EP2400 - Network Algorithms
Algorithms for Distributed Hashtables

Amir H. Payberah Tallat Mahmood Shafaat
amir@sics.se tallat@sics.se

November 17, 2008

Contents

1 Project Outline

The aim of this project is to compare three algorithms for *Distributed Hash Tables (DHT)* from various aspects. The evaluations will be carried out in SICSIM [5], an event-based simulator for peer-to-peer systems.

You will be required to compare Chord [1], Chord# [2] and Kademlia [3], which are introduced in the tutorials. The evaluations have to be done to compare the following metrics:

1. Bandwidth consumption
2. Lookup forwarding load
3. Average lookup hops
4. Failed lookups under churn

You are expected to run extensive simulations for each of the above and write a report on the findings. To get an idea of doing evaluations, you can have a look at PVC [4]. The algorithms have to be compared for various network sizes, churn levels and identifier distributions. The final report submitted should be considered as a prototype research paper.

It is recommended that you read the simulator document before starting your assignment.

2 Metrics and Network Settings

This section shows the definition of metrics that should be evaluated in this assignment and also different network settings.

The following metrics are important for us:

1. **Bandwidth consumption:** To measure the overhead of maintenance traffic generated by each DHT, we compare bandwidth consumption. For the assignment, we define the bandwidth consumption as the number of maintenance messages sent by each peer. In Chord and Chord#, the maintenance traffic is messages send by a peer when it calls the successor stabilization and fixes the finger list, whereas in Kademlia, it is the messages sent when a peer refreshes its kbucket list. Intutively, the more often a peer refreshes its routing table during the simulation time, the more the bandwidth comsumption.

2. **Lookup forwarding load:** To measure the load of queries forwarding, we compare the number of lookup messages that each peer has to forward. For example the lookup messages that a peer receives from other peers and forwards it to others during simulation time.

3. **Lookup hops:** To measure the latency of reaching from any peer in the system to any other peer, we measure the number of hops that a lookup message passes through to find the result.

4. **Failed lookups under churn:** The number of unsuccessful lookups, i.e. the lookup comes across a dead node.

The mentioned metrics have to be evaluated under various setting defined by the following parameters:

1. **Network size:** Different number of nodes in the system.
2. **Churn level:** Having different levels of churn. Churn in a system refers to joins and failures of peers. Churn level is defined by (i) the ratio of the join events to the failure events, and (ii) their inter-arrival time. SICSIM uses exponential distribution to distribute the inter-arrival time of events. There are three levels of churns:
   (a) **No churn:** In this model, the network size is static i.e. no joins and failures.
   (b) **Low churn:** In this model, peers join and fail continuously and the mean inter-arrival time of events is relatively high, e.g. 10 time units.
   (c) **High churn**: In this model, peers join and fail continously and the mean inter-arrival time of events is relatively low e.g. 5 time units. Note that the lower the inter-arrival time of events, the higher the number of events in a period of time, thus producing more churn.
3. **Different identifier distribution:** Identifier distribution refers to the distribution according to which nodes are assigned identifiers. The purpose of varying the distribution of identifiers is to see it effects on the three DHTs. You have to experiment with two identifier distributions:
   (a) **Uniform distribution:** In uniform distribution, the nodes' identifiers are distributed uniformly over the identifier range.
   (b) **Skewed distribution:** In skewed distribution, the identifier range is divided into a number of clusters and the peers are placed in these clusters with a user defined probability (Read more about skewed distribution in the SICSIM document).

3 Requirements

Before starting the assignment, it is good to find a global view to the whole assignment.

3.1 Big Picture

This assignment has three main parts:

1. Implementing Chord, Chord# and Kademlia.
2. Analyzing the behaviour of each DHT in different configurations.

3. Comparing the implemented DHTs together in different scenarios.

Along with this document, you are given one Java package for each DHT:

- `sicsim.dht.chord`
- `sicsim.dht.chordsharp`
- `sicsim.dht.kademlia`

Each package contains two classes: (i) `Peer`, a class for implementing the behaviour of a peer, and (ii) `DHTMonitor`, a class to verify the structure of DHT, e.g. to see the correctness of ring in Chord.

