NS3 - Chord

Cooperative Mirroring and Time-shared Storage

Lei “Ricky” Jin
Pragadheeshwaran Thirumurthi
Steven Tsiang

Table Of Contents

Abstract — **We implemented part of a Peer-to-Peer Networking system to gain further insight into the domain of Distributed Hash Table Peer-to-Peer Systems (DHT). The popularity of Peer-to-Peer (P2P) networks and systems have been a relatively recent development in the field of computer science. While many of the original P2P were popular and successful in transmitting data between peers, most of them were hybrid systems that depended on a centralized host. In an attempt to avoid the downfalls accompanying such a setup, researchers at MIT have come up with a completely decentralized solution: Chord, a scalable P2P lookup service for internet applications. Instead of bearing a single point of failure, among other flaws, chord is designed to be completely distributed, with no node more important than any other. In addition, Chord aims to tackle many of the fundamental problems in today's networking protocols, such as load balance, scalability, and availability. Using NS-3, a discrete-event network simulator, we were able to successfully implement the Chord protocol as well as many of its key functions.**

1. Introduction into Chord

Peer-to-Peer networks have proved to be incredibly useful and popular in today's Internet-driven culture. Although peer-to-peer networks have been around for a long time, it wasn't until 1999, with the introduction and tremendous popularity of Napster, that the technology began to gain momentum. However, with Napster also came the shortcomings of P2P networks, mainly a central bottleneck that impeded search speed and scalability. In an effort to solve some of these flaws, in 2001, a team of engineers at MIT developed the Chord protocol. The key to the robustness and scalability of the protocol was in its simplicity. The Chord protocol supports just a single operation: given a key, the protocol maps the key to a node. And in order to ensure that the nodes are widely distributed, a consistent hashing function, SHA-1, is used to assign keys to the Chord nodes. All other functionality is variable with the application utilizing the Chord protocol. Because of this, Chord exhibits features that distinguish it from many other P2P lookup protocols. Unlike Napster's lookup protocol, which was bogged down from a central server, Chord features provable performance and correctness.

Chord itself is a structured network, with a network topology in the shape of a ring. Because of the cyclic nature of the Chord network, the protocol guarantees that the target node will eventually be found. In addition, the ring-like nature of the network topology ensures that there is minimal overhead in routing information on each node. Instead of keeping information on every node in the network, a single node will only be needed to carry about O(log N) other nodes in order for the protocol to work properly. Each node keeps a finger table, essentially a routing table with nodes at specific distances from itself, that allows itself to find nodes quickly. Because of the consistent hashing of the node ID's, the entries in the finger table should not be too uneven. Consistent hashing is also the primary reason that nodes are able to join and leave the network without disrupting the whole network.

2. Porting Chord from NS-3.7 to NS-3.13

2.1 Installing NS-3.13 and NS-3.7 Chord in Ubuntu

1. Update Ubuntu to the most current version
    a. sudo apt-get update
    b. sudo apt-get upgrade
2. Make sure to have the proper build tools
    a. sudo apt-get install build-essential g++ python
3. Make sure to have Mercurial version control system installed
    a. sudo apt-get install mercurial
4. Retrieve the NS-3.13 source using Mercurial
    a. hg clone http://code.nsnam.org/ns-3-dev/
5. Build NS-3.13, using waf tool inside **ns-3-dev/**
    a. ./waf configure
    b. ./waf build
6. In a separate directory, retrieve the NS-3.7 Chord source using Mercurial
    a. hg clone http://code.nsnam.org/gillh/ns-3-chord/
7. Build NS-3.7 Chord, using waf tool inside **ns-3-chord/**
    a. ./waf configure
    b. ./waf build

2.2 Porting Steps (Tested on the latest ns-3-dev version - 3/19/2012)

1. Now that both NS-3.13 and NS-3.7 Chord are built successfully, time to copy over the core **chord-ipv4/** source folder from **ns-3-chord/applications/src/** to **ns-3-dev/src/**
2. Also copy over two extra helper source files **chord-ipv4-helper.cc** and **chord-ipv4-helper.h** from **ns-3-chord/src/helper/** to **ns-3-dev/src/chord-ipv4/**
3. Modify the waf build script **wscript** in **ns-3-dev/src/chord-ipv4/** as follows:
    a. Replace the create module line with new dependency modules:
        i. ***from:*** module = bld.create_ns3_module('chord-ipv4', ['node', 'internet-stack', 'csma'])
        ii. ***to:*** module = bld.create_ns3_module('chord-ipv4', ['core', 'network'])
    b. Replace the new task generation line with new syntax:
        i. ***from:*** headers = bld.new_task_gen('ns3header')
        ii. ***to:*** headers = bld.new_task_gen(features=['ns3header'])
    c. Add both helper header and source files to the build list:
        i. module.source = [..., 'chord-ipv4-helper.cc', ...]
        ii. headers.source = [..., 'chord-ipv4-helper.h', ...]
4. Modify the Chord helper header file **chord-ipv4-helper.h** in **ns-3-dev/src/chord-ipv4/** as follows:
    a. Modify two include directives:

i. ***from:*** #include "application-container.h" *and* #include "node-container.h"

ii. ***to:*** #include "ns3/application-container.h" and #include "ns3/node-container.h" respectively

5. Rebuild NS-3.13, using waf tool inside **ns-3-dev/**
    c. ./waf configure
    d. ./waf build

2.3 Running chord-run.cc (Tested on the latest ns-3-dev version - 3/19/2012)

1. In order to test the new Chord library in NS-3.13, we should also copy over a run script **chord-run.cc** from **ns-3-chord/examples/chord-run/** to **ns-3-dev/scratch/**
2. Copy (non-clobber) all the header files from **ns-3-chord/build/debug/ns3/** to **ns-3-dev/build/ns3/**. We basically just need all the user-made module-aggregator header files.
3. Modify the waf build script **wscript** in **ns-3-dev/** to include the libcrypto linkflag as follows:
    a. Add the following line with proper spacing anywhere after which the environment variable 'env' is defined in the 'configure' function (e.g. line 309)
        i. env.append_value('LINKFLAGS','-lcrypto')
4. Rebuild NS-3.13, using waf tool inside **ns-3-dev/**
    a. ./waf configure
    b. ./waf build
5. To test if everything works correctly, run **chord-run.cc** in **ns-3-dev/**:
    a. ./waf --run chord-run

3. Chord-Run.cc vs. Chord-Main.cc

3.1 Limitations of chord-run.cc

There are several shortcomings of the example script **chord-run.cc** that we like to point to out, which have come up during initial benchmark tests on chord's performance.

