COMP3331/9331 Computer Networks and Applications
Assignment for Term 1, 2019
Individual Assignment
Version 1.1
Due: 26th April 2019

Updates to the assignment, including any corrections and clarifications, will be posted on the course website (https://moodle.telt.unsw.edu.au/course/view.php?id=37585). Please make sure that you check the course website regularly for updates. You can post questions in the assignment forum.

1. Change Log

1. Version 1.0 released on Feb 18, 2018.

2. Goal and learning objectives

For this assignment, you will be asked to implement a part of the peer-to-peer (P2P) protocol Circular DHT which is described in Section 2.6 of the text Computer Networking (6th ed) and would be discussed in the lecture. A primary requirement for a P2P application is that the peers form a connected network at all time. A complication that a P2P network must be able to deal with is that peers can join and leave the network at any time. For example, if a peer leaves the network suddenly (because it has crashed), then the remaining peers must try to keep a connected network without this peer. It is therefore necessary to design P2P networks so that they can deal with these complications. One such method has been described in Section 2.6 of the text, under the heading of Peer Churn for circular DHT. You will also learn how to implement reliable data transmission over UDP.

2.1. Learning Objectives

On completing this assignment, you will gain sufficient expertise in the following skills:

1. Understanding of routing mechanism and connectivity maintenance in peer-to-peer systems, and Circular DHT in particular.
2. Socket programming for both UDP and TCP transport protocols.
3. Protocol and message design for applications.
4. Basic understanding of creating a reliable connection.

3. Background

The following is extracted from the Peer Churn section of the text:
“In P2P systems, a peer can come or go without warning. Thus, when designing a DHT, we also must be concerned about maintaining the DHT overlay in the presence of such peer churn. To get a big-picture understanding of how this could be accomplished, let’s once again consider the DHT in Figure 2.27(a) [Reproduced here as Figure 1]. To handle peer churn, we will now require each peer to track (that is, know the IP address of) its first and second successor; for example, peer 4 now tracks both peer 5 and peer 8. We also require each peer to periodically verify that its two successors are alive (for example, by periodically sending ping messages to them and asking for responses). Let’s now consider how DHT is maintained when a peer abruptly leaves. For example, suppose peer 5 in Figure 2.27(a)[Figure 1 in this assignment spec] abruptly leaves. In this case, the two peers preceding the departed peer (4 and 3) learn that 5 has departed, since it no longer responds to ping messages. Peers 4 and 3 thus need to update their successor state information. Let’s consider how peer 4 updates its state:

1. Peer 4 replaces its first successor (peer 5) with its second successor (peer 8).
2. Peer 4 then asks its new first successor (peer 8) for the identifier and IP addresses of its immediate successor (peer 10). Peer 4 then makes peer 10 its second successor.

Having briefly addressed what has to be done when a peer leaves, let’s now consider what happens when a peer wants to join the DHT. Let’s say a peer with identifier 13 wants to join the DHT, and at the time of joining, it only knows about peer 1’s existence in the DHT. Peer 13 would first send peer 1 a message, saying ”what will be peer 13’s predecessor and successor?” This message gets forwarded through the DHT until it reaches peer 12, who realises it will be peer 13’s predecessor and its current successor, peer 15, will become its successor. Next, peer 12 sends this predecessor and successor information to peer 13. Peer 13 can now join the DHT by making peer 15 its successor and by notifying peer 12 that it should be its immediate successor to peer 13.

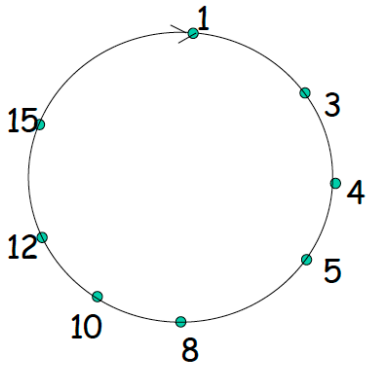

Figure 1. DHT configuration.

4. Assignment description

For this assignment, you are asked to write a Java, C, or Python program which can handle query/response and peer churn (leaving only) for circular DHT as described in Section 3. However, there are a few important points that you need to note:

1. You will be running each peer in an xterm on one machine. Therefore, tracking a peer no longer means knowing the IP address but in your case, it means knowing the port numbers (we will further elaborate on this point later).
2. You will not be able to use built in ping in OS to determine whether the two successors of a peer are still alive. You will need to implement your own ping mechanism.

