

CS138 Programming Assignment 1: Chord

<i>Assignment Out:</i>	Jan. 22, 2009
<i>Helpsession:</i>	Jan. 27, 2009 (Location TBD, 7pm)
<i>Assignment Due:</i>	Feb. 9, 2009 (11:59 pm)

1 Introduction

Welcome to CS138!

This project is a self-contained introduction to many of the concepts that you'll be using frequently in CS138. You'll be implementing a real distributed system whilst getting practice using RMI and concurrency control in Java. Before you dive into this project, make sure you've signed and turned in the collaboration policy and that you've read the coding guidelines that we've posted on the website. The second document will explain some of the conventions we expect you to follow when writing your code, and it will also provide some of the necessary background knowledge of RMI and multi-threaded programming.

2 Chord

Chord is a distributed hash table (DHT) protocol currently under development at MIT. It was proposed in 2001 in a paper titled "Scalable Peer-to-peer Lookup Service for Internet Applications"¹. From an application's perspective, Chord simply provides a service that can store key-value pairs and find the value associated with a key reasonably quickly. Behind this simple interface, Chord distributes objects over a dynamic network of nodes, and implements a protocol for finding these objects once they have been placed in the network. Every node in this network is a server capable of looking up keys for client applications, but also participates as key store. Hence, Chord is a decentralized system in which no particular node is necessarily a performance bottleneck or a single point of failure (if the system is implemented to be fault-tolerant).

2.1 Keys

Every key inserted into the DHT must be hashed to fit into the key-space supported by the particular implementation of Chord. The hashed value of the key will take the form of an m bit unsigned integer. Thus, the keyspace (the range of possible hashes) for the DHT resides between

¹http://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf

0 and $2^m - 1$, inclusive. Standard practice for most DHT implementations is to use a 128 or 160 bit hash, where the hash is produced by a message digest algorithm such as MD5 or SHA-1. Using these hashing algorithms ensures with high probability that the hashes generated from keys are distributed evenly throughout the keyspace. Note that this does not restrict the number of distinct keys that may be stored by the DHT, as the hash only provides a guide for locating the key in the network, rather than providing the identifier for the key. It is possible, though unlikely, for the hash values of distinct keys to collide.

2.2 The Ring

Just as each key that is inserted into the DHT has hash value, each node in the system also has a hash value in the keyspace of the DHT. To get this hash value, we could simply give each node a distinct name and then take the hash of the name, using the same hashing algorithm we use to hash keys inserted into the DHT. Once each node has a hash value, we are able to give the nodes an ordering based on those hashes. Chord orders the node in a circular fashion, in which each node's successor is the node with the next highest hash. The node with the largest hash, however, has the node with the smallest hash as its successor. It is easy to imagine the nodes placed in a ring, where each node's successor is the node after it when following a clockwise rotation.

To locate the node at which a particular key-value pair is stored, one need only find the successor to the hash value of the key.

2.3 The Overlay Network

As the paper in which Chord was introduced states, it would possible to look up any particular key by sending an iterative request around the ring. Each node would determine whether its successor is the owner of the key, and if so perform the request at its successor. Otherwise, the node asks its successor to find the successor of the key and the same process is repeated. Unfortunately, this method of finding successors would be incredibly slow on a large network of nodes. For this reason, Chord employs a clever overlay network that, when the topology of the network is stable, routes requests to the successor of a particular key in $\log n$ time, where n is the number of nodes in the network.

This optimized search for successors is made possible by maintaining a “finger” table at each node. The number of entries in the finger table is equal to m , where m is the number of bits representing a hash in the keyspace of the DHT. Entry i in the table, with $0 \leq i < m$, is the node which the owner of the table believes is the successor for the hash $h + 2^i$ (h is the current node's hash). When node A services a request to find the successor of the key k , it first determines whether its own successor is the owner of k (the successor is simply entry 0 in the finger table). If it is, then A returns its successor in response to the request. Otherwise, node A finds node B in its finger table such that B has the largest hash smaller than the hash of k , and forwards the request to B .

2.4 Dynamics

Chord would be far less useful if it were not designed to support the dynamic addition and removal of nodes from the network, requiring a static allocation of nodes instead. A production ready implementation of Chord would support the ability to add and remove nodes from the network at arbitrary times, as well as cope with the failure of some nodes, all without interrupting the ongoing client requests being served by the DHT. This functionality complicates the implementation considerably, though.

To allow membership in the ring to change, protocols for creating a ring, adding a node to the ring, and leaving the ring must be defined. Creating the ring is easy. The first node fills its finger table with references to itself, and has no predecessor. Then, when node *A* joins the network, it asks an existing node in the ring to find the successor of the hash of *A*. The node returned from that request becomes the successor of *A*. The predecessor of *A* is undefined until some other node notifies *A* that it believes that *A* is its successor. In order to determine the successor and predecessor relationships between nodes as they are added to the network (and voluntarily removed), each Chord node performs `stabilize` and `fixNextFinger` periodically:

```
n - this node
h - hash of n
m - the number of bits in a hash

stabilize()
  x = successor.predecessor
  if x is between n and successor
    successor = x
  successor.notify(n)

fixNextFinger()
  next_hash = h + 2^(next) mod 2^m
  finger[next] = findSuccessor(next_hash)
  next = next + 1
  if next > m - 1
    next = 1

notify(p)
  if predecessor is null OR p is between predecessor and n
    predecessor = p
    transfer appropriate keys to predecessor
```

The following method is called when a node is asked to leave the network:

```
leave()
```

```

transfer all keys to successor
successor.predecessor = null
predecessor.successor = predecessor

```

Fault tolerance is achieved by maintaining successor lists, rather than a single successor so that the failure of a few nodes is not enough to send the system into disrepair. Keys must also be replicated across a number of nodes so that they are available in the event that some of the nodes storing them fail. The implementation details of replication are beyond the scope of this assignment.