First, we've noticed that almost half of the script is dedicated to parsing the command-line or Chord input command-script that for example tells the NS-3 simulator to insert some nodes, insert some content, and retrieve some content given content name etc. However, after running multiple tests with different sized Chord input command-scripts as well as different number of nodes, we have noticed that on many runs, NS-3 tends to crash with segmentation fault mid-way through the simulation. After digging deeper into the script code, we've noticed that the author(s) used pthread_create to fork off threads to create multiple ChordRun objects. Upon further inspection, each of those ChordRun objects schedule tasks using the NS-3 simulator to parse through and execute the input commands. First thing that comes to mind is race conditions and since NS-3 is not multiprocessing friendly[2], this is huge problem.

Second, various ChordRun functions no longer execute successfully in NS-3.13 when ported over to the newer version. These functions include DetachNode, ReAttachNode, CrashChord, and RestartChord. DetachNode and ReAttachNode basically allows the user to disconnect various nodes at the CSMA channel level and reconnect them to the channel if possible. Perhaps the simulator allowed for this behavior at an older version, but NS-3.13 just crash with segmentation fault if these functions are executed. As for CrashChord and RestartChord, which basically tells the simulator to set each of the Chord applications running on each node to stop running immediately and rerun with time 0, the current version of NS-3.13 ignores these stop commands to individual applications and continues to run until Simulator::Stop is called.

Finally, not in terms of ChordRun unexpectedly terminating, but in terms of limited ability to display the contents stored on each of the nodes in the DHT. Since simulation means everything done in terms of message passing, setting callbacks, and printing to the screen at various points during the run, objects are rarely returned from the callee to the caller percolating up to the top ChordRun object. Therefore, to print any extra information about either Chord or the DHT underneath, we have to make changes to the **chord-ipv4/** source library.

3.2 Overcoming these Limitations in chord-main.cc

In **chord-main.cc**, we essentially rewrote **chord-run.cc** to gain robustness and allow for time-shared storage and cooperative mirroring. We kept the same topology as before, with a Chord application running on a CSMA channel on top of a typical node with a network device. We basically took out anything related to POSIX Threads, meaning only one ChordMain object will be running during the simulation. Although this grants us with fault tolerance, this also means that the NS-3 simulator will only schedule Chord input commands that are hard coded in a predetermined function. However, this is well worth the trade as NS-3 now only crashes when running out of memory when generating a network with large number of nodes.

Although the **chord-run.cc** script might be an afterthought, core functionality in the **chord-ipv4/** library is very well designed and implemented. All the components are very modular and extra functionality is easy to add to any of these parts. In terms of design, the authors decided that the function RemoveVNode in **chord-ipv4.cc** or removing the virtual node form the Chord ring should first try to send all of its contents to its successor before leaving gracefully. The design here is clearly for robustness, trying to keep content alive as long as possible. However, this is not realistic in the real world as nodes rarely leave the network gracefully. This design also impedes our project, as we are trying to remove nodes and wipe its content from the network to show the need for cooperative mirroring. To somehow simulate nodes dropping from the network without using RemoveVNode, DetachNode, or CrashChord for reasons stated previously, we must somehow tell Chord that a node is inactive and all of its contents are lost.

To overcome this issue[2], we created a function DisableNode in ChordMain that shuts down all of the interfaces of the node so that no messages can be sent or received and clears its hash-table of any contents stored. When the neighboring nodes in the virtual ring realize that no periodic messages are coming from this disabled node, the predecessor will change its successor to the closest valid node, keeping the virtual ring intact. To actually clear the hash-table, we've added the function ClearDHash in both **dhash-ipv4.cc** and **chord-ipv4.cc**, which essentially just clears the map STL container of any entries.

In terms of displaying content stored in the hash-table of each node, we've added to the function DumpDHashInfo in **dhash-ipv4.cc** to iterate through each of its items in the hash-table and print its value to the screen in string format.

3.3 Time-shared Storage

Time-shared storage essentially means that many copies of the same content should be distributed through the network to ensure content availability, even when several nodes are offline and disconnected from the network[1][2]. In ChordMain for **chord-main.cc**, we can specify how many backup copies will be distributed into the network given a single content insertion command. Chord states that the content will be hashed based on the content name, and the content will be stored at the node with the closest hash of the node name. To hash content to a different node, we need to either modify the content name in a predefined way or simply take the hash of the content name and use that as input to a new hash[2]. We chose the latter and simpler approach as the number of backup copies is simply the number of times we rehash the previous message digest. This way, it is easy to figure out exactly which hashes correspond to a certain content name, because the original content name hash is the seed message digest for the backup hashes. Content insertion and retrieval are simply computed the same way.

Time-shared storage is implemented in all content insertion, node lookup, and content retrieval functions in ChordMain, including: LookupVNode, LookupVNodeChunks, InsertContent, InsertContentChunks, RetrieveContent, RetrieveContentChunks, and RetrieveContentCheck. Of course, it makes sense that total time to insert, lookup, and retrieve grows linearly with the number of backups, as the number of insertions, lookups, and retrievals increase by one for each backup.

3.4 Cooperative Mirroring

Cooperative mirroring essentially means that nodes in the network should store an even amount of content or on average is balanced in term of load[1][2]. Since it does not make sense to move content around in Chord when the implementation dictates that certain content is hashed to a certain node, load balancing in this case means content should be broken up in to smaller chunks. Smaller sized chunks means more chunks, and more chunks means higher chance of more even distribution of content in the network. We can specify the size of each chunk in the number of bytes in ChordMain for **chord-main.cc**. The key strategy is to systematically label chunks of the same content name with a count offset appended in the content name. This way, each content chunk is hashed to possibly a different node, and retrieval uses the same count offset in the content name to get the chunks back.

Cooperative mirroring is implemented in content insertion, node lookup, and content retrieval functions in ChordMain with Chunks in the function name, including: LookupVNodeChunks, InsertContentChunks, RetrieveContentChunks, and RetrieveContentCheck. Of course, it also makes sense that total time to insert, lookup, and retrieve grows linearly with the number of chunks, as the number of insertions, lookups, and retrievals increase by one for each chunk. It seems just by eyeballing that the message passing overhead in Chord outweighs the penalty in size of content passed around.

