School of Computer Science and Software Engineering
University of Wollongong
CSCI319 Distributed Systems 2014

Assignment 2 - 12 Marks
Due date: 22/Aug/2014 at 21:59 Singapore time.

Submission of the programming tasks via online submission using the submission link for assignment 2 on the site http://www.uow.edu.au/~markus/SIM/CSCI319/ . This assignment requires the submission of one ZIP file containing your source code file(s) only. A Makefile is not allowed. The submission system will confirm the submission within 5 minutes via Email.
You may submit multiple times. Only the last (most recent) submitted version will be assessed. Submissions made after the due time will be assessed as late submissions counting as full days (i.e. 5 minutes late counts as a 1 day late submission). Submissions more than 3 days late will not be assessed.
Each source code file must include a comment header which lists your full name and your student ID. You may provide additional compilation guidelines with the "Message to the Tutor" field when submitting your assignment.
Note that there is a file size limit of 2MB for submitted material.
This is an individual assignment. Plagiarism (even if it affects only part of your submission) will result in having 0 marks for all students involved! This is a University policy, and must be enforced strictly.
You are permitted to write the code in either plain C, C++, or Java. Ensure that your code will compile and run correctly within the Ubuntu Unix environment that was made available to you during the Laboratory class. The lab computer will be the standard operation environment. If your code does not compile or run successfully on a lab computer environment then this will attract zero marks. This is particularly important to those who chose to write the code in Java.
Please check regularly for updates to this assignment on the subjects' Web site. Any update on the subject's Web site is deemed to have notified all students.

Programming task 1: **(12 marks)**
The Chord system is a structured peer-to-peer network architecture realizing a scalable distributed hash table. In Chord, the peers are organized as a ring. A position within Chord which is held by a computing node is called an *index node* or simply a *peer*. Each index node has a finger table, and, in addition, holds any data items associated with its own key value and any other data items associated with keys larger than the previous index node in the Chord ring.
To succeed with this programming task, you will need to understand the Chord system very well! D**o not commence the programming task until you have developed a good understanding of the Chord system.** You may need to study the lecture notes, the relevant paragraphs in the text book, and relevant internet sources in order to acquire the level of understanding of CHORD required for this assignment. Commence the work on this assignment once you are **confident** to have understood how CHORD works.
Your task is to simulate the Chord system on a single computer as a single (non-parallelized, not multi-threaded) process. We will keep this task as simple as possible. The main objectives are:

- to understand how the nodes in Chord communicate.
- to understand how nodes are managed (added/removed) from the system.
- to understand the effects of adding and removal of index nodes.

- to understand how the finger table is computed and maintained.
- to understand how data distribution takes place in Chord.
- to understand how searching is done efficiently in Chord.
- to understand the concept of scalable code.

To keep the task as simple as possible, you are to write a single-threaded simulation (i.e. no additional threads are to be created during runtime). Follow the following guidelines:
Develop data structures which are suitable to hold required information for each peer in a CHORD system. Note that index nodes can hold local information only (I.e, an index nodes never knows of all the other index nodes). Remember that your implementation is to simulate a distributed system. **Do not make use of any global variables**!

On the basis of the data structures developed by you, write a single threaded (single process) simulation in either C, C++, or Java of the Chord system with the following functions (this is the minimum requirement). You may introduce any number of functions in addition to those stated here [1]:

`Init(n,...):` Create and initialize a new Chord system. The parameter n indicates the size of the new CHORD (the size of the hash ring). You can assume that n is a positive power-of-two value not exceeding 2^{31} (thus, n can be any one of the following values: 1,2,4,8,16,32,64,...,2^{31}). Note that in general the size of a Chord is much larger than the number of index nodes. Therefore, n can be a very large value whereas the number of index nodes can remain small. Ensure that memory is only allocated when needed (i.e. when adding an index node, when inserting a data item).
This function will also print your name and student number to the screen (print to standard output). Normally, in a real world CHORD, when a new CHORD is created than there will be at least one index node (the node which created the CHORD will become the first index node in the system), and the new node would assume a random location within CHORD. We will simulate this by creating a new index node at location 0 (zero) within the ring as part of the `Init(.)` function. The size of the finger table (for each of the peers in this new system) is computed based on the parameter n such that $k=(\log_2(n))$, where k is the size of the finger table, and $\log_2$ is the logarithm to the base of 2. Note that this equation is derived from the lecture notes which state that the CHORD is of size $n=2^k$. Thus, for example, if n=32, then k=5, or if n=4096 then k=12.
If `Init(.)` is called when there was a previously existing Chord (i.e. another Chord was generated by an earlier call to `Init(.)`. Then this function will completely remove any pre-existing CHORD from memory then initialize a new Chord system. The removing of a pre-existing CHORD requires the removal of all its associated peers from the system and the release of all allocated memory (in case of a C or C++ implementation).
The value of n is obtained from a script file (see example below).