3.2 Testing the Implementation

There are some local variables defined in the provided `Peer` classes. These variables are used to store the properties of each peer. For example, `Peer` in Chord uses `succ` as a variable to store the successor of the peer and `pred` to store the predecessor. Since `DHTMonitor` uses these variables to verify the structure of the DHT, it is strongly recommended to use these variables to store the peer information. If you change the name of these variables or use some other structure to store the local information of a peer, you should modify the `DHTMonitor` as well. Note that you can add as many variables as you like.

Your implementation should work correctly in case of having churn or there is no churn in the system.

3.3 Measurements and Gathering Statistics

As mentioned in Section 2, there are four important metrics in this assignment. Class `Peer` provides some variables to gather statistics for these metrics:

- `failedLookup`: This is a `static` variable that counts all the failed lookups in system during the simulation time.
- `hopCount`: It is a `static` variable that shows the total hop counts in all lookups.
- `bwConsumption`: It is also a `static` variable that measures the traffics generated from all peers during the simulation time.
- `lookupForwardLoad`: This is a variable that keeps the lookup traffic in each peer. The result of this variable is useful only when there is no churn in the system.

`DHTMonitor` uses these variables to show the collected statistics. If you want to measure more metrics, it is enough to define a variable for that metric in `Peer` and modify the `verify` method of `DHTMonitor` to read this variable and prints out its value.

The peers gather the statistics when the first peer in the system receives a signal from the simulator. The simulator sends signal to peers by calling the `signal` method of `Peer`. The signals are sent according to the signal scenario that a user defines in scenario file. In task 2 and task 3, you should implement the `signal` method, such that the peer performs a lookup for a random key. The lookup service should be called periodically. Assume the period of this service is 50 time unit. To make it more clear, the implementation of `signal` method for Chord is given to you, but you should implement it for Chord# and Kademlia.

By reading the following scenario, the simulator sends signal 1 to 512 random selected peers. Note that in signal scenario the peers are selected randomly, so a peer may be selected more than once. Read more about scenarios in Section 6 and SICSIM document.

```
type:       signal
count:      512
interval:   1
signal:     1
```

One of the important metrics is the number of failed lookup. A lookup can be failed in two cases:

1. **Time out:** If the peer does not receive the reply of a key lookup.
2. **Wrong reply:** If the reply of a key lookup is wrong.

As mentioned earlier each peer performs the key lookup service periodically every 50 time unit. To check the time out, before starting the next lookup, each peer should check whether it has received the reply of previous lookup or not. If they have received the reply, they should ask simulator about the correctness of the result. To do so, the peers can ask from `DHTMonitor` about it, by calling its `checkLookup` method, e.g. `((DHTMonitor)this.monitor).checkLookup(host, id)`. This method returns a boolean as result. In Chord and Chord# this method gets the key and the responsible node as input, while in Kademlia instead of getting responsible node, gets a list of responsible nodes.

Note1. To do the measurements in tasks 2 and 3, define the signal scenario such that the `count` equals to the number of nodes in the system, and set `interval` to 1.

Note2. In all the measurements, set the `SIM_TIME` to 25000, and `MAX_NODE` to 5000.

Note3. Set the time of delay for the stabilization part to 5000 time unit (See Section 3.4).

3.4 Typical Experiment

A typical experiment comprises of three sequential parts, (i) network creation, (ii) network stabilization and (iii) taking measurements. In the first part, the overlay is constructed by letting nodes join the system. This is done until the desired network size is achieved. In the second part, the overlay is allowed to stabilize. During this period, peers continuously perform overlay maintenance. The delay for this part should be enough so that the peers have created their routing tables/buckets. Now, the network is ready for the actual experiment. In the third part, you have to trigger events as per requirements of the experiment and gather statistics that have to be reported. Section 6 contains some sample scenarios for experiments.

4 Your Tasks

In this section you find the detail information about each task.