A brief explanation of the RetrieveContentCheck should be in order, as it is the most useful function in ChordMain for checking whether time-shared storage and cooperative mirroring is implemented correctly. Its functionality is after a RetrieveContentChunks call, it looks at the retrieval callbacks and check if all the chunks and the chunk backups (if any) have been successfully retrieved for a certain content name. If all the chunks or chunk backups successfully reduce to the entire content value, RetrieveContentCheck prints out the entire content value. If some chunks and their respective chunk backups are missing, RetrieveContentCheck prints out the number of chunks of the content value that is missing. To count the number of chunks and its backups, the callback function RetrieveSuccess simply adds the chunk hash or message digest into a boolean key-value map. This way, RetrieveContentCheck can just compute the message digests of the chunks and its backups in the systematic way described above and check if the map contains such chunk hashes. Of course, the map is cleared for every retrieval. An example in the following walkthrough section will help demonstrate this function’s usefulness.

3.5 Running chord-main.cc

1. Assuming running **chord-run.cc** is successful, simply copy the **scratch/** folder included with our project into **ns-3-dev/** and merge the contents. Our **scratch/** folder includes **chord-main.cc**.
2. Also copy the **chord-ipv4/** source folder included with our project into **ns-3-dev/src/** or the location of the original **chord-ipv4/** source folder and replace the original folder entirely as we have slightly modified its contents.
3. Rebuild NS-3.13, using waf tool inside **ns-3-dev/**
   a. ./waf configure
   b. ./waf build
5. To run **chord-main.cc**, inside **ns-3-dev/**:
   a. ./waf --run chord-main

4. Walkthrough of an Example Scenario

4.1 Usage Parameters

ChordMain object when created has needs the user to set some constraints for the Chord simulation. These constraints include number of nodes in the network, ID value of the bootstrap nodes, total simulation time in seconds, number of bytes for a chunk of content, and number of backup content values.

For example: ChordMain chordMain(10, 0, 240.0, 10, 1);
In this case, we are simulating 10 nodes with the 0th node being the bootstrap node and constrain the simulation to stop after running for 240 seconds. Also in this case, each size chunk of content is 10 bytes long, and 1 backup copy for each chunk will be made to distribute into the Chord ring.

4.2 Input Commands

All the Chord input commands are now hardcoded in the ChordMain function SetCommands, where individual commands are simulated using Simulator::schedule. Simulator::schedule basically takes an exact time in which to execute a certain function, the function pointer, the class in which the function resides, and the function's parameters.

Let us go through a possible scenario below using various common Chord commands to see how a typical Chord ring would be used over some time.

```
// Schedule tasks to insert 10 physical nodes into virtual ring,
// 30 seconds into the simulation with node names VNode0 to VNode9.
// A pointer to Chord application running on each physical node is necessary.
```

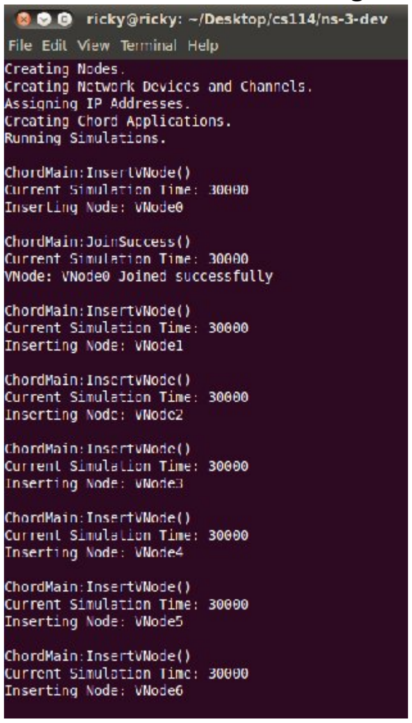
```
Simulator::Schedule (Seconds (30), &ChordMain::InsertVNode, this,
                              chordAppPtrList[0], "VNode0");
                              …
                              …
Simulator::Schedule (Seconds (30), &ChordMain::InsertVNode, this,
                              chordAppPtrList[9], "VNode9");

// Insert content name (e.g. "Sherlock Holmes") and content value
// (e.g. "Elementary, my dear Watson.") into the virtual ring at 50 seconds.
// Content value is broken up into 10byte chunks and inserted with 1 backup.
// Chord application pointer must be pointing to a physical node that is
// successfully joined or insertion will fail (e.g. node 2).
Simulator::Schedule (Seconds (50), &ChordMain::InsertContentChunks, this,
                              chordAppPtrList[2],
                              "Sherlock Holmes",
                              "Elementary, my dear Watson.");

Simulator::Schedule (Seconds (55), &ChordMain::InsertContentChunks, this,
                              chordAppPtrList[2],
                              "Apollo 13",
                              "Houston, we have a problem.");

// Lookup the physical node address and port number for the given content name
// (e.g. "Apollo 13") and a starting physical node to look from
// (e.g. node 3), internally aggregating chunks and any backups
Simulator::Schedule (Seconds (60), &ChordMain::LookupVNodeChunks, this,
                              chordAppPtrList[3], "Sherlock Holmes");
Simulator::Schedule (Seconds (70), &ChordMain::LookupVNodeChunks, this,
                              chordAppPtrList[3], "Apollo 13");

// Retrieve the content value given the content name (e.g. "Sherlock Holmes")
// starting from a physical node (e.g. node 4), internally aggregating chunks
// and any backups.  Check if all the chunks and any backups are returned and
// return the final content value if possible.
Simulator::Schedule (Seconds (80), &ChordMain::RetrieveContentChunks, this,
                              chordAppPtrList[4], "Sherlock Holmes");
Simulator::Schedule (Seconds (90), &ChordMain::RetrieveContentCheck, this,
                              "Sherlock Holmes");
Simulator::Schedule (Seconds (100), &ChordMain::RetrieveContentChunks, this,
                              chordAppPtrList[4], "Apollo 13");
```

```
Simulator::Schedule (Seconds (110), &ChordMain::RetrieveContentCheck, this,
                                    "Apollo 13");

// Shut down the interfaces of the node (e.g. node 2) and clear its hashtable
// to imitate the removal of that node at 120 seconds into the simulation.
// Chord periodically thereafter checks the interfaces of the neighbors, so
// the Chord neighbors thinks the removed node is dead and Chord fixes the
// successors and finger tables of the nodes in the ring
Simulator::Schedule (Seconds (120), &ChordMain::DisableNode, this,
                                    chordAppPtrList[2], nodes, 2);

// Since we disabled the node at 120 seconds, we want to see if this affects
// retrieval of the same content (e.g. "Sherlock Holmes" and "Apollo 13")
// starting from a physical node (e.g. node 5), internally aggregating chunks
// and any backups.  Check if all the chunks and any backups are returned and
// return the final content value if possible.
Simulator::Schedule (Seconds (150), &ChordMain::RetrieveContentChunks, this,
                                    chordAppPtrList[5], "Sherlock Holmes");
Simulator::Schedule (Seconds (160), &ChordMain::RetrieveContentCheck, this,
                                    "Sherlock Holmes");
Simulator::Schedule (Seconds (170), &ChordMain::RetrieveContentChunks, this,
                                    chordAppPtrList[5], "Apollo 13");
Simulator::Schedule (Seconds (180), &ChordMain::RetrieveContentCheck, this,
                                    "Apollo 13");

// Dump physical node information and its content in this case form VNode0 to
// VNode9.  This time, we need Chord application pointer, node’s CSMA
// interfaces container, and physical node number (e.g. 0 - 9).
Simulator::Schedule (Seconds (190), &ChordMain::DumpDHashInfo, this,
                                    chordAppPtrList[0], csmaInterfaces, 0, "VNode0");
                                    …
                                    …
Simulator::Schedule (Seconds (190), &ChordMain::DumpDHashInfo, this,
                                    chordAppPtrList[9], csmaInterfaces, 9, "VNode9");

// Print the virtual ring in a list format starting with the given virtual
// node name (e.g. VNode0) and a physical node (e.g. node 0)
Simulator::Schedule (Seconds (200), &ChordMain::TraceRing, this,
                                    chordAppPtrList[0], "VNode0");
```