`AddPeer(ID, ...):` Adds one new peer (an index node) to your Chord system. The ID (an integer value) of the new peer is provided, and can be assumed to be unique. You will need to create a finger table of appropriate size (note that $n=2^k$, where n is the size of the Chord, and k is the size of the finger table) for this new index node. Optional: you may also decide to associate a key-list to this Chord. A key-list is a list of IDs which are in-between the current index and the previous index node as was described in the textbook (see the section on Architectures). However, this is optional as it is possible to implement an efficient and functional CHORD without this key-list. The value for ID is obtained from a script file (see example below). If the addition of the peer was successful then this function prints to the screen (standard output) the following: "PEER <ID> inserted", where <ID> is the ID of the peer just inserted. **Note:** The addition of a new peer can require an update of the finger table of some of the other index nodes. This is a very important aspect of Chord!

`RemovePeer(ID, ...):` Removes a given peer from Chord, associates any data items that this peer held

1 Note: Functions which have three dots in their parameter list may accept any number of additional parameters. For example, the function Init(n, …), this function will accept the parameter n as well as any other parameter that you may wish to pass to this function.

with an appropriate peer in the remaining CHORD, and appropriately updates the finger tables of the remaining peers. This function removes the peer completely, not just removing its finger table. You will need to think about how the removal of an index node affects (the index table, data items, etc) of the other, remaining index nodes. Make sure that data stored at this peer is moved appropriately (to another peer) before deleting this peer. Thus, data items should not get lost when peers are removed. If the removal of a peer was successful then this function prints to the screen “PEER <ID> removed”. If this function causes the removal of the last remaining peer in the Chord, then all data items are lost (memory freed), and the program is to print “This was the last peer. The CHORD is now empty.”

`Find (ID, key, ...):` Starting from a peer identified by ID, this function searches for the peer responsible for the given hash key. You can assume that ID refers to the ID of an existing peer. The search is to be performed by using information provided by the finger tables of the peers visited during the search (as described in the lectures and the text book). Follow the algorithm as is described in the book (and illustrated in Figure 5-4) to locate the peer responsible for the key quickly. The function prints to standard output the IDs of the peers visited separated by the '>' symbol and a newline character at the end. For example, if the IDs of the visited peers was 1,7,12 then this function will print to stdout 1>7>12

`Hash (string):` Computes a hash key for a given data item (a null terminated string). The function returns a key value which is an integer within [0;n). The algorithm that must be used by this function is shown at the end of this assignment.

`Insert (ID, string, ...):` This function states that the index node ID inserts a data item into the Chord. The data item is given as a string parameter which can contain any alphanumeric character and include white spaces and tabs. The node will use the string to compute the associated hash-key. The string is then to be stored at the index node that is responsible for the computed key value. Note that this means that a peer may have to store more than one data value. Note also that the Find(.,.) may need to be called in order to locate the node responsible to hold the data item. This function will print to the screen the string “INSERTING <string> AT: “ followed by the value of the hash key computed.

`Delete (ID, string, ...):` This function states that the index node ID requests the removal of a data item identified by the parameter string from the Chord. You can assume that the data item is given as a string parameter. This function uses the hash function to compute the associated key value, then Find(.) the peer responsible for the data item, then removes the string from the memory of that peer. If successful, this function will print to the screen: “REMOVED <string> FROM: “ followed by the ID value of the peer from which the string was removed.

`Print(key, ...):` Find the peer that is responsible for the given key value. This function will then print “DATA AT INDEX NODE <ID:>” followed by a newline, then followed by the list of data items (strings) stored at this node (each string is to be separated by a newline), followed by the string “FINGER TABLE OF NODE <ID>”, a newline, the content of this nodes finger table separated by a single white-space, then a final newline.