In order to understand how the system works as a whole, it is important to see that although optimizations based on the finger table may not always be available, it is always possible to find the correct successor node for a given hash. This is an invariant of the system, even when nodes are joining and leaving the network with great frequency.

3 The Assignment

You will be implementing a simple version of Chord. Your implementation need not be fault tolerant, and we are stressing correctness over performance in this assignment. To remove unnecessary complication, you should treat key-value pairs in the DHT as **immutable**. This means that once a key-value pair is inserted into the DHT, it cannot be deleted and the value associated with the key may not change (you should enforce this requirement). Your implementation of Chord should support dynamic insertion of nodes (but not removal), and continue to serve `get` and `put` requests simultaneously and correctly (if a value exists for a key, it must always be accessible). Keys should never reside at more than two nodes at any given time, and only one node when the ring is in a stable state.

Your TAs have written a significant amount of support code for you. The code you must write is isolated to the `ChordNodeImplementation` class, which is contained in the `ChordNodeFactory.java` file. You should implement the following methods:

```

public Serializable getLocal(String key) throws RemoteException
public boolean putLocal(String key, Serializable data) throws RemoteException
public ChordNode locate(String key) throws RemoteException
public RemoteChordNode findSuccessor(BigInteger hash) throws RemoteException
private void stabilize()
public void notify(RemoteChordNode node) throws RemoteException
private void fixNextFinger()

```

The difficulty in this project does not lie in the quantity of code you will write. Understanding how Chord works and protecting code that may be run concurrently are the biggest hurdles to completing a correct implementation.

4 Testing

There are a few annoyances related to Java RMI, so we have provided an `ant` build file with `compile`, `run`, and `debug` targets (read the comments in the `build.xml` file to see how they work). There are multiple ways to compile and run your code using this file.

1. **ant** - Use the `compile` target to compile your code. Use the `run` target to create nodes. The `run` target will interactively ask you to input some information about the node you are setting up. The `debug` target is used for connecting to the `eclipse` remote debugger, but it gets complicated. If you want to debug your code using `eclipse` you should write all of your code in `eclipse` (see below).
2. **node.sh** - This script is a wrapper around the build file which allows you to enter information on the command line rather than having to enter it interactively. The format is:

```
node.sh <node id> [<host name> <remote node id>] [<debug port>]
```

This script will create a node named `<node id>` on the local host. If `<host name>` and `<remote node id>` are specified then the new node will look up the named node on the given host and join its ring. If `<debug port>` is specified then debugging will be enabled and listening on the given port. If you wish to use debugging we suggest you write all of your code in `eclipse` instead. All code will be compiled before starting the node.

3. **eclipse** - This is the most complicated to set up, but it will make debugging your code a lot simpler. First create a new project in `eclipse`. The project's name *must* be `chord`, and it must use a directory called `src` to hold all source files (this is the default if you created your project using the "New Project" wizard). Import the support code into the `src` directory. Import the `build.xml` file and all of the `.launch` files into the base directory of the project (should be the parent directory of `src`). Now you should be all set. Take a look on the toolbar to the right of the save button where you see the pictures of a bug and a play button right next to each other. If you click on the drop down arrow of the right most button in this set (the one with a play button in the background and a red object in the foreground) you should see "External Tools Configurations...". In that dialog you should find the `debug` and `run` targets. Each of these options will start one node. Dialogs will pop up to ask you for the same information you enter if you use the build file. If you look at the drop down menu for the debug menu (the picture of the bug) you should see the "Debug Configurations..." dialog, which contains the `8000` run target. This option will start a remote debugging session for the node whose debugging session connects to port "8000". You can easily change the port number and create new targets for other ports using the "Debug Configurations..." dialog.

We definitely suggest you come to the Chord help session where we will be going over the quirks in RMI that necessitate these steps as well as go over setting up debugging in `eclipse` (in addition to actually going over Chord).

You're encouraged to write client applications, using the `ChordClient` class provided, to test your implementation. We've written one that you can use, too. In the `/course/cs138/demo/chord/` directory, you'll find a script called `console.sh` that will allow you to perform `get` and `put` operations with your nodes. It also allows you to get some debugging information from the node you're connected to, such as the contents of the finger table and a list of which keys are held at the node. You can run `console.sh` with no arguments to get an interactive prompt, or specify a host, node, and command as arguments when you run the script. To see what commands are available, run the script interactively and enter `help` at the prompt.

In the same directory, you also have access to the TA implementation of the node, which is accessed with the `node.sh` script in that directory. It uses the same arguments that the one in your project directory does. Your implementation should behave as well as the TA implementation to get an A on the assignment. Please note that your implementation is not expected to perform well with multiple node insertions made rapidly. It should behave correctly with about a 1 to 2 second lapse between node insertions.

5 Code Exchange

To get started, copy the `/course/cs138/asgn/chord` directory to your CS138 course directory.

6 Handing in

You need to write a simple README, documenting any bugs you have in your code, any extra features you added, and anything else you think the TAs should know about your project. You should hand in your chord by running `"/course/cs138/bin/cs138_handin chord"` from your project directory.