4.3 Output Explained

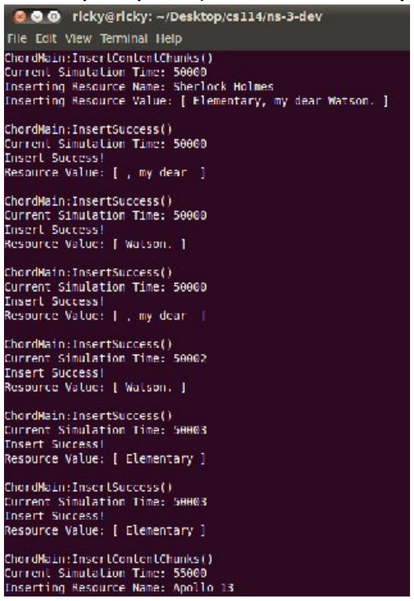
Screenshots of the output of the previous scenario is described in this section. For an output of a scenario with more inserted content, refer to the **out.txt** in our project **scratch/** folder.

1. Nodes, net devices, channels, interfaces, Chord apps are set up. Virtual nodes are inserted into the Chord ring. Bootstrap node VNode0 seems to join instantaneously.


ricky@ricky: ~/Desktop/cs114/ns-3-dev
File Edit View Terminal Help

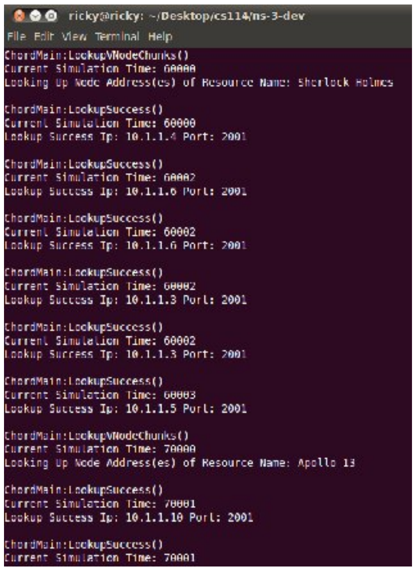
```
Creating Nodes.
Creating Network Devices and Channels.
Assigning IP Addresses.
Creating Chord Applications.
Running Simulations.

ChordMain:InsertVNode()
Current Simulation Time: 30000
Inserting Node: VNode0

ChordMain:JoinSuccess()
Current Simulation Time: 30000
VNode: VNode0 Joined successfully

ChordMain:InsertVNode()
Current Simulation Time: 30000
Inserting Node: VNode1

ChordMain:InsertVNode()
Current Simulation Time: 30000
Inserting Node: VNode2

ChordMain:InsertVNode()
Current Simulation Time: 30000
Inserting Node: VNode3

ChordMain:InsertVNode()
Current Simulation Time: 30000
Inserting Node: VNode4

ChordMain:InsertVNode()
Current Simulation Time: 30000
Inserting Node: VNode5

ChordMain:InsertVNode()
Current Simulation Time: 30000
Inserting Node: VNode6
```

2. Inserting content for Sherlock Holmes is internally breaking up into chunks and add backup copies (in this case only one backup).


ricky@ricky: ~/Desktop/cs114/ns-3-dev
File Edit View Terminal Help

```
ChordMain:InsertContentChunks()
Current Simulation Time: 50000
Inserting Resource Name: Sherlock Holmes
Inserting Resource Value: [ Elementary, my dear Watson. ]

ChordMain:InsertSuccess()
Current Simulation Time: 50000
Insert Success!
Resource Value: [ , my dear  ]

ChordMain:InsertSuccess()
Current Simulation Time: 50000
Insert Success!
Resource Value: [ Watson. ]

ChordMain:InsertSuccess()
Current Simulation Time: 50000
Insert Success!
Resource Value: [ , my dear  ]

ChordMain:InsertSuccess()
Current Simulation Time: 50002
Insert Success!
Resource Value: [ Watson. ]

ChordMain:InsertSuccess()
Current Simulation Time: 50003
Insert Success!
Resource Value: [ Elementary ]

ChordMain:InsertSuccess()
Current Simulation Time: 50003
Insert Success!
Resource Value: [ Elementary ]

ChordMain:InsertContentChunks()
Current Simulation Time: 55000
Inserting Resource Name: Apollo 13
```

