Lab Exercise 3 – Implementing a Decentralized P2P Instant Messaging System

CIS656 – Fall 2011

Due Date: November 3, 2011 (see below for description of deliverables.)
Grade Weight: 25 points (out of a total of 100 Lab points)

Objective

The objective of this project is to give the student hands-on experience building a decentralized peer-to-peer application using distributed hash tables (DHT).

Lab Description

In the previous lab exercise you implemented a simple instant messaging application with a centralized name server/registry to maintain which clients were present on the system, and their current status (busy or available). Unfortunately, the centralized name server will not scale well if there is a large population of clients. One interesting way to approach this problem is to simply eliminate the centralized name server and implement a decentralized name service based on a DHT.

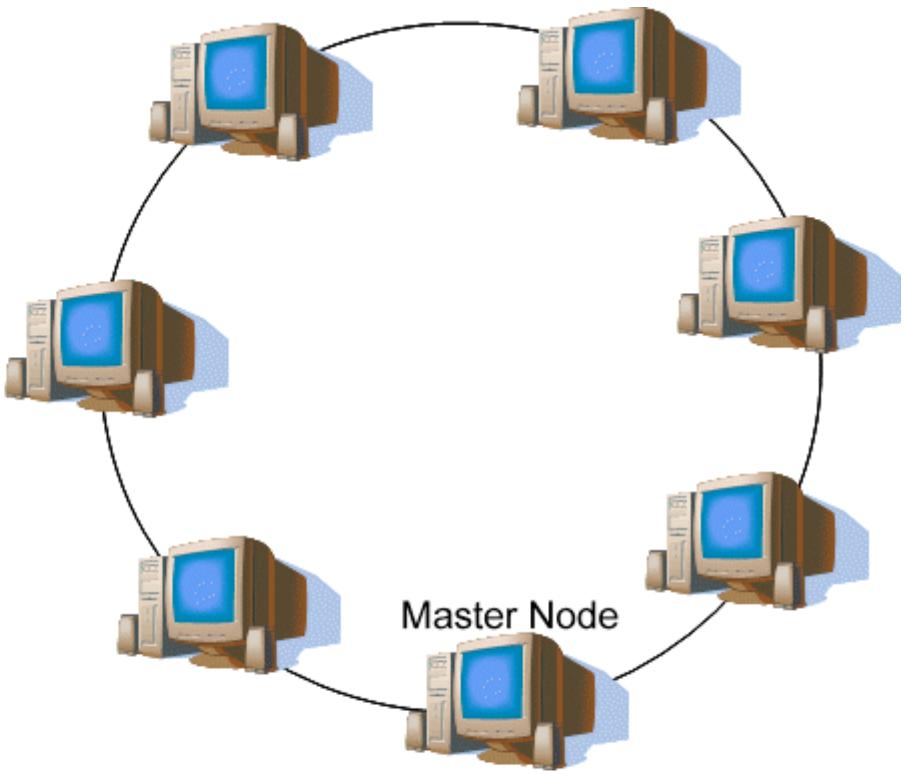

Figure 1. Decentralized P2P IM Architecture

In this lab exercise you will use OpenChord (version 1.0.4), a Java implementation of the Chord P2P architecture that we have discussed in previous lectures, to replace the centralized Java-RMI based name server you implemented in the previous exercise.

The architecture of the instant messaging system you will build is shown in Figure 1 above. Recall that Chord implements a DHT by arranging the participating nodes into a ring overlay. Hence, in your implementation you will not have a centralized name server process, rather each IM client when it starts up will join as a participant in a Chord network and share the burden of implementing the name server functionality with the other computers that are participating.

In order to bring the Chord network into existence, the first client you will startup will be called the **master**, and is responsible for creating the Chord network. All subsequent nodes that join will simply bootstrap themselves via the master node initially. Anytime a client terminates, it must cleanly remove itself from the Chord network[1].