Please make sure that your P2P program is working in the CSE laboratory as we will test (and mark) your assignment using CSE machines.

If your program does not work on CSE machine, you will get '0' mark for assignment as we cannot accept running the code on your own machine.

In the following description, we will continue to use the example in Figure 1, but a very important point that you need to note is that: you **should not** make any assumption regarding the number of peers and their identities in your program. The number of peers and their identities to be used for the evaluation can change from student to student. You should therefore make sure that you test that your program can deal with the general situation. You can assume that there would be no more than 10 peers and the peer identities is within the range of 0-255.

There are a number of mandatory requirements:

1. You must name your program *cdht.c*, *cdht.java*, or *cdht.py*.
2. This program uses 3 input arguments. The three input arguments are all integers in the range of [0,255]. They are used to initialise the network as discussed in section 4.1 (initialization step).
3. For python programs, please indicate the version of python you used clearly in the first lines of your report.
4. For c programs, please include makefile with your submission and explain the process for making the file in your report. You must also show this on your demo.

4.1. Assignment specification

In this section, we will discuss the steps that you must take to implement this assignment. Make sure your program covers all these steps to get full mark. The marking guideline is discussed in section 7.

Step 1: Initialization

The main aim of this step is to initialize the DHT. For the example in Figure 1, which has 8 peers, 8 xterms will be open to initialise 8 peers. This step will be performed by using a set-up script, of which the following is an example:

```
xterm -hold -title "Peer 1" -e "java cdht 1 3 4 400 0.1” &
xterm -hold -title "Peer 3" -e "java cdht 3 4 5 400 0.1" &
xterm -hold -title "Peer 4" -e "java cdht 4 5 8 400 0.1" &
xterm -hold -title "Peer 5" -e "java cdht 5 8 10 400 0.1" &
xterm -hold -title "Peer 8" -e "java cdht 8 10 12 400 0.1" &
xterm -hold -title "Peer 10" -e "java cdht 10 12 15 400 0.1" &
xterm -hold -title "Peer 12" -e "java cdht 12 15 1 400 0.1" &
xterm -hold -title "Peer 15" -e "java cdht 15 1 3 400 0.1" &
```

This script opens up 8 xterms and starts a peer in each xterm. The -e option asks the xterm to execute the command specified after the switch (For meaning of the other xterm options, please use man xterm).
In order to understand what the input arguments represent, let us look at the first line more closely. The executable part of this line is `“java cdht 1 3 4 400 0.1”`. The first input argument, i.e., 1” is the identity of the peer to be initialised. The second and third arguments, i.e., 3 and 4, are the identities of the two successive peers. The forth argument is Maximum Segment Size (MSS) used to determine the size of the data that must be transferred in each segment (discussed more in Step 3). MSS in this example is set to 400. The last argument is the drop probability which must be between 0-1 (discussed more in Step 3). You will find that the above file initialises the configuration of the circular DHT given in Figure 1. The above configuration file is available from the assignment website. There are also C and Python versions.
Given the above set-up script, each peer will know its identity and the identities of its two successors. There are a few assumptions that you can safely assume:

1. We will ask you to work with at most 10 peers.
2. The minimum number of peers is 4.
3. The identity of a peer is always in the range of [0,255].
4. The set-up script that we give you will always correctly describe a circular DHT.

You can always create more set-up scripts for testing. Although for the demo (see Section 5) you will run your code with the above configuration, but we will also test your code with another configuration that we have to ensure that you did not hardcode anything in the program. You would not know that configuration, but the configuration will follow the above rules. Thus, make sure you test your code with different configurations.

Step 2: Ping successors

After initialisation, each peer will start pinging its two successors to see whether they are alive. The ping mechanism should define two types of messages: 1) ping request, and 2) ping response. The ping messages should use **UDP** protocol. You can assume that a peer whose identity is *i* will listen to the UDP port *50000* + *i* for ping messages. For example, peers 4 and 12 will listen on UDP ports 50004 and 50012 respectively for ping request messages. Each peer should output a line to the terminal when a ping request message is received from any of its two predecessors. For example, peer 10 is expected to receive ping request messages from peers 5 and 8; when peer 10 receives a ping request message from peer 5, it should output the following line to the terminal:

```
A ping request message was received from Peer 5.
```

Similarly, if peer 10 receives a ping request message from peer 8, it should output the following line to the terminal:

```
A ping request message was received from Peer 8.
```

The numbers ”5” and ”8” are correct for this example, your program is of course expected to print the correct identities for the situation.