`Read(filename):` Reads a set of instructions from a given file. The instructions are named analogous to the functions which need to be called. For example, a file may contain the following list of instructions:

```
init 32
addpeer 7
addpeer 3
removepeer 3
addpeer 12
addpeer 3
addpeer 9
removepeer 3
```

```
addpeer 17
insert 0 THIS IS A TEST
insert 7 Markus Hagenbuchner
insert 7 CSCI319
print 12
delete 0 THIS IS A TEST
print 12
print 0
print 7
print 9
print 17
```

When compiled, the program should be able to be executed at a command line and accept the file name of a file containing the instructions. Assuming that the compiled code is named CHORD, and assume that the file myfile.dat contains instructions (in the format as was shown before), then the program should be able to run by issuing the following command line parameter

```
./CHORD myfile.dat
```

this would execute all instructions within the file myfile.dat. There is one instruction per line in myfile.dat. Instructions not recognized by your program should be ignored quietly (i.e. no error message). Do not assume a maximum file size! In other words, the file may contain arbitrary number of instructions.
A sample output of the chord for the example shown above is given in Attachment 3 below.

Note1: Any of the functions mentioned above may return a value of your choosing. You are free to define the type of value that these functions return.

Note2: the task is not to implement a full Chord system. There is no need to produce a multi-threaded implementation. The aim is to:

1.) Simulate basic concepts of Chord. In particular, a deeper understanding to searching withing Chord, and the maintenance of index nodes is to be obtained.
2.) Develop an appreciation of mechanisms required for maintaining index tables.
3.) Understand the impact of the size of an index table, and the number of index nodes on the efficiency of the chord system.

Note 3: Your code will be compiled and marked using gcc, g++, or java as available within the Ubuntu virtual machine on the lab computers for CSCI319. Ensure that your code compiles and runs within the UNIX environment used in the computer lab. The correctness, functionality, and efficiency of your program will be marked. The code itself is being verified to contain your own work, and that it complies with the given task specifications, but is otherwise not marked. We will use our own scripts to test your program. Do not submit your own script file(s).

Note 4: While the code will not be marked, the lecturer and the tutors will look at the code in order to identify whether the task was addressed correctly. Remember that the task is to simulate Chord. This also means that no single peer has complete information. For example, this means that a `search` needs to be carried out when inserting or removing a peer from Chord. We will also look at the efficiency of your code.

Note 5: Your code is not to produce outputs other than those specified above. This means that you do not print to stdout, stderr, or any other output stream any debugging information, or any other information not required by this assignment. A violation to this will attract penalty marks!

Note 6: The completeness, correctness and in particular the speed (the efficiency) of your code will influence the marks that you can earn for this task. We will test the scalability of your code. For this, we will test how well yoru code can handle an increasingly large CHORD, increasingly large number of index nodes, increasingly large number of data items, and increasingly large number of peers and data items being added and removed.

Note 7: We will check the time required by your code to process given script files, and we will test the correctness of the finger tables and the correct storage location of any data item stored in the Chord.

Late submissions will be marked with 25% deducted for each late day. Submissions 4 or more days late will not be marked.

Attachments:

Attachment 1: Algorithm of a hash function for character strings:
Given a character string of arbitrary content and length, then the hash algorithm is as follows:

```
BEGIN Hash (string)
  UNSIGNED INTEGER key = 0;
  FOR_EACH character IN string
    key = ((key << 5) + key) ^ character;
  END FOR_EACH
  RETURN key;
END Hash
```

Note that '<<' is a bit shift operation which shifts bits to the left. Hence, key<<5 shifts the bits in key by 5 positions to the left. Note also that ^ corresponds to the XOR operation, and that "character" corresponds to the ASC-II value of the character, and that the terminating NULL character is not considered part of a string.

Attachment 2: The Chord system, an example (from the text book):