Your client application will be comprised of essentially two *software components* running within a single process on a single machine:

1. Presence Service – The Presence Service is a very thin layer of code implemented on top of the OpenChord platform. The abstract interface (PresenceService.java) and related data structures (RegistrationInfo.java) have been specified *a priori*, and can be downloaded from the class website. Note that these files are very similar to the analogous files in the previous exercise. This will actually make your life really simple in that approximately 95% of the code you wrote in the previous lab can be reused! The presence service will be instantiated within your client, it will not run as a separate process as it did in the previous lab.
2. ChatClient - The Chat Client is the application that end users will run when they wish to see who is on-line, and/or have on-line discussions with other users. The ChatClient will resolve names of other users to address/port using the Chord-based Presence Service, but will exchange messages directly with other clients via Java Sockets.

The ChatClient process will be expected to conform to the following command conventions:

```
java [necessary java flags go here] ChatClient [-master] {user} {host}
```

where *user* refers to the name of the user invoking the client, and *host* refers to the address of the master node from which the client will bootstrap itself. The optional *–master* option is added when you run the initial client to designate it as the master node. Upon startup, the ChatClient will register its presence with the Presence Service. In

[1] Note that the master node itself in theory can leave the network, and things will keep functioning fine with the remaining nodes. However, once the master leaves no more nodes can join the network, so we will assume the master node is a special node that will participate indefinitely.

essence, this means some peer somewhere in the Chord network will be maintaining the mapping from the user's name to the user's host/port.

The ChatClient will implement a very simple command-line-interface that interprets and executes four[2] commands.

- talk {username} {message} - When this command is entered, the client 1) first checks to see if the user is present *and available* (via the PresenceService). 2) If the user is registered and available, a connection to the target client is established, based on their registration info, and they are sent the given message. Note that when a client receives a message, it will simply print it out to the console, and re-prompt the user to enter his/her next command. (i.e. you will need multiple threads of execution here within your client process.)
- busy – The client updates its registration with the presence server, indicating it is not currently available. If the client is already in not available when this command is entered, nothing needs to be done, though it would be good to prompt the user and indicate they already are not available. A client that is busy should not receive any messages.
- available – The client updates its registration information with the presence server, indicating it is now available. If the client is already available when this command is entered, nothing needs to be done, though it would be good to prompt the user and indicate they are already registered as available.
- exit – When this command is entered, the ChatClient will unregister itself with the PresenceService and *leave* the Chord network. The client will

Implementation Details

Your implementation strategy is to implement the PresenceService interface on top of the Chord APIs. Once you have accomplished that, you will simply replace the RMI boilerplate code (e.g. the code in which you bound to the remote PresenceService implementation via RMI) in your existing client with code to create the Chord network if the client is the master, or bootstrap yourself into the existing Chord network if not.

All of your user interface and messaging code should remain intact if you properly utilized the interfaces defined in the previous project.

Deliverables

There are two deliverables required in order to receive full credit for this lab exercise.

[2] We've eliminated the *broadcast* and the *friends* command. There is no easy/efficient way to get a list of all keys registered in the DHT, nor should there be, since in a real IM system each user is only interested in a subset of registered users. If your client were to maintain a local list of "friends" that it was interested in these commands could be easily implemented, but this is not required.

Provide Demo to Instructor
Provide a demo of the working system to the instructor during office hours before or on the designated due date.

Electronically Submit the Source Code
Archive all the source code needed to build and execute your implementation in a zip file and submit it to Blackboard. The timestamp on your submission must be no later than the due date to receive credit.

Resources

You will want to consult the following references – all of which are posted on the class blackboard site for your convenience:

- I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, Chord: A scalable peer-to-peer lookup service for internet applications," in Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications. ACM Press, 2001, pp. 149-160. [Online]. Available: http://citeseer.ist.psu.edu/442155.html
- OpenChord version 1.0.4 - http://tinyurl.com/ysjsdg
- Sample OpenChord code covered in lecture.