4.1 Task One

Your first task is to implement the DHTs in SICSIM.

▷ **Chord** and **Chord#**
Section 7 contains the pseudo code for Chord (Algorithms 1 and 2) and Chord# (Algorithm 3). These pseudo codes are presented in RPC style, while you have to implement them in message-passing model. In the message-passing model, peer p instead of calling a remote procedure of a peer q directly, should send a message to q, and q after receiving that message should process it and send another message back to p. For details of how to implement an RPC call in message-passing model, please consult `samples.helloworld` in SICSIM source code.

After a node has joined the overlay i.e. has set its successor, it should join the `overlay` structure. The reason is that peers can only see the peers who have joined the `overlay` structure.

To implement this DHTs, assume the peers have access to the failure detector for their successors and predecessors. The failure detector helps the peers to detect the failure of peers of interest. Note that peers can not use the failure detector for their finger list.

Note. The simulator provides some `static` methods in `MathMisc` class, which can be helpful in implementation of Chord and Chord#:

1. `belongsTo`: Checks whether $id \in (start, end]$ is correct or not.
2. `belongsTonn`: Checks whether $id \in (start, end)$ is correct or not.

How to verify your implementation: The Chord and Chord# implementation should satisfy the following properties in the absence of churn: First, the constructed ring should be correct, which means that the successor and predecessor of each peer should point to the correct peer. Second, the finger list and the successor list should be filled with the proper peers in the overlay. Finally, a key lookup should return the correct result. To verify the above properties, the provided code contains a class called `DHTMonitor`. After the simulation ends, this class checks the properties via the `succ`, `pred`, `fingers` and `succList` variables of `Peer` and reports whether they are correct or not. You can modify the `verify` method of this class if you want to check additional properties of the constructed DHT. To test your implementation, execute the first two parts explained in section 3 and check the output.

▷ **Kademlia**

Section 7 contains the pseudo code for Kademlia (Algorithms 4 and 5). These pseudo codes are also presented in RPC style, while you have to implement Kademlia in message-passing model. In Kademlia, the peers do not have access to failure detector and they can not use failure detector to detect the failure of peers of interest.

Besides `Peer` and `DHTMonitor`, there are two other classes in the provided Kademlia package that implement `KBucket` and `KBucketList`. In Kademlia, each peer maintains the information of other peers in a structure called *kbucket list*. The kbucket list contains a list of *kbuckets* [3]. `KBucketList` is a class that implements the kbucket list. `DHTMonitor` uses the `kbucketList` variable, which is an instance of `KBucketList`, to verify the structure of Kademlia. If you want to use another structure to store the information of kbucket list, you should modify `DHTMonitor` accordingly. Following are some of the important methods of `KBucketList`:

- `add`: Adds a node into the kbucket list.
- `remove`: Removes an existing node from the kbucket list.
- `update`: Update the time of an existing node in the kbucket list.
- `contains`: Shows whether a node exists in the kbucket list or not.
- `hasSpace`: Shows if there is enough space in a kbucket to add a new node.
- `getRandomNodes`: Returns a number of random nodes (specified by `num` as input argument), from the specific kbucket. The kbucket is specified by its index in the kbucket list or by the node ID.
- `oldestNode`: Returns the oldest node in a kbucket. The specific kbucket is specified by a node ID as input argument.
- `getClosestNodes`: Returns a number of closest nodes to a node ID from the kbucket list. The number is specified by `num` as input argument.

How to verify the implementation: The kbucket list of each peer should be complete and updated properly according to the peers in the overlay. In the absence of churn, a key lookup should return the correct results. To verify the correctness of overlay, `DHTMonitor` initiates a key lookup in all the peers of the system and if all of them reply correctly the structure will be accepted.

4.2 Task Two

The main idea of task 2 is to analyze the behaviour of each DHT in different configurations. According to the PVC [4], there are some parameters in each DHT that influence the behavior of that DHT. In this task you are supposed to change the defined parameters and show their effects on each DHT.