NOTE: Printing ping messages to the terminal is highly important as the evaluator will mark you according to the outputs. So, if no output is printed, the evaluator will assume that the ping was unsuccessful. Even if you implement the ping correctly but there is no output, you will get 10% of the mark for that part (see Section 7).

Since the title of each xterm displays the identity of each peer, you should be able to keep track of each peer. When a peer receives a ping request message from another peer, it should send a ping response message to the sending peer so that the sending peer knows that the receiving peer is alive. When a peer receives a ping response from another peer, it should display on the terminal. For example, if peer 10 is still alive, peer 5 is expected to receive a ping response message from peer 10, and peer 5 should display:

```
A ping response message was received from Peer 10.
```

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 very often, otherwise you may overwhelm the computer. Please don’t make the evaluator waiting for more than 3 minutes before it shows the output. Otherwise it will be assumed that the program does not work.

Step 3: Requesting a file

An application of DHT is distributed file storage. In this step, you will be using the P2P network that you have created to request for files. To request a file with filename X from peer Y, the requester will type “`request X`” to the xterm of peer Y. Thus, your program must be able to receive string inputs. Before describing what will be evaluated, we first specify the rules for filenames, hashing, file location and messages:

- *Filename:* You can assume all the filenames in this P2P system are four digit numbers. Some examples of valid filename are 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:* All the peers in the network use the same hash function. Note that you need to implement a simple hash function, so you do not need to use conventional hash functions. The hash function will be applied to the filename. Given the filename format defined earlier, each filename is an integer in the range [0, 9999]. To compute the hash of a file, you must compute the remainder of the filename integer when it is divided by 256. This gives you a hash value as an integer in [0,255]. For example, for the file with filename 2012, its integer

equivalent is 2012 and the remainder when 2012 is divided by 256 is 220; thus, the hash of the file 2012 is 220.

- *File location:* The location where a file is stored in a P2P network depends on the hash of the file as well as the peers that are currently in the network. The rule is, for a file whose hash is *n* (where n is an integer in [0, 255]), the file will be stored in the peer that is the closest successor of *n*. We will use the P2P network in figure 1 to illustrate this rule. If the hash values of three different files are 6, 10 and 210, then they will be stored, respectively, in peers 8, 10 and 1.
- *Request and response messages:* If a peer wants to request for a file, the peer (which we will call the requesting peer) will send a file request message to its successor. The file request message will be passed round the P2P network until it reaches the peer that has the file (which we will call the responding peer). The responding peer will send a response message directly to the requesting peer. We require both these messages, i.e., file request and response, to be sent over **TCP**. You can assume that a peer whose identity is *i* will listen to the TCP port *50000* + *i* for these messages.
- Transfer the file: The responding peer has to transfer a file to the requesting peer over UDP connection. Unlike ping packets, the file has to be sent directly to the requesting peer. Recall that UDP is not reliable. You need to make the connection reliable by implementing a simple protocol with stop-and-wait behaviour. Recall that in stop-and-wait, the sender sends a data packet to the receiver, and waits until it receives an acknowledgement, or a timeout happens. In case an ACK is received, the sender sends the next part of data, and if a timeout occurs, the sender re-transmits the data. Note that, this will be a very simple reliable connection which only deals with packet lost. The responding peer forms packet with MSS bytes of data. Then, it has to add the sequence number and encapsulate both as a packet. This packet will then be transmitted to the requesting peer. The responding peer will start a timer for the packet which essentially is needed for timeout operation. Once the requesting peer received the data packet, it will generate a corresponding acknowledgement and send back to the responding peer. On receipt of the acknowledgement packet, the responding peer will transfer the next MSS bytes of data. If the data packet gets lost, the requesting peer will not transfer the acknowledgement (as no data is received) and thus the timeout will happen in the responding peer. Recall that the responding peer maintains a timer for each data packet it sends. The timeout-interval is set to 1.5 seconds in this assignment. If the responding peer does not receive an acknowledgement for the sent packet within timeout-interval, it considers the packet to be lost, and thus have to re-transmit the packet.

