COMP3331/9331 Computer Networks and Applications
Assignment for Term 1, 2020

Individual Assignment

Due: 5pm, 21 April 2020

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

[Updated Version: Released on 17 March 2020]

NOTE: There will be no further versions of the assignment specs released. From now on, any updates or clarifications will be posted on the Assignment Forum in WebCMS. You must keep an eye on this forum and use it for all discussions regarding this assignment.

Goal and Learning Objectives

The goal of this assignment is to gain hands-on experience in implementing the concept of peer-to-peer (P2P) networks. Instead of connecting geographically distributed peers over the Internet, *the scope of the assignment is confined to a single operating system with sockets or processes acting as independent peers*.

On completion of this assignment, students will:

1. Learn techniques for implementing peer-to-peer networks
2. Gain experience in implementing distributed hash tables (DHTs), the bedrocks for many distributed systems
3. Master socket programming

Background on P2P Networks and DHT

Week 3 lecture covers the necessary background on P2P networks. The lecture notes are supplemented by relevant pages from the 6th edition of the textbook as well as a scientific article, called CHORD, which provide more detailed coverage of DHT. You will implement the methods described in these documents.

Assignment Specifications

You will implement the following **five P2P services**:

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.

For insertion, retrieval, or joining services, you are required to display the route taken along the circular DHT to complete the service. The route is simply defined as the sequence of peers traversed from the originating peer to the final one.

Following guidelines apply for implementing these services:

- A peer is implemented as a *process* and represented as an *xterm*, which can be used as an interface to approach the peer. The *xterm* can also be used for the peer to display any internal messages.
- The DHT identity of a peer is drawn from the range [0,255]. This means that technically, the DHT can support a maximum of 256 peers to join the network. For tractability, however, actual testing will be conducted with a small number of peers.
- Data records are files stored in the same directory where the assignment codes are stored.
- Filenames are four-digit numbers such as 0000, 0159, 1890, etc. Filenames such as a912 and 32134 are invalid because the former contains a non-numeral character and the latter does not consist of exactly 4 numerals
- Hash function used to produce the key is given as *modulus(filename/256),* which results in a key space of [0,255] matching the DHT identity space. For example, the hash of file 2012 is 220.
- A file producing a hash of *n* is stored in the peer that is the closest successor of *n* in the *circular DHT.*

You should use client-server socket programming paradigm to implement all communications between the peers. For example, you can choose specific unique port numbers for each peer to listen for communications, which should be known by all peers. You can use TCP sockets for reliable transfer of data during data insertion and delivery and UDP sockets for regular status checking (e.g., regular pinging) between predecessor and successor peers. The Appendix at the end of this document provides a more detailed guideline and tips for implementing the assignment specs.

What to submit and how to submit

You will submit the following:

1. Your code files, the **main file** as well as any **header/helper files**, along with a **readme file** explaining how to run your code. You can choose from C, Java, or Python to implement your code. If you are using C, you MUST submit a makefile/script along with your code. If you use Python, you must indicate the Python version in your readme file. You must make sure that the submitted code compiles and run on CSE Linux servers, because the markers will test your code there. Your assignment cannot be marked if it does not work in CSE servers and will be treated as a failure.

2. In addition to your code files, you must also submit a PDF **report** file not exceeding 3 pages in total. In the report, you should give a description of how your system works, possible improvements and extensions to your program/code that could be made in the future if you had time and indicate how you could realise them. If your program does not work under any particular circumstances, please report this too. Also indicate any segments of code that you have borrowed from the Web or other sources.

You can submit your assignment files using the **give command** in an *xterm* from any CSE machine. Make sure you are in the same directory as your code and report, then execute the following:

1. Type tar -cvf assign.tar filenames (e.g. tar -cvf assign.tar *.java report.pdf) **[NOTE: The system will only accept "assign.tar" submission name. All other names will be rejected.]**
2. When you are ready to submit, at the bash prompt type 3331
3. Next, type: give cs3331 assignment assign.tar (You should receive a message stating the result of your submission).

IMPORTANT: You can submit multiple times before the deadline. A later submission, however, will override the earlier submission, so make sure you submit the correct file. Do not leave until the last moment to submit, as there may be technical, or communications error and you will not have time to rectify it.