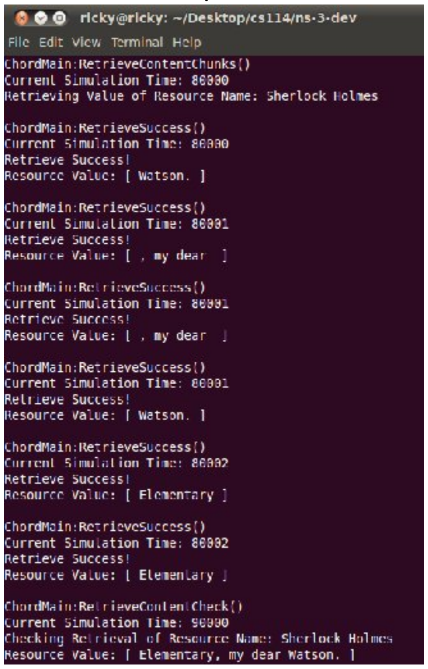
3. Look up the node’s address and port number that stores the chunks and any backups for Sherlock Holmes



ricky@ricky: ~/Desktop/cs114/ns-3-dev
File Edit View Terminal Help

```
ChordMain:LookupVNodeChunks()
Current Simulation Time: 60000
Looking Up Node Address(es) of Resource Name: Sherlock Holmes

ChordMain:LookupSuccess()
Current Simulation Time: 60000
Lookup Success Ip: 10.1.1.4 Port: 2001

ChordMain:LookupSuccess()
Current Simulation Time: 60002
Lookup Success Ip: 10.1.1.6 Port: 2001

ChordMain:LookupSuccess()
Current Simulation Time: 60002
Lookup Success Ip: 10.1.1.6 Port: 2001

ChordMain:LookupSuccess()
Current Simulation Time: 60002
Lookup Success Ip: 10.1.1.3 Port: 2001

ChordMain:LookupSuccess()
Current Simulation Time: 60002
Lookup Success Ip: 10.1.1.3 Port: 2001

ChordMain:LookupSuccess()
Current Simulation Time: 60003
Lookup Success Ip: 10.1.1.5 Port: 2001

ChordMain:LookupVNodeChunks()
Current Simulation Time: 70000
Looking Up Node Address(es) of Resource Name: Apollo 13

ChordMain:LookupSuccess()
Current Simulation Time: 70001
Lookup Success Ip: 10.1.1.10 Port: 2001

ChordMain:LookupSuccess()
Current Simulation Time: 70001
```

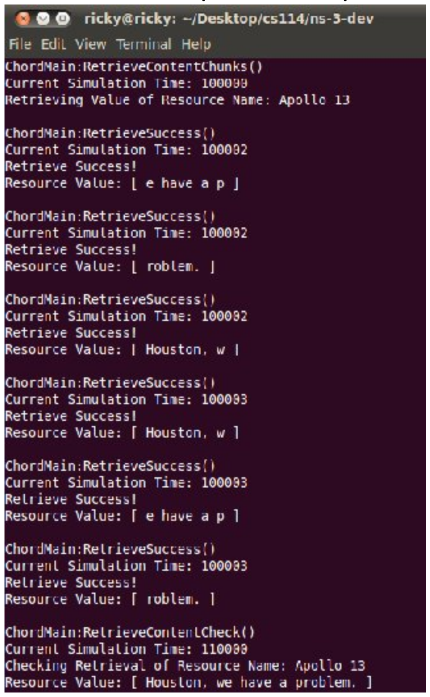
4. Retrieval of content for Sherlock Holmes is internally aggregating chunks and any backup copies (in this case only one backup copy per chunk). As you can see, the chunks are out of order and separated. RetrieveContentCheck checks for all the necessary chunks and outputs the entire content value.



ricky@ricky: ~/Desktop/cs114/ns-3-dev
File Edit View Terminal Help

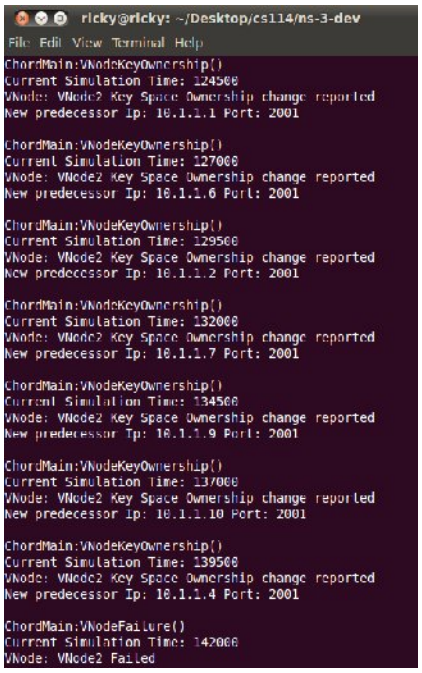
```
ChordMain:RetrieveContentChunks()
Current Simulation Time: 80000
Retrieving Value of Resource Name: Sherlock Holmes

ChordMain:RetrieveSuccess()
Current Simulation Time: 80000
Retrieve Success!
Resource Value: [ Watson. ]

ChordMain:RetrieveSuccess()
Current Simulation Time: 80001
Retrieve Success!
Resource Value: [ , my dear  ]

ChordMain:RetrieveSuccess()
Current Simulation Time: 80001
Retrieve Success!
Resource Value: [ , my dear  ]

ChordMain:RetrieveSuccess()
Current Simulation Time: 80001
Retrieve Success!
Resource Value: [ Watson. ]

ChordMain:RetrieveSuccess()
Current Simulation Time: 80002
Retrieve Success!
Resource Value: [ Elementary ]

ChordMain:RetrieveSuccess()
Current Simulation Time: 80002
Retrieve Success!
Resource Value: [ Elementary ]

ChordMain:RetrieveContentCheck()
Current Simulation Time: 90000
Checking Retrieval of Resource Name: Sherlock Holmes
Resource Value: [ Elementary, my dear Watson. ]
```