  The responding and requesting peers must maintains a log file named responding_log.txt and requesting_log.txt where they record the information about each segment they send and receive. The format of the log file shall be as follow:

  <event> <time> <sequence-number> <number-of-bytes-of-data> <acknowledgement-number>

  Where <event>=snd/rcv/drop/RTX. Snd = send, rcv= receive, drop= packet dropped, and RTX= retransmission.

  <time> is the time since the start of the program.

  Note that we will test your program in a single machine, thus packet drop in UDP will be rare (if any). To test your program, the peers shall accept a

drop_rate value as input which is the probability in which packets are dropped. Before delivering a packet to the socket to be sent to the requesting peer (i.e., after setting the timeout and forming the packet), the responding peer creates a random number (between 0-1) and checks if the random number is smaller than the drop probability. If yes, the responding peer will not send the packet. The timer set by the responding peer will eventually reach the timeout-interval which in turns triggers packet re-transmission. In log file, you shall print the log corresponding to the drop packet when the packet is dropped. Thus, the time difference between the dropped packet and the retransmitted packet must be at least timeout-interval.

Know let us describe the process you need to take for your demo. The illustration is based on the network in figure 1. Use peer 8 as the requesting peer and the filename 2012 as the filename. Place the PDF named 2012 in the same folder as the execution files are located. Type the string "`request 2012`" in the xterm for peer 8 to inform peer 8 to begin the file request process. Peer 8 should format a file request message and forward it to its successor. Peer 8 should display in its xterm the following:

```
File request message for 2012 has been sent to my successor.
```

The successor to peer 8, which is peer 10, should decide whether it has the file. The decision should be negative in this case and peer 10 should then forward the file request message to its successor. Peer 10 should display:

```
File 2012 is not stored here.
File request message has been forwarded to my successor.
```

The file request message will then be passed onto peer 10, 12 and then peer 15. All these peers should decide that they do not have the file and forward the file request message to their respective successor. Peers 10, 12, and 15 are expected to display:

```
File 2012 is not stored here.
File request message has been forwarded to my successor.
```

After peer 1 has received the file request message, it should decide that it has the file 2012. Peer 1 will then format a response message and send it directly to requesting peer, which is peer 8. Peer 1 also must print that it starts sending the file and when finished it must print that the transmission is finished. Peer 1 should display:

```
File 2012 is here.
A response message, destined for peer 8, has been sent.

We now start sending the file ……

The file is sent.
```

After peer 8 has received the response message from peer 1, it should display:

```
Received a response message from peer 1, which has the file
2012.

We not start receiving the file ……

The file is received.
```

At the end of the data transition there should be a duplicate of the file in the directory in which the codes are located. You must open the duplicate copy as well. You can name the copy of the file as "received_file.pdf". You also need to show the log files.

NOTE:

1) Printing messages to the terminal and the log files are highly important as the evaluator will mark you according to the outputs. So, if no output is printed, the evaluator will assume that the file request was unsuccessful. Even if you implement the request process correctly but there is no output, you will get 10% of the mark for this part.

2) Please check marking criteria in section 7 for more detailed discussion on how we will require you to do the demo.

Important note: In the above illustration, we have assumed that the requesting peer does not have the file that it is requesting. For this assignment, you can safely assume that this is the case. You can always assume that we will give you a requesting peer and filename combination such that the requesting peer and the responding peer are different.

Step 4: Peer departure

In this step, one of the peers depart from the network in a graceful manner, which in this assignment means the peer informs its predecessors before it departs from the networks. In order to realise graceful departure, we ask each peer should monitor the standard input for the input string "`quit`" (The other string that each peer should monitor is request followed by a valid filename, as in Step 3).

For demo, consider that peer 10 departs from the network gracefully. Type the string `quit` (followed by carriage return) in the xterm for peer 10 to inform peer 10 to begin the departure operation. Peer 10 should send a departure message to peers 5 and 8 to inform them that it wants to depart from the network (Note that peer 10 learns that its predecessors are peers 5 and 8 from the ping request messages). Peer 10 will also inform peers 5 and 8 in the departure message that its successors are peers 12 and 15. We require that the departure messages are to be sent over **TCP**. You can assume that a peer whose identity is *i* will listen to the TCP port *50000* + *i* for these messages. After peer 10 has sent the departure message to peers 5 and 8, it should make sure that peers 5 and 8 have received the message before terminating the cdht program.

