Coursework 2: IM over Open DHT

Due date: 12 noon, 25th April, 2006

In this coursework, you will write a simple instant messaging (IM) application. The coursework is worth a total of 7 marks, and represents 7 percent of your final grade for 3C36/D15. There are two components to this coursework: writing a complete, correctly functioning, and well documented IM application (5 marks), and answering a design-related question about how to improve the IM application (2 marks).

As you are undoubtedly already aware, instant messaging allows two users sitting at Internet-connected hosts to send short textual messages to one another. Each user is known by a globally unique *user ID*, a short string that is used to identify the recipient of an instant message.

To write your IM application, you will use two networking abstractions taught in lecture: TCP sockets, and a distributed hash table (DHT). You are to write your IM application in Java.[1]

There are two main problems to solve in instant messaging:

- **Rendezvous:** Two users who wish to IM one another know only one another's user IDs. They may be sitting at any hosts on the Internet, and thus, neither user knows the IP address for the other. Yet each user must be able to address his packets so that they reach the other user. Thus, the users must *rendezvous* with one another by an application-level name, although the underlying communication system (the Internet) does not address traffic by such names.

- **Reliable, in-order delivery:** When a user sends a textual IM to another user, the bytes of the text must arrive intact, without being lost, duplicated, reordered, or corrupted. A *reliable byte stream* transport protocol, like TCP, can help ensure that these reliability constraints are met.

Approach: Rendezvous and Sending Messages

DHTs are useful for solving the rendezvous problem, in that the key-value storage they provide is useful for *indirection*. Suppose that user `fred` sits down at the Internet host with IP address 128.16.64.22. If a DHT were running on a collection of Internet hosts, and it were available to all Internet users, `fred` could invoke the DHT to store the binding of his user ID to his current IP address:

[1] If you strongly prefer to write code in C or C++, you are permitted to do so, but **be warned:** (1) You should only do so if you are already proficient in the language you choose; 3C36/D15 is not a course in how to program in C or C++, and the course staff will not answer basic questions about how to program in either. (2) The instructions below are written assuming you will be programming in Java. If you write in another language, you will be responsible for figuring out how to communicate with the DHT in that language.

H = hash(`fred`)
DHT-put(H, 128.16.64.22)

Suppose that some other user wishes to IM `fred`. If he can communicate with the same DHT as `fred`, he can learn `fred`'s IP address as follows:

H = hash(`fred`)
IP = DHT-get(H)

Here, the DHT serves as a sort of Internet-accessible memory, used by IM sender and IM receiver to achieve rendezvous using an application-level name (user ID).

Once the sender knows `fred`'s IP address, he can open a TCP connection to `fred`. Doing so requires action by both sender and receiver. First, `fred`, the receiver, must open a socket that listens for connections on some *well known destination port*.[2] All IM requests are sent to this destination port, to distinguish them from requests for other applications that use TCP (in the same way that HTTP requests are in most cases sent to destination port 80). Second, the sender must open a TCP connection to `fred`'s IP address and the well known IM destination port. The sender can then send his message to `fred`.

Implementation: Sockets, Open DHT, and XML RPC

You should write two entirely separate programs.

The first, `imrecv`, will be run by a user who wishes to receive instant messages. It should be invoked with the following arguments:

```
imrecv <myname>
```

where `<myname>` is the user ID under which the user wishes to receive messages. `imrecv` should put the user's IP address in the DHT, then listen for connections on a TCP socket at the well known port for the 3C36/D15 IM application, which is hereby defined as 3615. `imrecv` should then forever (until it is terminated) accept a connection, repeatedly read the next line of input from the connection and print that line to the console, until the connection closes, and then accept the next connection, and so on.

The second, `imsend`, will be run by a user who wishes to send instant messages. It should be invoked as follows:

```
imsend <myname> <dstname>
```

[2] Note that the receiver could actually choose any available port on his host; he could then publish the port on which he listens in the DHT, along with his IP address, so that senders would know which destination port to use to reach him. In the interest of simplicity, your code should just use a well known destination port.

where `<myname>` is the user ID of the sender and `<dstname>` is the user ID to which messages should be sent. `imsend` should get the IP address associated with `<dstname>` from the DHT, then open a TCP connection to that IP address on port 3615. `imsend` should then loop, accepting a line of input from the user, sending it over the TCP connection with `<myname>:` prepended, and so on. The user enters a single period (".") to indicate he has finished sending messages, at which point `imsend` should close the TCP connection and terminate.

