COMP3331/9331 Computer Networks and Applications
Assignment for Session 1, 2015

Version 1.0
Due: 11:59pm Friday, 29 May 2015 (Week 12)

Updates to the assignment, including any corrections and clarifications, will be posted on the subject website. Please make sure that you check the subject website regularly for updates. Please post any questions related to assignment on Message Board.

1 Change Log

1. Version 1.0 released on March 25, 2015.

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.)*. 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 are asked to implement this using socket programming and your assignment marker will evaluate these.

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

3 Background

The following is extracted from the Peer Churn section of the text (pages 155-156):

> *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 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) 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.*

4 Assignment description

For this assignment, you are asked to write a Java or C 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:

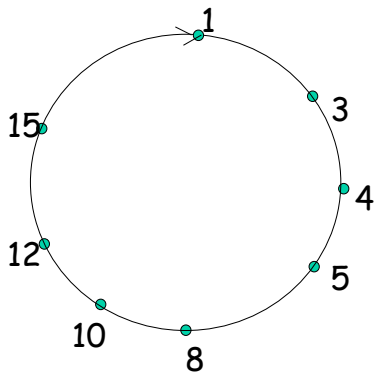

Figure 1: DHT configuration

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 come back to this point later.)

2. You will not be able to use ping to determine whether the two successors of a peer are still alive. You will need to implement your own ping mechanism.

You will be asked to make sure that your P2P program is working in the CSE laboratory. You can assume that your assignment will be evaluated on CSE machines. 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 your programs can deal with the general situation.

4.0.1 Standard and extended versions

Note that there are two versions of this assignment, a standard version (with a total of 60 marks) and an extended version (with a total of 65 marks of which 5 marks are bonus marks). The stan-

dard version requires you to implement Steps 1–4 (see section 4.1), while the extended version requires you to implement Steps 1–5.

4.0.2 Mandatory file requirements

There are a number of *mandatory* requirements.

1. You must name your program `cdht.c` or `cdht.java` if you attempt the standard version, or `cdht_ex.c` or `cdht_ex.java` if you attempt the extended version

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.

4.1 Steps of evaluation

The steps of your evaluation will be:

Step 1

In this step, evaluator will initialise a circular DHT. For the example in Figure 1 which has 8 peers, you will open 8 xterm and initialise each peer. 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" &
xterm -hold -title "Peer 3" -e "java cdht 3 4 5" &
xterm -hold -title "Peer 4" -e "java cdht 4 5 8" &
xterm -hold -title "Peer 5" -e "java cdht 5 8 10" &
xterm -hold -title "Peer 8" -e "java cdht 8 10 12" &
xterm -hold -title "Peer 10" -e "java cdht 10 12 15" &
xterm -hold -title "Peer 12" -e "java cdht 12 15 1" &
xterm -hold -title "Peer 15" -e "java cdht 15 1 3" &
```

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 first input argument is the identity of the peer to be initialised. The second and third arguments are the identities of the two successive peers. 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 is also a version that works for C language. If you attempt the extended version, you should replace `cdht` by `cdht_ex`.

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 identity of a peer is always in the range of [0,255].

3. The set-up script that we give you will always correctly describe a circular DHT.

We have provided another set of set-up scripts on the web-site for you to use. You can always create more set-up scripts for testing. It is important to note that you will not know the set-up beforehand as many set-up scripts will be used in marking process of your assignment.

Step 2

After initialisation, the peers will start pinging its two successors to see whether they are alive. The ping mechanism should define two types of messages: ping request and ping response messages. We require that the ping message should use the 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. In order for you to make sure that the ping messages are working. 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.
```

Note that 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 also that 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. We will call this a ping response message. Since a peer is going to receive both ping request and ping response messages, your message format should take that into account. When a peer receives a ping response from another peer, it should display on the terminal, so that the marker knows. 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. On the other hand, you should not send them too rarely because your examiner will on average spend 12 minutes to mark your work.

Step 3

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. 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 has the form `wxyz` where `w`, `x`, `y` and `z` are taken from the numerals `0,1,...,9`. 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. The hash function will be applied to the filename. Given the filename format defined earlier, each filename can be mapped to be an integer in the range $[0, 9999]$, as follows: The filename `wxyz` is mapped to the integer $w \times 10^3 + x \times 10^2 + y \times 10 + z$. Thus `0000` is mapped to 0, `0001` to 1, ..., `9999` to 9999. We call this the integer equivalent of the filename. To compute the hash of a file, you first determine the integer equivalent of its filename and then compute the remainder of the integer equivalent 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.

  Remark: For simplicity, we have chosen to compute the hash of the filename. In reality, a P2P system may compute the hash of the contents of the file instead.

- *File location* The location where a file is stored in a P2P network depends on the hash of 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 these messages 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.

Note: In order to simplify the assignment, you will not be working with real files. Only file request messages and response messages will be used.

We will now describe some sample steps that your assignment marker might run. These sample steps also serve to illustrate the rules specified above. The illustration is based on the network in figure 1.

The marker will pick a peer as the requesting peer and a filename. Let us assume that peer 8 and the filename `2012` have been chosen. He will then 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 and then peer 15. Both peers should decide that they do not have the file and forward the file request message to their respective successor. Both peers 10 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 should display:

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

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

In this step, the marker will randomly select one of the peers and make it 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.)

Let us assume that the marker has decided to make peer 10 depart from the network gracefully. He will 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 and we will check that.

The marker will then choose a peer as the requesting peer and a filename. The network should function correctly with the behaviour described in Step 3. For example, if the marker selects peer 4 as the requesting peer and the filename `0010`, then the responding peer should be peer 12 because peer 10 has just been departed.

Step 5 [Extended version only]

In this step, the marker will randomly select one of the xterms and kill it. This is to mimic the event of a peer leaving the network ungracefully. Let us assume that we have chosen to 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 marker will then choose a peer as the requesting peer and a filename. The network should function correctly with the behaviour described in Step 3.

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.

5 Additional Notes

- This is not a group assignment. You are expected to work on this individually.
- How to Start: Sample client and server programs have been uploaded to the Assignment webpage. The textbook contains code for a simple UDP and TCP client server application, which is a good place to start (Section 2.8).
- Language and Platform: You are free to use either C 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 the course discussion 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.

6 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, report.pdf (no more than 3 pages) describing the program design, a brief description of how your system works and your message design. 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 or more days late: NOT accepted

NOTE: The above penalty is applied to your final total. For example, if you submit your assignment 1 day late and your score on the assignment is 30, then your final mark will be 30 − 3 (10% penalty) = 27.

7 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 Internet sites 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.

8 Marking Policy

You should test your program rigorously before submitting your code. Your code will be marked using the following criteria:

- Correct compilation of all files: 2 mark
- Source code design (good structure and well commented): 6 marks
- Successful implementation of Steps 1 and 2 (Section 4): 12 marks
- Successful implementation of Step 3 (Section 4): 22 marks
- Successful implementation of Step 4 (Section 4): 14 marks
- Successful implementation of Step 5 (Section 4): 5 marks

- Report: 4 marks

We would of course like you to complete the entire assignment. You should demonstrate as much as you are able to finish to the marker. For example, in case you cannot complete Step 3 but are able to complete Steps 1, 2 and 4, you can implement these few steps and get the marks for the steps that you are able to finish.