After peer 8 has received the departure message from peer 10, it should output the line to the terminal:

```
Peer 10 will depart from the network.
```

Since peer 10 is to depart, peer 8 will then make peer 12 its first successor and peer 15 its second successor. Note that peer 8 deduce the identities of its new successors from the departure message from peer 10. Peer 8 will output the lines:

```
My first successor is now peer 12.
My second successor is now peer 15.
```

The above describes what peer 8 should do. In a similar way, peer 5 should output these lines to the terminal:
Peer 10 will depart from the network.

```
My first successor is now peer 8.
My second successor is now peer 12.
```

Note that for correct implementation, after peers 5 and 8 have updated their successors, they should start pinging their new successors. Ping messages will be used to confirm this.

NOTE: Printing messages to the terminal is highly important as the evaluator will mark you according to the outputs. So, if no output is printed, the evaluator will assume that the file request was unsuccessful. Even if you implement the departure process correctly but there is no output, you will get 10% of the final mark for this part.

Step 5: Kill a peer

In this step, you need to select one of the xterms and kill it (by pressing Ctrl + C in the xterm). This is to mimic the event of a peer leaving the network ungracefully. Kill the terminal in which peer 5 is running. After some time (depending on how often ping messages are sent), peers 3 and 4 should find out that peer 5 is no longer there. Peer 4 should output the line to the terminal:

```
Peer 5 is no longer alive.
```

We will come back to explain how you can determine whether a peer is alive shortly. Since peer 5 is no longer alive, peer 4 will then make peer 8 its first successor. It will output the line:

```
My first successor is now peer 8.
```

Peer 4 will then send a message to peer 8 to find out the identity of the first successor of peer 8. We require this message to be sent over **TCP**. You can assume that a peer whose identity is i will listen to the TCP port 50000+i for these messages. Peer 8 should reply to peer 4's query and let peer 4 knows that its successor is peer 12 (Note that peer 10 has departed from the network gracefully earlier). After peer 4 receives this message, it should output the line:

```
My second successor is now peer 12.
```

The above describes what peer 3 should do. In a similar way, peer 3 should output these lines to the terminal:

```
Peer 5 is no longer alive.
My first successor is now peer 4.
My second successor is now peer 8.
```

Note that for correct implementation, after peers 3 and 4 have updated their successors, they should start pinging their new successors. The ping messages would be evaluated for this aim. Let us now return to explain how a peer can know whether its successors are alive. You can determine this by using ping messages and it is important that you include a sequence number field in the ping messages. Let us assume that you have decided to have the peers ping each other every *t* seconds. Let us consider what happens when peer 4 pings peer 5. Peer 4 will send peer 5 a ping request message every *t* seconds. The first ping request message will have sequence number 0, the next one has sequence number 1, the one after will have a sequence

number 2 and so on. When peer 5 receives a ping request message from peer 4 with sequence number *n*, it will formulate a ping response message and includes the same sequence number *n* in the ping response message. In this way, peer 4 knows that peer 5 has received the ping request message with sequence number *n*. Of course, a peer can only respond to a ping request message if it is still alive. Let us assume that peer 4 sends 16 messages to peer 5 with sequence numbers 0 to 15. Peer 5 sends ping responses for the first 12 ping request messages (with sequence numbers 0-11) but after that peer 5 fails to respond because it has been killed. From peer 4's perspective, it knows that peer 5 has not given any replies to the last four consecutive messages (with sequence numbers 12-15), it may use this property that peer 5 has not responded to the last 4 messages to claim that peer 5 is no longer alive. Note that this example has assumed that a peer will declare that a successor is not alive if 4 consecutive ping messages are not replied, you will need to experiment to find out what a good number is. You should be aware that UDP does not provide guaranteed delivery, therefore there is a chance that UDP messages may be lost. Therefore, it is not a good idea to declare a peer is not alive if one ping request message is not responded because the message could have been lost in transit. Note that there are three parameters that you will need to decide. The first parameter is the number of consecutive ping request messages that a peer fails to respond to before that node is declared to be no longer alive. The second parameter is the frequency of the ping messages. The third parameter, which is generally called timeout in computer networking literature, is the time that a peer needs to wait before declaring that a particular ping response has not been received.