5. Same retrieval process as previous step now for Apollo 13.



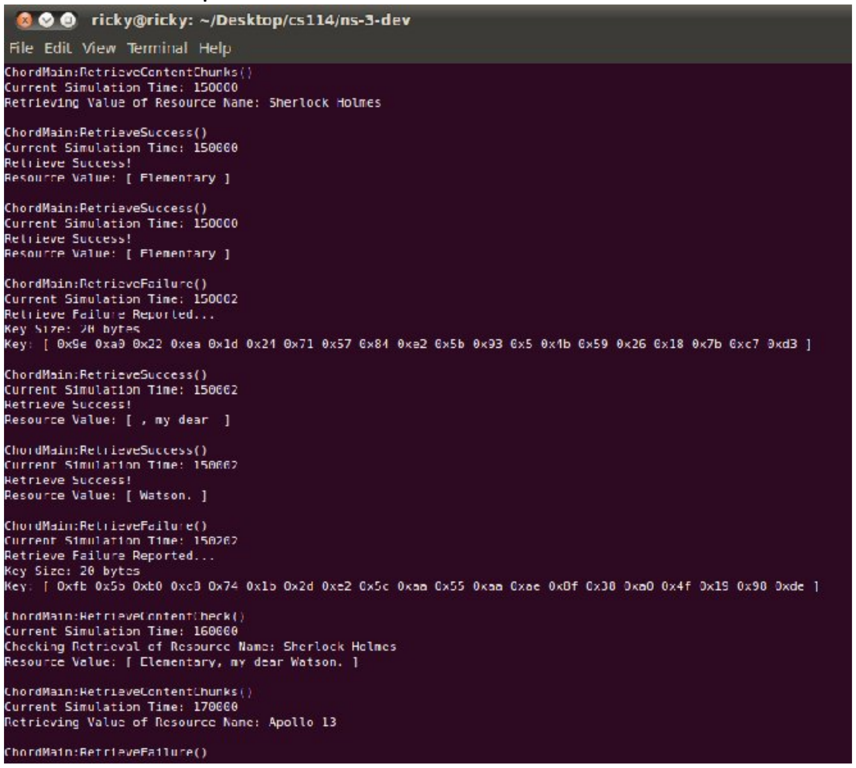
```
ChordMain:RetrieveContentChunks()
Current Simulation Time: 100000
Retrieving Value of Resource Name: Apollo 13

ChordMain:RetrieveSuccess()
Current Simulation Time: 100002
Retrieve Success!
Resource Value: [ e have a p ]

ChordMain:RetrieveSuccess()
Current Simulation Time: 100002
Retrieve Success!
Resource Value: [ roblem. ]

ChordMain:RetrieveSuccess()
Current Simulation Time: 100002
Retrieve Success!
Resource Value: [ Houston, w ]

ChordMain:RetrieveSuccess()
Current Simulation Time: 100003
Retrieve Success!
Resource Value: [ Houston, w ]

ChordMain:RetrieveSuccess()
Current Simulation Time: 100003
Retrieve Success!
Resource Value: [ e have a p ]

ChordMain:RetrieveSuccess()
Current Simulation Time: 100003
Retrieve Success!
Resource Value: [ roblem. ]

ChordMain:RetrieveContentCheck()
Current Simulation Time: 110000
Checking Retrieval of Resource Name: Apollo 13
Resource Value: [ Houston, we have a problem. ]
```

6. Node 2 or VNode2 was disabled at the specified 120 second mark and its content wiped. Chord recognizes the death of this node at the 122 second mark and routes around it looking for a new successor. However, as you may have noticed, the VNodeFailure callback executed at the 142 second mark. It seems just from this run that fixing the loss of one node in the Chord ring takes on the order of 10's of seconds, in this case 20 seconds.



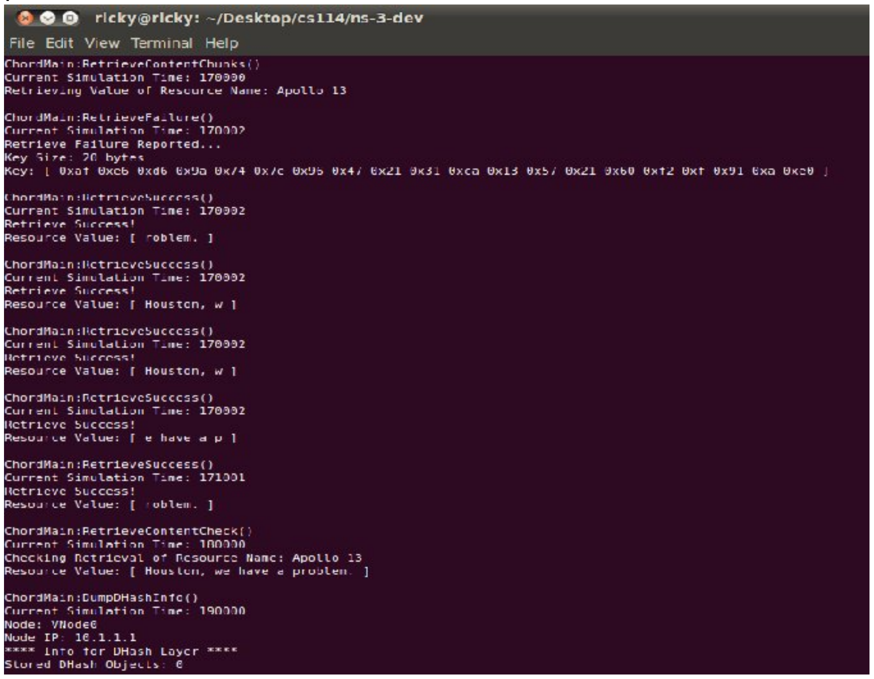
```
ChordMain:VNodeKeyOwnership()
Current Simulation Time: 124500
VNode: VNode2 Key Space Ownership change reported
New predecessor Ip: 10.1.1.1 Port: 2001

ChordMain:VNodeKeyOwnership()
Current Simulation Time: 127000
VNode: VNode2 Key Space Ownership change reported
New predecessor Ip: 10.1.1.6 Port: 2001

ChordMain:VNodeKeyOwnership()
Current Simulation Time: 129500
VNode: VNode2 Key Space Ownership change reported
New predecessor Ip: 10.1.1.2 Port: 2001

ChordMain:VNodeKeyOwnership()
Current Simulation Time: 132000
VNode: VNode2 Key Space Ownership change reported
New predecessor Ip: 10.1.1.7 Port: 2001

ChordMain:VNodeKeyOwnership()
Current Simulation Time: 134500
VNode: VNode2 Key Space Ownership change reported
New predecessor Ip: 10.1.1.9 Port: 2001

ChordMain:VNodeKeyOwnership()
Current Simulation Time: 137000
VNode: VNode2 Key Space Ownership change reported
New predecessor Ip: 10.1.1.10 Port: 2001

ChordMain:VNodeKeyOwnership()
Current Simulation Time: 139500
VNode: VNode2 Key Space Ownership change reported
New predecessor Ip: 10.1.1.4 Port: 2001

ChordMain:VNodeFailure()
Current Simulation Time: 142000
VNode: VNode2 Failed
```

7. Since node 2 died with its content, some of the chunks or backup chunks fail to be retrieved for Sherlock Holmes. If you notice, those same chunks were successfully retrieved in step 4.