Late submission penalty

Late submissions will be accepted but will be penalised at the rate of **10% per day late**. For example, a student receiving an original (without penalty) mark of 18/20 but submitting 2 days late, will receive a final mark (after penalty is applied) of 18x0.8 = 14.4. **No submissions will be accepted after 5pm 28th of April 2020.**

Plagiarism

You are to write all of the code for this assignment yourself. All source codes are subject to strict checks for plagiarism, via highly sophisticated plagiarism detection software. These checks may include comparison with available code from the Internet and assignments from previous semesters. In addition, each submission will be checked against all other submissions of the current term. Do not post this assignment on forums where you can pay programmers to write code for you. We will be monitoring such forums. We are aware that a lot of learning takes place in student conversations, and do not wish to discourage those. However, it is important, for both those helping others and those being helped, not to provide/accept any programming language code in writing, as this is apt to be used exactly as is, and lead to plagiarism penalties for both the supplier and the copier of the codes. Write something on a piece of paper, by all means, but tear it up/take it away when the discussion is over. It is OK to borrow bits and pieces of code from sample socket code out on the Web and in books. You MUST however acknowledge the source of any borrowed code. This means providing a reference to a book or a URL when the code appears (as comments). Also indicate in your report the portions of your code that were borrowed. Explain any modifications you have made (if any) to the borrowed code. If detected, the penalty for plagiarism can be a ZERO mark for the assignment as well as further measures related to student conduct.

APPENDIX – Detailed Guideline and Implementation Tips

In this appendix, we provide more details and tips for implementing the required peer-to-peer network. We will use Figure 1 as reference for the steps described below.

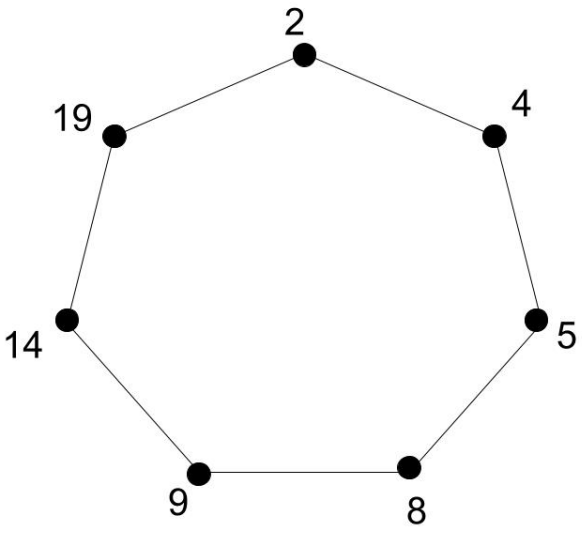

Figure 01

Step 1 - Initialization

This step shows how a set-up a script that can open 7 xterm windows to start the 7 peers shown in Figure 1.

Xterm -hold -title "Peer 2" -e "java p2p init 2 4 5 30" &
Xterm -hold -title "Peer 4" -e "java p2p init 4 5 8 30" &
Xterm -hold -title "Peer 5" -e "java p2p init 5 8 9 30" &
Xterm -hold -title "Peer 8" -e "java p2p init 8 9 14 30" &
Xterm -hold -title "Peer 9" -e "java p2p init 9 14 19 30" &
Xterm -hold -title "Peer 14" -e "java p2p init 14 19 2 30" &
Xterm -hold -title "Peer 19" -e "java p2p init 19 2 4 30" &

In order to understand the input arguments, let us look at the first line more closely:

java p2p init 2 4 5 30
java p2p <TYPE> <PEER> <FIRST_SUCCESSOR> <SECOND_SUCCESSOR> <PING_INTERVAL>

TYPE - Type of run
init - we use when we initialize the network,
join -use when new peer is joining to existing network (we'll discuss this more later)
PEER - Peer ID
FIRST_SUCCESSOR - Id of first successor
SECOND_SUCCESSOR -Id of second successor

PING_INTERVAL (seconds) - We'll discuss this shortly

We can identify port number of each by 12000+i (i = peer ID), for example.

As an instance,
Xterm -hold -title "Peer 1" -e "java p2p init 2 4 5 30"
Start peer 1 at port 12002
Peer 1 can find first successor on port 12004 and second successor on port 12005.