Please don't make the evaluator wait for more than 3 minutes for your program to detect a node has been killed.

5. Mandatory Screencast Demo

The screencast demo must go through steps 2-5 outlined in section 4.1 (inclusive). You may describe your program during demo or just run the steps mentioned above. Demo is limited to 5 minutes and longer demos are subjected to 50% penalty of the final mark (see Section 7). You must show your code during the screencast and you must log in to your CSE account while doing demo. We will test your program using CSE machines, thus, please make sure you run and demo your code on CSE machines. Failure to do this may result in failure of running your code on CSE machines and thus loosing significant part of mark. The evaluator will compare the results shown in your demo with the results he/she gets from running your code on CSE machines.

Please upload your screencast on YouTube as *unlisted video* **before the submitting** and include the YouTube link in your report.

Check the marking guide in section 7. Make sure that you show and do all the steps as outlined in this marking guide. We will mark you according to the marking guide and if you fail to do any of the criteria, then you will lose mark for that (e.g., failure in showing that you used TCP or UDP for messages).

You can use the following command in CSE machines to record the screen:

```
avconv -f x11grab -r 10 -s 1920x1080 -i :0.0 -vcodec libx264 -threads 4 /tmp/screencap.avi
```

Important note

The screencast is **mandatory** and is considered as demo for assignment 1. Without demo, you will get maximum 10% of the final mark of the assignment with no exception.

6. Additional Notes

- This is not a group assignment. You are expected to work on this individually.
- How to Start: Sample client and server programs you have done for Labs 2 and 3 can prepare you for this assignment. Also, you can use sample client-server programs in Moodle.
- Language and Platform: You are free to use either C, Python, or JAVA to implement this assignment. Please choose a language that you are comfortable with.
- The programs will be tested on CSE Linux machines. So please make sure that your entire application runs correctly on these machines (i.e., your lab computers). This is especially important if you plan to develop and test the programs on your personal computers (which may possibly use a different OS or version).
- You are free to design your own format and data structure for the messages. Just make sure your program handles these messages appropriately.
- You are encouraged to use assignment forum to ask questions and to discuss different approaches to solve the problem. However, you should not post your solution nor any code fragment on the forum.
- You must submit a maximum 3 pages report (named report.pdf) with your codes. In report, describe the steps you take to implement the program, design choices including the interval between pings and the number of lost packets before assuming a peer is killed, and the message formats you used. If you have borrowed code, please indicate that here with the source reference. If you are using python, include the version of python you are using in your report. Make sure you include the **link to your demo**.

7. Marking Policy

You should test your program rigorously before submitting your code. Note that you must submit a demo of your work otherwise you will lose 90% of the final mark. Your code will be marked using the following criteria:

7.1. Ping successors (2 marks)

7.1.1. Correct compilation of all the files*:* Compile your files during the Demo. For languages that don't need compiling files, simply run your code (part 1.3 below) and you will get the mark for this part (0.25 mark).
7.1.2. Using UDP for ping messages: You must show the part of your code that proves that you used UDP for ping (0.25 mark).
7.1.3. Correct receiving of ping requests: Run all peers. Ping receive message must be shown in all peers with the correct ids (0.75 mark).
7.1.4. Correct receiving of ping response: Ping response message must be shown in all peers with the correct ids (0.75 mark).

7.2. Requesting a file (7.5 marks)

7.2.1. Initializing and sending the file request to the successor: The requesting peer must initialize a packet and send it to its successor once “request x” is typed to its terminal (where x is the filename) (1.5 marks).

7.2.2. Forwarding the request to the correct node that has the requested file: The intermediate peers must show the proper message that they received and forwarded the request as per discussion in section 4.1. (1.5 marks).

7.2.3. Sending the response directly to the requester using TCP: The requester must print that it sends the response to the requesting peer. No other node should receive this message (and print anything). Show in your code that you use direct messaging using TCP for this part (0.75 marks).