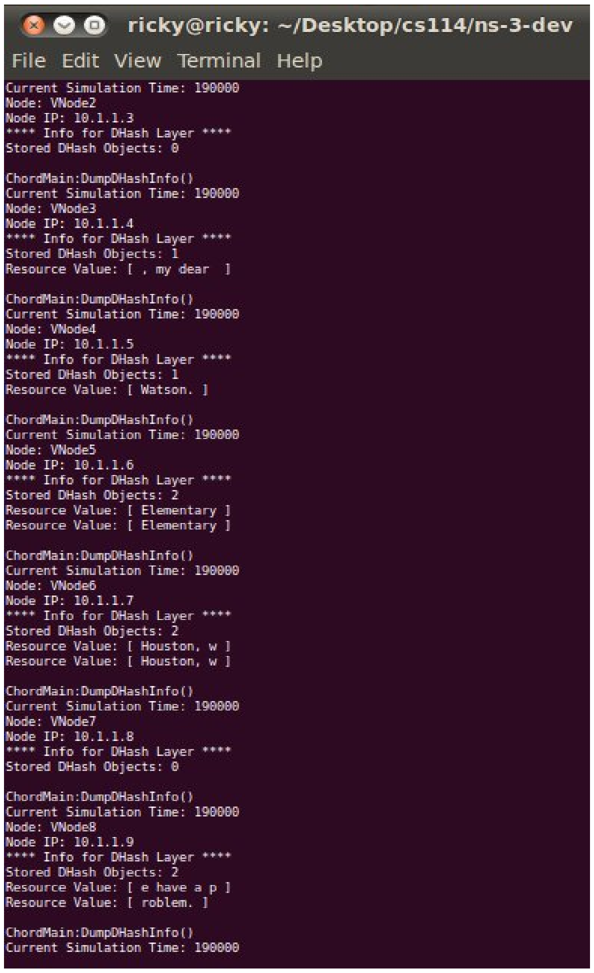
```
ChordMain:RetrieveContentChunks()
Current Simulation Time: 150000
Retrieving Value of Resource Name: Sherlock Holmes

ChordMain:RetrieveSuccess()
Current Simulation Time: 150000
Retrieve Success!
Resource Value: [ Elementary ]

ChordMain:RetrieveSuccess()
Current Simulation Time: 150000
Retrieve Success!
Resource Value: [ Elementary ]

ChordMain:RetrieveFailure()
Current Simulation Time: 150002
Retrieve Failure Reported...
Key Size: 20 bytes
Key: [ 0x9e 0xa0 0x22 0xea 0x1d 0x21 0x71 0x57 0x84 0xe2 0x5b 0x93 0x5 0x1b 0x59 0x26 0x18 0x7b 0xc7 0xd3 ]

ChordMain:RetrieveSuccess()
Current Simulation Time: 150002
Retrieve Success!
Resource Value: [ , my dear  ]

ChordMain:RetrieveSuccess()
Current Simulation Time: 150002
Retrieve Success!
Resource Value: [ Watson. ]

ChordMain:RetrieveFailure()
Current Simulation Time: 150202
Retrieve Failure Reported...
Key Size: 20 bytes
Key: [ 0xfb 0x5b 0xb0 0xc0 0x74 0x1b 0x2d 0xe2 0x5c 0xaa 0x55 0xaa 0xae 0x0f 0x30 0xa0 0x4f 0x19 0x90 0xde ]

ChordMain:RetrieveContentCheck()
Current Simulation Time: 160000
Checking Retrieval of Resource Name: Sherlock Holmes
Resource Value: [ Elementary, my dear Watson. ]

ChordMain:RetrieveContentChunks()
Current Simulation Time: 170000
Retrieving Value of Resource Name: Apollo 13

ChordMain:RetrieveFailure()
```

8. Same retrieval process as previous step, now for Apollo 13. Thing to point out is that for both retrievals, since we have backup copies of chunks, we are able to successfully return the entire content value: Elementary, my dear Watson. and Houston, we have a problem.



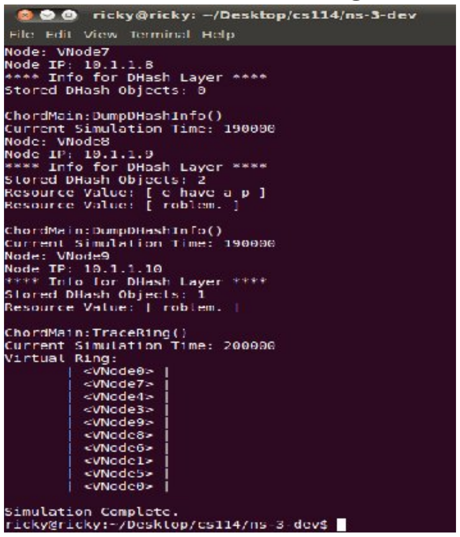
```
ChordMain:RetrieveContentChunks()
Current Simulation Time: 170000
Retrieving Value of Resource Name: Apollo 13

ChordMain:RetrieveFailure()
Current Simulation Time: 170002
Retrieve Failure Reported...
Key Size: 20 bytes
Key: [ 0xaf 0xc6 0xd6 0x9a 0x74 0x7c 0x95 0x47 0x21 0x31 0xca 0x13 0x57 0x21 0x60 0xf2 0xf 0x91 0xa 0xc0 ]

ChordMain:RetrieveSuccess()
Current Simulation Time: 170002
Retrieve Success!
Resource Value: [ roblem. ]

ChordMain:RetrieveSuccess()
Current Simulation Time: 170002
Retrieve Success!
Resource Value: [ Houston, w ]

ChordMain:RetrieveSuccess()
Current Simulation Time: 170002
Retrieve Success!
Resource Value: [ Houston, w ]

ChordMain:RetrieveSuccess()
Current Simulation Time: 170002
Retrieve Success!
Resource Value: [ e have a p ]

ChordMain:RetrieveSuccess()
Current Simulation Time: 171001
Retrieve Success!
Resource Value: [ roblem. ]

ChordMain:RetrieveContentCheck()
Current Simulation Time: 180000
Checking Retrieval of Resource Name: Apollo 13
Resource Value: [ Houston, we have a problem. ]

ChordMain:DumpDHashInfo()
Current Simulation Time: 190000
Node: VNode0
Node IP: 10.1.1.1
**** Info for DHash Layer ****
Stored DHash Objects: 0
```