For this task assume the system has the following setting:

- **Network size:** 512 nodes.
- **Churn level:** Low churn.
- **Identifier distribution:** Uniform distribution.

Note. As mentioned in 3.4, you should first construct the overlay, and provide a delay to let the system be stabilized and then do the measurements.

▷ **Chord** and **Chord#**
There are three main configuration parameters in Chord and Chord#:

- Number of successors (successor list size).
- Successor stabilization interval
- Finger list stabilization interval

As mentioned in Chord paper [1] to increase the robustness of DHT, each peer maintains a successor list. Increasing the size of successor list, increases the robustness of Chord overlay. Here, we assume the size of successor list is $log_2 N$, and by considering $N = 512$ nodes, the size of successor list equals 9. Assume the successor stabilization and the finger list stabilization are synchronized, and they are called together. Set the stabilization interval to 20, 50, 80 time unit and measure the following metrics:

1. **Bandwidth consumption:** Draw a plot such that the X-axis shows the stabilization interval and the Y-axis shows the total bandwidth consumption in all peers.
2. **Failed lookups:** Draw a plot such that the X-axis shows the stabilization interval and the Y-axis shows the average failed lookup during the simulation time.

Scenario 4 in Section 6 is a sample scenario you can use to do the measurements for this task. To use this scenario for Chord# substitute `sicsim.dht.chord` with `sicsim.dht.chordsharp`.

▷ **Kademlia** Followings are the main configuration parameters of Kademlia:

- Nodes per entry (k)
- Parallel lookups (α)

- Number of ID returned
- Stabilization interval

k specifies the size of each kbucket, and α specifies the number of parallel lookups. The next parameter shows the number of IDs that a peer returns in reply to a request, and the last parameter specifies the refreshing interval of kbucket list. Here, assume the size of k is 5, each peer returns 5 IDs in replay and the stabilization interval is 200 time unit. Set the α to 1, 3 and 5 and measure the following metrics:

1. **Lookup hops:** Draw a plot such that the X-axis shows the values of α and the Y-axis shows the average lookup hops during the simulation time.
2. **Failed lookups:** Draw a plot such that the X-axis shows the values of α and the Y-axis shows the average failed lookup during the simulation time.

You can use Scenario 4 in Section 6 by replacing `sicsim.dht.chord` with `sicsim.dht.kademlia`.

4.3 Task Three

In this task you should compare the implemented DHTs together in different network setting.

Assume the following configuration for Chord and Chord#:

- Number of successors = $log_2 N$
- Successor stabilization interval = 50
- Finger list stabilization interval = 50

And the following configuration for Kademlia:

- Nodes per entry (k) = 5
- Parallel lookups (α) = 3
- Number of ID returned = 5
- Stabilization interval = 200

▷ **Setting 1**
Assume the system has the following setting:

- **Network size:** Set the number of nodes to 128, 512 and 1024.
- **Churn level:** No churn.
- **Identifier distribution:** Uniform distribution.

You should measure the following metrics:

1. **Lookup forward load:** Draw a plot such that the X-axis shows the number of peers in the system, and the Y-axis shows the average lookup forward load. Draw the result of measurements of Chord, Chord**#** and Kademlia in one plot, as shown in Figure 1. Note that the numbers in this plot are randomly generated.

2. **Lookup hops:** Draw a plot such that the X-axis shows the number of peers in the system, and the Y-axis shows the average number of lookup hops for each DHT.

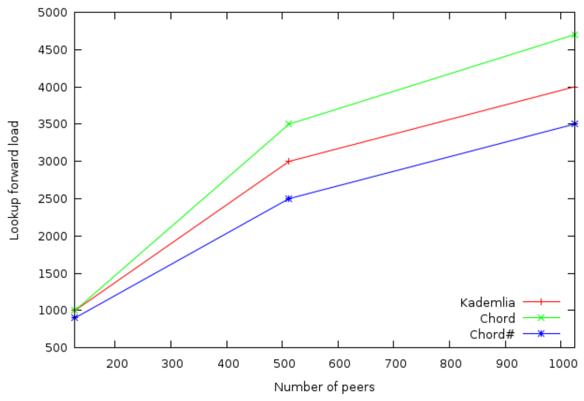

Figure 1: A sample plot

You can use Scenario 4 for this task, but you should remove the churn event from it (sub-scenario 4). This scenario is for 512 nodes, and you should change the number of nodes according to your setting.