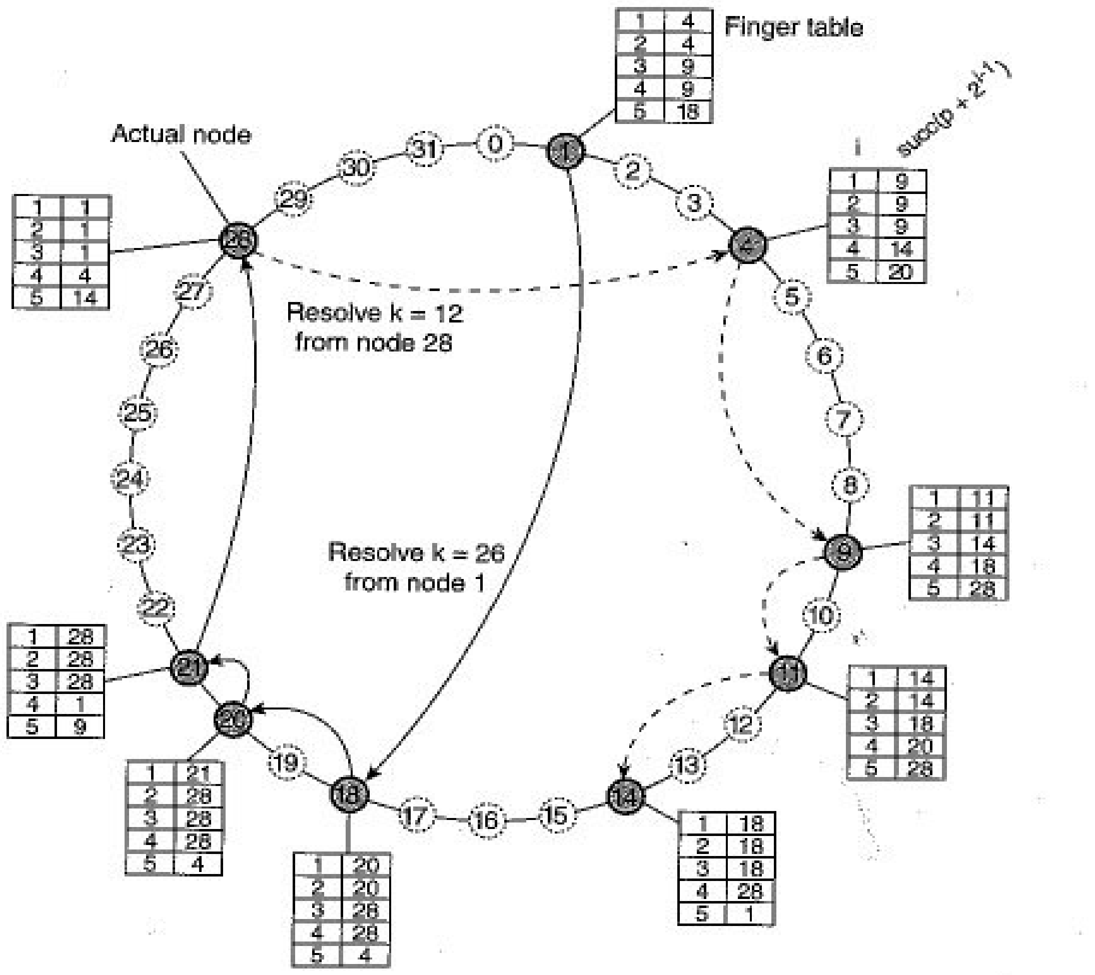

Figure 5-4. Resolving key 26 from node 1 and key 12 from node 28 in a Chord system.

Attachment 3: Sample output (for the sample script shown above):
This is a guide only. Depending on your implementation, the output of your code may differ from what is shown here.

```
Markus Hagenbuchner
1234567
0>7
PEER 7 inserted
7>0>3
PEER 3 inserted
PEER 3 removed
7>12
PEER 12 inserted
12>0>3
PEER 3 inserted
3>7>9
PEER 9 inserted
PEER 3 removed
9>17
PEER 17 inserted
INSERTING THIS IS A TEST AT 11
17>0
0>9>12
INSERTING Markus Hagenbuchner AT 19
17>7
7>17>0
INSERTING CSCI319 AT 1
17>7
7>0>7
17>7>12
DATA AT INDEX NODE 12:
THIS IS A TEST
FINGER TABLE OF NODE 12:
17 17 17 0 0
17>0
0>9>12
DELETED THIS IS A TEST FROM 12
17>7>12
DATA AT INDEX NODE 12:
FINGER TABLE OF NODE 12:
17 17 17 0 0
17>0
DATA AT INDEX NODE 0:
Markus Hagenbuchner
FINGER TABLE OF NODE 0:
7 7 7 9 17
17>7
DATA AT INDEX NODE 7:
CSCI319
FINGER TABLE OF NODE 7:
9 9 12 17 0
```

```
17>7>9
DATA AT INDEX NODE 9:
FINGER TABLE OF NODE 9:
12 12 17 17 0
DATA AT INDEX NODE 17:
FINGER TABLE OF NODE 17:
0 0 0 0 7
```