9. After node 2 was disabled, we can see here that VNode2 has 0 pieces of content stored in its hashtable. Other content chunks or backups are shown in the hash dump for the other nodes.



```
Current Simulation Time: 190000
Node: VNode2
Node IP: 10.1.1.3
**** Info for DHash Layer ****
Stored DHash Objects: 0

ChordMain:DumpDHashInfo()
Current Simulation Time: 190000
Node: VNode3
Node IP: 10.1.1.4
**** Info for DHash Layer ****
Stored DHash Objects: 1
Resource Value: [ , my dear  ]

ChordMain:DumpDHashInfo()
Current Simulation Time: 190000
Node: VNode4
Node IP: 10.1.1.5
**** Info for DHash Layer ****
Stored DHash Objects: 1
Resource Value: [ Watson. ]

ChordMain:DumpDHashInfo()
Current Simulation Time: 190000
Node: VNode5
Node IP: 10.1.1.6
**** Info for DHash Layer ****
Stored DHash Objects: 2
Resource Value: [ Elementary ]
Resource Value: [ Elementary ]

ChordMain:DumpDHashInfo()
Current Simulation Time: 190000
Node: VNode6
Node IP: 10.1.1.7
**** Info for DHash Layer ****
Stored DHash Objects: 2
Resource Value: [ Houston, w ]
Resource Value: [ Houston, w ]

ChordMain:DumpDHashInfo()
Current Simulation Time: 190000
Node: VNode7
Node IP: 10.1.1.8
**** Info for DHash Layer ****
Stored DHash Objects: 0

ChordMain:DumpDHashInfo()
Current Simulation Time: 190000
Node: VNode8
Node IP: 10.1.1.9
**** Info for DHash Layer ****
Stored DHash Objects: 2
Resource Value: [ e have a p ]
Resource Value: [ roblem. ]

ChordMain:DumpDHashInfo()
Current Simulation Time: 190000
```

10. Just to show the virtual ring in list format, which gives an idea who are the neighbors.



```
Node: VNode7
Node IP: 10.1.1.8
**** Info for DHash Layer ****
Stored DHash Objects: 0

ChordMain:DumpDHashInfo()
Current Simulation Time: 190000
Node: VNode8
Node IP: 10.1.1.9
**** Info for DHash Layer ****
Stored DHash Objects: 2
Resource Value: [ e have a p ]
Resource Value: [ roblem. ]

ChordMain:DumpDHashInfo()
Current Simulation Time: 190000
Node: VNode9
Node IP: 10.1.1.10
**** Info for DHash Layer ****
Stored DHash Objects: 1
Resource Value: [ roblem. ]

ChordMain:TraceRing()
Current Simulation Time: 200000
Virtual Ring:
        | <VNode0> |
        | <VNode7> |
        | <VNode4> |
        | <VNode3> |
        | <VNode9> |
        | <VNode8> |
        | <VNode6> |
        | <VNode1> |
        | <VNode5> |
        | <VNode0> |

Simulation Complete.
ricky@ricky:~/Desktop/cs114/ns-3-dev$
```

5. Performance Evaluation

We gave different inputs and measured the performance of the chord protocol. Experiments are carried out on different versions of chord protocol.

<u>10 nodes:</u>
We tried to insert 10 nodes and look up for the contents in the nodes. Data is split into chunks and inserted in the nodes. Insertion of nodes into chord is done very fast. The ten nodes are inserted in 2 milliseconds. but this is just the raw insertion of nodes into the chord protocol. The finger table pointers consumes time in getting updated. The updation of finger table takes 1000 milliseconds. So in one seconds, insertion of nodes with proper finger table entries is complete.

The contents are split into chunks and inserted into the nodes. This overall process takes 400 milliseconds. Lookup for a content are faster. It takes 2 milliseconds to check for a content in the available nodes. Retrieval of content varies based on the location of the node in the chord protocol. It varies from 4 milliseconds to 200 milliseconds.

Incase of any failure, fixing the finger table with proper entries take 200 milliseconds. Insertion of content took 1 second.

<u>30 nodes:</u>
We repeated the same set of experiments for 30 nodes. Node insertion for 30 nodes took 7 seconds. Insertion the content into the nodes took 1 second. So the performance of Chord protocol depends on the number of nodes. Another interesting finding is the performance of chord protocol does not depend much on the type of content insertion. If the content is inserted without splitting into chunks the time consumed is 405 milliseconds (In contrast to 409 milliseconds for insertion of chunks of data).

Time Taken (milliseconds)		
Operation	10 nodes	30 nodes
Node Insertion	1000	7000
Content Insertion	500	1000
Insertion by Chunks	405	409

6. Conclusion

Chord implemented by the folks at MIT is very robust and their code is extremely modular. The Chord library **chord-ipv4/** is relatively standalone and thus fairly easy to port to a newer version of NS-3. Adding functionality to the various components in the core library is fairly painless with very little code refactoring. However, their example testing script **chord-run.cc** requires user-made module-aggregator header files and additional link flags for building. In addition, several methods for testing Chord does not work in the newer version of NS-3. In terms of the actual implementation, it seems Chord is viable or scalable in inserting and retrieving content. In fact, adding cooperative mirroring and time-shared storage reinforces its theme in availability of content in the Chord ring. However, the drawback for Chord or all structured DHT peer-to-peer systems is that inserting nodes or maintaining the virtual Chord ring when nodes drop from the network is a fairly slow process. In this respect, Chord is not viable or scalable for large networks. Hopefully this paper sheds some light into Chord's characteristics.

7. References

[1] *Professor Giovanni Pau.*
[2] *TA Alexander Afanasyev.*
[3] *Chord (peer-to-peer)*, <http://en.wikipedia.org/wiki/Chord_%28peer-to-peer%29>.
[4] *Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan.* Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications.
[5] *MIT Chord*, <http://pdos.csail.mit.edu/chord/>.
[6] *NS-3 Documentation*, <http://www.nsnam.org/doxygen-release/index.html>.
[7] *NS-3 Tutorial*,
<http://www.nsnam.org/docs/release/3.12/tutorial/singlehtml/index.html>.
[8] *NS-3.7 Chord Source*, <http://code.nsnam.org/gillh/ns-3-chord/>.
[9] *NS-3.13 Source*, <http://code.nsnam.org/ns-3-dev>.