All the work of building and deploying a DHT has already been done for you; you are to use the Open DHT public DHT service, described at `http://opendht.org`. That site includes full documentation for Open DHT; click on the "User's Guide" link for a primer on how to use the service.

Briefly, Open DHT is DHT software that runs on 200–300 Linux hosts across the Internet. These hosts are deployed on five continents as part of PlanetLab, a platform used by computer scientists to experiment with new types of distributed systems (like DHTs). Open DHT runs 24/7, and is available to all to use, over the familiar "put(key, value)" and "value ← get(key)" interface described in lecture.

There are a few important details you need to know in order to use Open DHT:

Soft state When you put data into Open DHT, it won't stay there forever; if it did, Open DHT would quickly run out of storage. Instead, each time you put data into Open DHT, you must specify a *time to live (TTL)* for the data. The TTL is the duration you would like the data to remain in the DHT. When the TTL expires, Open DHT will remove the (key, value) pair you've put. You specify the TTL to Open DHT in seconds. The maximum TTL Open DHT supports is 604,800 seconds, or one week.

Given that Open DHT spontaneously deletes stored (key, value) pairs when their TTLs expire, you may wonder how one can keep data stored in Open DHT for a longer period. The solution is to *re-put* a (key, value) pair before its TTL expires. Open DHT will then enforce the TTL from the time of this new, later put.

Note that you need not explicitly delete (key, value) pairs from Open DHT; you can choose a TTL that is relatively short (where "short" depends on the nature of your application), and simply stop re-putting the value. This point bears consideration—you must choose how long a TTL to use when putting a (user ID, IP address) pair into Open DHT.

The main consequence of soft state for your IM application is that `imrecv` will need to re-put the (user ID, IP address) pair sufficiently often to keep the pair stored in the system, so that other users can look up the user ID. You may find that it is easiest to do so by running a separate thread that alternates sleeping and re-putting.

XML RPC The easiest way by far (and the one you should use for this coursework) to communicate with Open DHT is via XML RPC, carried over a TCP connection. *Remote Procedure Call (RPC)* refers to the general technique of a network-attached host invoking a procedure on another network-attached host. A client issues an RPC request to a server, which responds with an RPC response. The RPC protocol formats the client's arguments

to the procedure for transmission over the network, and decodes them at the server; it does the same for the server's response in the reverse direction. Each RPC literally corresponds to an invocation of a procedure. In the case of Open DHT, the DHT service makes two procedures available to clients: put and get, the basic two operations supported by a DHT, as described in lecture.

As its name suggests, XML RPC uses XML to express RPCs. Because XML is a text-based protocol, its format is fairly easy to read. The Open DHT User's Guide on the web gives the full details of the syntax of the XML RPC calls one can make to invoke Open DHT. Because it is tedious to write code to format and parse XML RPC requests and responses, you should use a ready-made library to do this work for you. The most widely used XML RPC library for Java is Apache XML-RPC, produced by the same organization that produces the popular Apache web server. You should download and compile the Apache XML-RPC library, and use it to generate the put and get RPCs that your IM application sends to Open DHT. Documentation and code for Apache XML-RPC is available at the project's web site, `http://ws.apache.org/xmlrpc/`. The "first example" on the "Client Classes" page shows you practically all you need to know to send RPCs. Note that you will *only* be using the client-side interface; Open DHT implements the server side already.

Gateways Clearly, to send an RPC, you must know the Internet host name or IP address of the destination RPC server. All hosts that participate in Open DHT are *gateways* that accept all inbound XML RPC requests that conform to the Open DHT API. When you set up an XML RPC client using Apache XML-RPC, you must configure the URL of the server (see `setServerUrl` in the first example on the "Client Classes" page on the Apache XML-RPC web site). You can find a list of the currently running Open DHT servers at `http://opendht.org/servers.txt`. As is explained in the Open DHT User's Guide on the web, Open DHT accepts XML RPC requests on port 5851. So an example server URL for Open DHT would be:

```
http://planet2.pittsburgh.intel-research.net:5851/
```

Sockets The standard `java.net` class hierarchy includes an easy-to-use API for TCP sockets; this is the API that was briefly covered in lecture. Note that only one instance of the `imrecv` process at a time may run on the same Internet host; that's because only one process can bind to the same destination port on the same host.

Your own IP address `imrecv` will need to know the IP address of the host it is running on, in order to put it into Open DHT. Java provides a simple API to learn a host's IP address; `InetAddress.getLocalHost()` does the job.