▷ **Setting 2**
Assume the system has the following setting:

- **Network size:** 512 nodes.
- **Churn level:** No churn.
- **Identifier distribution:** Set the identifier distribution to uniform distribution and skewed distribution.

You should measure the following metrics:

1. **Lookup forward load:** For each identifier distribution, draw a separate plot such that the X-axis shows the lookup forward load, and the Y-axis shows the number of peers.

2. **Lookup hops:** Draw a plot such that the X-axis shows the identifier distribution and the Y-axis shows the average lookup hops.

Note. To configure the simulator to use the skewed distribution, you should modify `sicsim.conf`. Set `SKEWED` to `true`, set `NUM_OF_CLUSTER` to 2 and `PROB_OF_CLUSTER` to 0.7.

▷ **Setting 3**
The system setting is as follows:

- **Network size:** 512 nodes.
- **Churn level:** Set it to low churn and high churn.
- **Identifier distribution:** Uniform distribution.

You should measure the following metrics:

1. **Lookup hops:** Draw a plot such that the X-axis shows the churn levels, and the Y-axis shows the average number of lookup hops for each DHT.
2. **Failed lookup:** Draw a plot such that the X-axis shows the churn levels, and Y-axis shows the average number of failed lookups for each DHT.

5 Things To Deliver

The assignment can be done individually or in group of two. You should hand in the following materials in a zip file. Name the file according to the following convention: `FirstnameFamilyname1_FirstnameFamilyname2.zip`. This file should include:

1. The source code of the implemented DHTs in different packages.
2. A report that contains the drawn plots and a brief explanation for each one. It is recommended to use Gnuplot [6] to draw the plots, but you are free to use any other applications.

6 Sample Scenarios

Here you can find some sample scenarios to test the implemented DHTs. Scenario 1 shows a sample scenario when there is no churn, and scenario 2 shows a scenario that considers churn in system. The following scenarios can be used for all DHTs simply by replacing `sicsim.dht.chord` to `sicsim.dht.chordsharp` or `sicsim.dht.kademlia`.

```
### Scenario 1 ###
type:               monitor
monitor:            sicsim.dht.chord.DHTMonitor
---
### Scenario 2 ###
type:               lottery
peer:               sicsim.dht.chord.Peer
link:               sicsim.network.links.ReliableLink
count:              512
interval:           10
join:               1
leave:              0
failure:            0
---
### Scenario 3 ###
type:               delay
delay:              5000
```

Scenario 1. Sample scenario when there is no churn in system.

```
### Scenario 1 ###
type:               monitor
monitor:            sicsim.dht.chordsharp.DHTMonitor
---
### Scenario 2 ###
type:               lottery
peer:               sicsim.dht.chordsharp.Peer
link:               sicsim.network.links.ReliableLink
count:              512
interval:           10
join:               1
leave:              0
failure:            0
---
### Scenario 3 ###
type:               delay
delay:              5000
---
### Scenario 4 ###
type:               lottery
peer:               sicsim.dht.chordsharp.Peer
link:               sicsim.network.links.ReliableLink
count:              512
interval:           10
join:               1
leave:              0
failure:            1
```

Scenario 2. Sample scenario when there is low churn in system.

As you see, Scenario 1 contains three sub-scenarios. The first sub-scenario is used to define the Monitor, the second one is a lottery event, which defines a scenario with no churn, in which 512 nodes join the system with an exponential inter-arrival time with mean 10 time unit, and the last sub-scenario is a delay scenario. Having the delay scenario after no-churn-scenario is to give time to the peers in the overlay to be stabilized. Scenario 2 shows a low-churn-scenario. As you can see, the low churn sub-scenario is defined after the delay sub-scenario.

