unsw.edu.au/COMP3331/20T1/forums/2760025)

Explanation of borrowed code

N/A

Extensions / Improvements

For debugging purpose, some extensions are available:

- Manually sending ping request from terminal with $currNodeId$. Format: `ping request [currNodeId] [destionationId]`
- List the files the the peer is storing via command `ls`.
- To notify other peers that this peer just joined the network. Type in the terminal: `tcp [knownPeerId] join`

Errors need debugging

- [ ] When typing `quit` in the terminal, the process is shut down but `SocketException` is thrown even all sockets are closed before.

Resources used

- http://www.java2s.com/Tutorials/Java/Java_Thread_How_to/Concurrent/index.htm

- https://docs.oracle.com/en/java/