7.2.4. File at the requesting peer is identical with the file sent by the responder peer (0.75)

7.2.5. Stop-and-wait behavior (alternate send and receive), correct MSS (1.5 mark)

7.2.6. Drop probability is correct (according to the drop_rate supplied as input) (0.75)

7.2.7. Segments are transmitted after timeout (0.75)

7.3. Peer departure (4.5 marks)

7.3.1. Sending the departure message to the predecessors: Upon typing “quit” request to terminal, the predecessors must appropriate message indicting that peer 10 left the network (1.75 marks).

7.3.2. Using TCP for sending departure message: Show this in your code (1 marks).

7.3.3. Choosing correct successors after departing node 10 by its predecessors: Printed ping messages will be used to evaluate this part (1.75 marks).

7.4. Kill a peer (4 marks)

7.4.1. Correct detection of the node that left the network in reasonable time (no more than 3 minutes): Appropriate message must be printed on the peers (1.25 marks).

7.4.2. Identifying the new successors by the predecessors of the node who has left: Appropriate message must be printed (1.75 marks).

7.4.3. Correct ping of the new successors: Appropriate ping message must be printed (1 marks).

Total mark with default configuration (i.e., the configuration discussed in Section 4.1 and shown in Figure 1): 18 marks

---

The tutor will run all the steps above (7.1 -7.4) with a different DHT configuration setup. The same marking policy applies. The final mark for the implementation part of your assignment would be:

Final implementation mark = 0.6*(mark of default configuration) + 0.4 * (mark of a different configuration)

Recall that you must also write a maximum 3 pages report. The report worth **2 marks** of your final assignment mark. Thus, the total mark for assignment is as follows:

Assignment mark = report + final implementation mark

Where report is out of 2 and implementation mark is out of 18.

Penalty

- If your demo is more than 5 minutes, your final mark will be penalized by **50%.**
- If your program does not print appropriate outputs, you will get **10%** of the mark for that particular part (see above).
- If demo is not included in the report or is modified after submitting the assignment, you will be penalized by **90%** of the final mark**.**
- If we cannot run your code on CSE machines, zero mark will be awarded for your implementation.

8. Assignment Submission

Please ensure that you use the mandated file name. You may of course have additional header files and/or helper files. If you are using C, then you MUST submit a makefile/script along with your code (not necessary with Java). In addition, you should submit a small report, named report.pdf (no more than 3 pages) describing the program design, a brief description of how your system works and your message design. If you use python you must indicate the python version in your report. Also discuss any design tradeoffs considered and made. Describe possible improvements and extensions to your program and indicate how you could realise them. If your program does not work under any particular circumstances please report this here. Also indicate any segments of code that you have borrowed from the Web or other books.

You are required to submit your source code and report.pdf. You can submit your assignment 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 do the following:

1. Type tar -cvf assign.tar filenames (e.g. tar -cvf assign.tar *.java report.pdf)
2. When you are ready to submit, at the bash prompt type 3331
3. Next, type: `give cs3331 assignment1 assign.tar` (You should receive a message stating the result of your submission).

Important notes

- The system will only accept “assign.tar” submission name. All other names will be rejected.
- Ensure that your program/s are tested in CSE Unix machine before submission. In the past, there were cases where students had problems in compiling and running their program during the actual demo. To avoid any disruption, please ensure that you test your program in CSE Unix-based machine before submitting the assignment.
- The evaluation is based on the files that you have submitted.

You can submit as many time before the deadline. A later submission 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 penalty will be applied as follows:

- 1 day after deadline: 10% reduction
- 2 days after deadline: 20% reduction
- 3 days after deadline: 30% reduction
- 4 days after deadline: 40% reduction
- 5 days after deadline: 50% reduction
- 6 days after deadline: 60% reduction
- 7 days after deadline: 70% reduction
- more than 7 days late: NOT accepted

NOTE: The above penalty is applied to your final total. For example, if you submit your assignment 1 day late (10% penalty) and you score 18, then your adjusted final mark, after penalty, will be 18x0.9= 16.2.

9. 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 semester. Do not post this assignment on forums where you can pay programmers to write code for you. We will be monitoring such forums. Please note that we take this matter quite seriously. The LIC will decide on appropriate penalty for detected cases of plagiarism. The most likely penalty would be to reduce the assignment mark to ZERO. We are aware that a lot of learning takes place in student conversations, and don't 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.