In task 2 and task 3, the peers should initiate the lookup service. The initiation is done by sending a signal to peers. To do so, you should add a signal sub-scenario exactly after delay sub-scenario (Scenario 3).

```
### Scenario 1 ###
type:                monitor
monitor:             sicsim.dht.kademlia.DHTMonitor
---
### Scenario 2 ###
type:                lottery
peer:                sicsim.dht.kademlia.Peer
link:                sicsim.network.links.ReliableLink
count:               512
interval:            10
join:                1
leave:               0
failure:             0
---
### Scenario 3 ###
type:                delay
delay:               5000
---
### Scenario 4 ###
type:                signal
count:               512
interval:            1
signal:              1
```

Scenario 3. Sample scenario to show the definition of signal.

Here in Scenario 3, after constructing the Kademlia overlay and waiting for a while to let the system to be stabilized, we define the signal sub-scenario. In this sub-scenario we ask simulator to send signal number 1, which should defined in `signal` method of `Peer`, to 512 random peers with an exponential inter-arrival time with mean 1 time unit. Note that the a peer can be selected more than once.

```
### Scenario 1 ###
type:                monitor
monitor:             sicsim.dht.chord.DHTMonitor
---
### Scenario 2 ###
type:                lottery
peer:                sicsim.dht.chord.Peer
link:                sicsim.network.links.ReliableLink
count:               512
interval:            10
join:                1
leave:               0
failure:             0
---
### Scenario 3 ###
type:                delay
delay:               5000
---
### Scenario 4 ###
type:                signal
count:               512
interval:            1
signal:              1
---
### Scenario 5 ###
type:                lottery
peer:                sicsim.dht.chord.Peer
link:                sicsim.network.links.ReliableLink
count:               100000
interval:            10
join:                1
leave:               0
failure:             1
```

Scenario 4. Scenario for measurements of Chord and Chord# in task 2.1. In case of using this scenario for testing the implementations in task 1, change the number of events in last sub-scenario from 100000 to the number of peers in the system, which is 512 here.

7 DHTs' Pseudo Code

In this section you can find the pseudo code of the target DHTs. These algorithms are shown in RPC style, whereas you should implement them in message-passing model.

7.1 Chord

Algorithms 1 and 2 show the RPC style pseudo code of Chord.

Algorithm 1 Chord Pseudo Code

```
// create a new Chord ring.
n.create() {
 predecessor = nil;
 successor = n;
}
//----------------------------------------------------------
// join a Chord ring containing node u.
n.join(u) {
 predecessor = nil;
 successor = u.find_successor(n);
}
//----------------------------------------------------------
// called periodically.  verifies n's immediate successor,
// and tells the successor about n.
n.stabilize() {
 x = successor.predecessor;
 if (x ∈ (n, successor))
  successor = x;
 successor.notify(n);
}
//----------------------------------------------------------
// n thinks it might be our predecessor.
n.notify(u) {
 if (predecessor is nil or u ∈ (predecessor, n))
  predecessor = u;
}
//----------------------------------------------------------
// called periodically.  refreshes finger table entries.
// next stores the index of the next finger to fix.
n.fix_fingers() {
 next = next + 1;
 if (next > m)
  next = 1;
 finger[next] = find_successor(n + $2^{next-1}$);
}
//----------------------------------------------------------
// called periodically.  checks whether predecessor has failed.
n.check_predecessor() {
 if (predecessor has failed)
  predecessor = nil;
}
```

Algorithm 2 Chord Pseudo Code - Cont.

```
// ask node n to find the successor of id
n.find_successor(id) {
 if (id ∈ (n, successor])
  return successor;
 else {
  m = closest_preceding_node(id);
  return m.find_successor(id);
 }
}
//------------------------------------------------------------
// search the local table for the highest predecessor of id
n.closest_preceding_node(id) {
 for i = m downto 1 {
  if (finger[i] ∈ (n, id)) {
   return finger[i];
  }
 return n;
}
```

7.2 Chord#

Chord# algorithm is similar to Chord with some changes in `fix_finger` method. Algorithm 3 shows the `fix_finger` method of Chord#.