Step 2 - Ping Successors

Soon after initialization, each peer pings its 2 successors to check their existence. As an instance, Peer 2 pings Peer 4 & 5 to verify whether they are dead or alive. Xterm needs to display the following messages:

Peer 2
> Ping requests sent to Peers 4 and 5
Peer 4
> Ping request message received from Peer 2
Peer 5
> Ping request message received from Peer 2

If Peer 4 and 5 still alive:

Peer2
> Ping response received from Peer 4
> Ping response received from Peer 5

Then after every **Ping Interval** (e.g., every 10 seconds) each peer needs to send ping messages to each successor to verify their existence. **Use UDP for pinging.**

*NOTE: It is important to note that the messages displayed in the terminal should differentiate between **ping request** and **ping response** messages. You will need to decide on how often you send the ping messages. You should not send them too often, otherwise you may overwhelm the computer. However, please do not make the evaluator/marker wait for more than 60 seconds before it shows the output. Otherwise it will be assumed that the program does not work.*

Step 3 - Peer Joining

Assumer Peer 15 is joining to the network. Peer 15 only knows about Peer 4's details (i.e., the ID of Peer 4). Then Peer 15 can start from a new xterm window by executing:

Java p2p join 15 4 30
Java p2p <TYPE> <PEER> <KNOWN_PEER> <PING_INTERVAL>

Correct position of Peer 15 is between Peer 14 and Peer 19. Hence peers 4,5,8, and 9 forward the request message to its successors over **TCP.** When Peer 14 receives a forwarded message from Peer 9, it will contact Peer 15 over **TCP** and send network join details. Similarly, Peer 14 sends details about Peer 15 to its first predecessor, Peer 9, via **TCP** (Peer 14 can learn about its predecessors by pinging successors in step 2). Finally Peer 15 can join the network by using details from Peer 14.

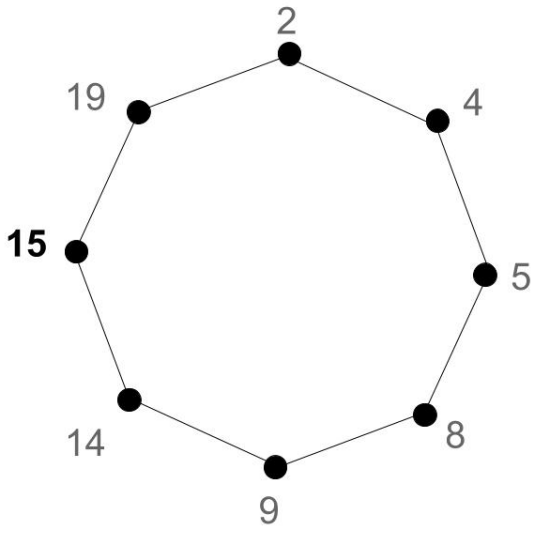

Figure 02

Expected outputs of each xterm windows are as follows:

Peer 4
> Peer 15 Join request forwarded to my successor
Peer 5
> Peer 15 Join request forwarded to my successor
Peer 8
> Peer 15 Join request forwarded to my successor
Peer 9
> Peer 15 Join request forwarded to my successor
Peer 14
> Peer 15 Join request received
> My new first successor is Peer 15
> My new second successor is Peer 19
Peer 9
> Successor Change request received
> My new first successor is Peer 14
> My new second successor is Peer 15
Peer 15
> Join request has been accepted
> My first successor is Peer 19
> My second successor is Peer 2

Step 4 - Peer Departure (Graceful)

Here, a peer informs its predecessors before it departs from the network. The peer which is going to depart, will enter "Quit" on its xterm terminal. For an evaluation of your assignment, let us assume Peer 9 wants to depart gracefully. Peer 8 and Peer 5 are notified by Peer 9 about its departure. Peer 9 sends details of its successors to Peer 14 and Peer 19 (Peer 9 can learn about its predecessors by pinging successors in step 2). All this information is sent over **TCP**. Finally, Peer 8 and 5 update their successors.

This is the network after Peer 9's departure:

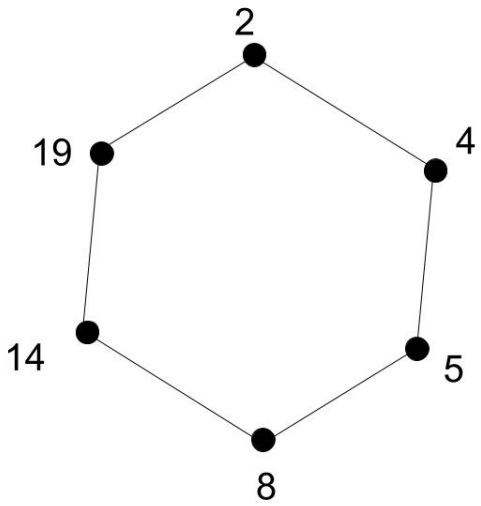

Figure 03

Expected outputs of each xterm windows are as follows:

Peer 9
> Quit
Peer 8
> Peer 9 will depart from the network
> My new first successor is Peer 14
> My new second successor is Peer 19
Peer 5
> Peer 9 will depart from the network
> My new first successor is Peer 8
> My new second successor is Peer 9

Step 5 - Peer Departure (Abrupt)

In this step, we kill one peer by pressing ctrl+c in the terminal. For an evaluation of your program, let us assume that Peer 14 is killed by pressing ctrl+c in the terminal. Now peer 9 and peer 8 (predecessors of Peer 14) learn that Peer 14 is no longer alive, by pinging successors that we discussed in step 2. Because we use UDP for ping successors, packets can be technically lost during the communication. Therefore, peers need to wait at least a few consecutive lost ping-successor messages before it decides its successor is dead. You need to find the best number of consecutive lost ping successors messages by doing some experiments (2 or 3 may be OK given that loss is unlikely to happen within an operating system).

After learning that Peer 14 is no longer alive, Peer 9 sends a message to Peer 19 to get its first successor. Similarly, Peer 8 sends message to Peer 9 requesting its successors (because when the time Peer 9 receives message from Peer 8, it may have already updated its successors, then Peer 8 needs to use the first successor of Peer 9 as its 2nd successor, otherwise needs to use the 2nd successor of Peer 9 as its 2nd successor)

Expected outputs of each xterm windows:

Peer 9
> Peer 14 is no longer alive
> My new first successor is Peer 19
> My new second successor is Peer 2
Peer 8
> Peer 14 is no longer alive
> My new first successor is Peer 9
> My new second successor is Peer 19

NOTE: Printing messages to the terminal is highly important as the evaluator/marker will mark you according to the outputs. So, if no output is printed, the evaluator will assume that the file request was unsuccessful. Please do not make the evaluator wait for more than 60 seconds for your program to detect a node has been killed.

Step 6 - Data Insertion

In this step, one of the peers in the network is requested to store a file in the network by entering "Store <file name>" in its xterm. Peers need to find the correct location in the network (correct peer) to store the file by using the hash function Mod(filename/256). For an example evaluation of your program, let us consider a file name of 2067. Remainder of 2067 when divided by 256 is 19. Hence correct location of the file is Peer 19. At the beginning, Peer 8 is asked to store File 2067. Peer 8 knows that it is not the correct location to store hence it forwards the message to its successor. Likewise, Peer 9 and Peer 14 do the same thing. Finally, Peer 19 accepts to store the file.

Expected outputs of each xterm windows:

Peer 8
> **Store 2067**
> Store 2067 request forwarded to my successor
Peer 9
> Store 2067 request forwarded to my successor
Peer 14
> Store 2067 request forwarded to my successor
Peer 19
> Store 2067 request accepted

Step 7 - Data Retrieval

In this step, one of the peers in the network requests a file that is stored in the network by entering "Request <file name>" in its xterm. Peers need to find the correct location in the network (correct peer) that stores the file by using hash function Mod(filename/256). As an example, let us assume that Peer 2 requests file 4103, which leads to a remainder of 7. Hence the file is stored in Peer 8. Peer 2 sends a message to its successor using **TCP**. Peer 4 and Peer 5 forward the request message to their successors. After Peer 8 receives the file request, it starts a direct **TCP** connection with Peer 2 and sends the file.

Initially we have a 4103.pdf file in the program directory. At the end, we have a duplicate file named received_4103.pdf at the same directory.

Expected output of each xterm windows

Peer 2
> Request 4103
> File request for 4103 has been sent to my successor
Peer 4
> Request for File 4103 has been received, but the file is not stored here
Peer 5
> Request for File 4103 has been received, but the file is not stored here
Peer 8
> File 4103 is stored here
> Sending file 4103 to Peer 2
> The file has been sent
Peer 2
> Peer 8 had File 4103
> Receiving File 4103 from Peer 8
> File 4103 received

Network setup script (init_java.sh & init_python.sh) can be executed using
./init_java.sh

If you cannot execute init_java.sh or init_python.sh, then fix the permissions by executing the following command in the command line

chmod u+x init_java.sh

End of Assignment Specs [We hope you enjoy this assignment!]