SHA-1 To use a DHT correctly, one typically hashes a key to obtain the DHT ID for that key; it is the DHT ID that is used as the first argument in a put.[3] Open DHT accepts IDs that are 160 bits (20 bytes) long; this is precisely the length of all outputs of the SHA-1 hash function. You can find an easy-to-use implementation of SHA-1 in the standard `java.security` class hierarchy. A simple example of using SHA-1:

```
use java.security;

MessageDigest md = MessageDigest.getInstance("SHA-1");
md.update("Brad");
byte[] ID = md.digest();
```

Example code you should not follow There is already a full implementation of IM for Open DHT available on the web at:

`http://opendht.org/ohchat-0.03.tar.gz`.

That code is in C++, and is written atop a C++ library known as `libasync`, that supports a style of programming known as *event-driven programming*. In event-driven programming, a client sends requests to a server asynchronously; it does not wait for a reply from the server before continuing with other processing, and processes responses from the server as they arrive using *callbacks*.

> *You need not understand event-driven programming. On the contrary, the code you submit for this coursework must* not *use event-driven programming, but must instead use the straightforward approach where the client sends a request (*e.g., *a get RPC) to the server and waits for the response from the server before continuing.*

You are welcome to peruse the event-driven IM code described above, but you will likely find it of limited use in programming your non-event-driven IM application in Java.[4]

To get full marks, your code should run correctly, implement all the functionality described above, and be well engineered (cleanly readable and commented). You must also turn in a design description (no longer than one side of A4) that explains your code; see below for "what to turn in."

Good luck!

[3] There is some ambiguity in the way different documents name keys and IDs. The important thing to remember is simply that when you call put, the first argument should be the output of a hash function. Thus, if you wish to store the pair (Brad, 15), you would invoke put with arguments (hash(Brad), 15).

[4] Event-driven programming is an extremely powerful abstraction, and one used in many high-performance servers. It is regrettably beyond the scope of 3C36/D15; if you're interested after the course is over, have a look at `http://pdos.csail.mit.edu/6.824-2004/async/`.

Essay Question: IM through NATs

Part of the coursework is to answer the following question (in no longer than one side of A4):

> Suppose that two users who wish to IM one another are both behind NATs. The IM application you've built will not work properly in this case. Why not? Propose an alternate design for an IM application that *would* work correctly for users behind NATs. *Hint: Open DHT is a generic tool for storing data such that others may retrieve it.*

What to Turn In

- Email a tarball (.tar.gz) or ZIP archive containing your *complete* source code (.java files for code you wrote, and .jar files for any extra libraries you used; no .class files) to `3c36cw@googlemail.com`. By "complete," I mean that I must be able to run your code on any machine with a standard Java installation. Any libraries you use that are not part of the standard Java class hierarchy must be included in the archive you submit.
- Submit to the 5th floor Computer Science reception desk the following hardcopy items:
  - a completed coursework cover sheet
  - a complete hardcopy of all your code (do not include code from any libraries you used)
  - a list of all libraries you used that are not part of the standard Java class hierarchy, including where you obtained the library (web page describing the library is fine), and a succinct (no more than two sentences) description of what you use the library for
  - a design description (no longer than one side of A4 in 10-point type) that explains in clear, well written English how your code works (**application code + design description: 5 marks**)
  - your answer to the above question about NATs (no longer than one side of A4 in 10-point type) (**answer: 2 marks**)

Academic Honesty

You are permitted to discuss your code with your classmates, and to help one another debug. You are also permitted (and indeed, expected) to use an XML RPC library written by others to format requests to Open DHT, and parse responses from Open DHT. As one

always does in an academic setting, you must acknowledge the work of others explicitly. In this case, that means you must indicate in your design desccription document what XML RPC library you used and who (or which open source project) wrote it.

All other code that you submit must be written entirely by you alone.

Copying of code from student to student is a serious infraction; it will result in automatic awarding of zero marks to all students involved, and is viewed by the UCL administration as cheating under the regulations concerning Examination Irregularities (normally resulting in exclusion from all further examinations at UCL). The course staff use extremely accurate plagiarism detection software to compare code submitted by all students and identify instances of copying of code; this software sees through attempted obfuscations such as renaming of variables and reformatting, and compares the actual parse trees of the code. Rest assured that it is far more work to modify someone else's code to evade the plagiarism detector than to write code for the assignment yourself!

Read the Mailing List

Please monitor the course mailing lists, `{3c36,d15}@cs.ucl.ac.uk`, during the period between now and the due date for the coursework. Any announcements (*e.g.,* helpful tips on how to work around unexpected problems encountered by others) will be sent to the lists.