Algorithm 3 Chord# Pseudo Code - `fix_finger` method

```
// called periodically.  refreshes finger table entries.
// next stores the index of the next finger to fix.
n.fix_fingers() {
 finger[0] = successor;
 next = next + 1;
 if (next ≥ m)
  next = 1;
 finger[next] = finger[next - 1].finger[next - 1];
}
```

7.3 Kademlia

Algorithms 4 and 5 show the RPC style pseudo code of Kademlia.

Algorithm 4 Kademlia Pseudo Code

```
// create a new Kademlia overlay.
n.create() {
 for i = 1 to m
  kbucket_list[i] = nil;
}
//----------------------------------------------------------
// join a Kademlia overlay containing node u.
n.join(u) {
 add u to its kbucket_list;
 kclosest = u.find_node(n);
 update its kbucket_list with the kclosest;
}
//----------------------------------------------------------
// ask node n to find k nodes it knows closest to id.
n.find_node(id) {
 return k nodes it knows about closest to id;
}
//----------------------------------------------------------
// ask node n to find the value of key,
// or find k nodes it knows closest to key.
n.find_value(key) {
 if contains key
  return the value;
 else
  return k nodes it knows about closest to key;
}
//----------------------------------------------------------
// probes a node too see if it is online.
n.ping() {
 return true if it is online;
}
//----------------------------------------------------------
// store a (key, value) pair for later retrieval.
n.store(key, value) {
 store the (key, value) pair for later retrieval;
}
```

Algorithm 5 Kademlia Pseudo Code - Cont.

```
// called periodically to refresh the kbucket_list.
n.stabilize() {
 for i = 1 to m {
  u = pick a random node from kbucket_list[i];
  kclosest = lookup(u);
  update kbucket_list by kclosests;
 }
}
//-----------------------------------------------------------
// search to find k closest node to id.
n.lookup(id) {
 kclosest = pick $\alpha$ nodes from the closest non empty kbucket or if
 that bucket has fewer than $\alpha$ entries, just take the $\alpha$ closest nodes
 it knows of;

 // loop until does not receive better result as the closest nodes to
 id.
 do {
 result = $\emptyset$;
  temp_list = kclosest;
  pick $\alpha$ node from temp_list;
  for i = i to $\alpha$
   // block till receive the result of all queries or time out.
   result = $\alpha$[i].find_node(id) $\cup$ result;
  kclosest = the k closest nodes to id in result;
 } while (kclosest $\neq$ temp_list)

 // now the k closest nodes are stored in kclosest.
 // query the peers in kclosest that have not been queried.
 result = $\emptyset$;
 for i = 1 to k {
  if kclosest[i] has not already queried
   // block till receive the result of all queries or time out.
   result = kclosest[i].find_node(id) $\cup$ result;
 }

 kclosest = the k closest nodes to id in (result $\cup$ kclosest);
 return kclosest;
}
```

References

[1] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions on Networking (TON)*, 11(1):17.32, 2003.

[2] Thorsten Schütt and Florian Schintk, Alexander Reinefeld. Structured Overlay without Consistent Hashing: Empirical Results. *In Proceedings of the Sixth Workshop on Global and Peer-to-Peer Computing (GP2PC'06)*

[3] P. Maymounkov, D. Mazieres. Kademlia: A Peer-to-Peer Information System Based on the XOR metric. *In Proceedings of IPTPS'02, Lecture Notes in Computer Science (LNCS), pages 53-65, London, UK, 2002. Springer-Verlag.*

[4] J. Li, J. Stribling, R. Morris, M. F. Kaashoek, T. M. Gil A performance vs. cost framework for evaluating DHT design tradeoffs under churn [PDF] *In Proceedings of the 24th IEEE Infocom, Mar 2005.*

[5] SICSIM - http://www.sics.se/∼amir/sicsim

[6] Gnuplot - http://www.gnuplot.info/