EECS 489 Lab 4: DHT *O*(*N*) Case

This assignment is due on Wednesday, 11 Feb 2015, 10 pm.

Introduction

In this lab, we will implement a simplified, Chord-like distributed hash table (DHT) that takes *O*(*N*) time to add a new node to the DHT. Instead of relying on a periodic process to fix inconsistencies in the DHT arising from node additions, we will rely on on-demand correction of such inconsistencies. The app `dhtn` built from the support code is our distributed hash table node, with the following command line options:

```
% dhtn [ -p <node>:<port> -I <ID> ]
```

If the node is run without the `-p` option, it forms a new DHT with itself being the only node overseeing the whole identifier space. The `-p` option specifies the target of the node's message to join an existing DHT. A node's position in the DHT may not end up being adjacent to the target node. A node's ID is based on the SHA1 hash of its IPv4 address and port number. As in Lab 3, we assume 8-bit identifiers. The `-I` option allows you to override the ID computation and give the node a static ID. This allows you to test how your code handles ID collision. It also allows you to test specific node addition order and scenarios.

A node is placed between its predecessor and successor in the identifier ring if its ID is larger than that of the predecessor and smaller than that of its successor, where the ordering of the IDs follow modulo arithmetic as in Lab 3. Entering '`p`' on the standard input (console), if not on Windows, prints out the node's and its predecessor's and successor's IDs (don't forget to hit `enter` or `return` after the '`p`').

Assumptions

We make some simplifying assumptions to make this lab more manageable:

- No node departure: once a node joins the DHT, it doesn't depart until you take down the whole DHT. This means that, as in PA1, you don't need to clean up after a node departure, only to make sure that you can take down the DHT without node crashing.
- Node join process does not fail. To assume otherwise would require a bit more complicated 2-phase commit join protocol.
- No concurrent joins. Nodes are added one a time. The provided support code most likely will work with concurrent joins, but it has not been tested for it. Moreover, until a node has completed its join process, it will interpret receiving a join packet from another node as an error.
- Everytime a node needs to send a message, it will open and then close a new connection to the target node. So there is no permanently opened connections, unlike in PA1. The only exception to this single message per connection rule is when performing on-demand correction of DHT inconsistency due to node addition, as explained below.

Join Protocol

When the first node runs, it starts a listening socket, prints out its FQDN:port on the screen and waits for join requests. Subsequent nodes should be started with an existing node's FQDN:port provided on the

command line. Each subsequent node first creates its own listening socket and then connects to the provided node, sending over a join request with its own ID, IPv4 address, and listening port number. See the structure `dhtmsg_t` in `dhtn.h` for the packet structure. The version number MUST be `DHTM_VERS`, as defined in `netimg.h`. The `dhtm_type` field encodes the type of packet. The different types of packets all use the same packet format, except for type `DHTM_WLCM`, which contains two `dhtnode_t` in the packet instead of just one. The first `dhtnode_t` in the welcome packet tells the newly joining node its successor in the identifier ring and the second `dhtnode_t` its predecessor.

After a node has sent out its join packet, it goes into a `select()` loop waiting for a connection to arrive at its listen socket, or, if not on Windows, for input on standard input (console). Since we open a separate connection for each message, when a connection is established, we immediately go into receiving mode. To simplify the code, each new connection will use a random, kernel-assigned ephemeral source port. Use the node's ID, instead of its port number, to identify a connecting node. [As a thought exercise, think what you would have to do if each node must use a fixed source port number (and be glad that you're not required to implement it ;-).]

There are two possible outcomes to a join attempt: either there is an ID collision and the node is asked to generate a new ID and try again (i.e, it receives a `DHTM_REID` packet), or it receives a welcome message, informing it of its successor and predecessor nodes (in that order in the `DHTM_WLCM` packet). Thus in `dhtn::handlepkt()` we check whether the returning packet is of type `DHTM_REID` or `DHTM_WLCM`. In the former case, we call `dhtn::reID()` to regenerate a new ID and then call `dhtn::join()` again with the new ID to retry the join attempt. In the latter case, we store the first `dhtnode_t` in the return packet at the `dhtn` class member variable `finger[0]`, which is where we keep the successor node's information. Then we store the second `dhtnode_t` in the class member variable `finger[DHTN_FINGERS]`. We use a `finger[]` array instead of separate sucessor and predecessor variables in anticipation of PA2.

In addition to receiving these two types of packets, a node could also receive a `DHTM_JOIN` packet. In which case, in `dhtn::handlepkt()` we simply call the function `dhtn::handlejoin()` to handle the arriving JOIN packet. The function `dhtn::handlepkt()` has been provided to you in full. Please take your time to read it carefully and make sure you understand what it is doing. Pay attention to how the socket whence a packet arrives is closed as soon as we are done with it. The only exception is when the arriving packet is a `DHTM_JOIN` packet, in which case we pass the socket along to `dhtn::handlejoin()`—we still need to communicate with the sender when handling a `DHTM_JOIN` packet. In `dhtn::handlejoin()` you must close the sender socket as soon as you're done with it. Otherwise, you could run into a dead lock situation where multiple nodes are waiting for each other to complete transmission and close connection.

Task 1

Your first task is to write the function `dhtn::handlejoin()`. There are four cases you need to consider in writing this function:

1. When the joining node's ID collides with that of an existing node.
2. When the correct spot has been found on the identifier ring for the joining node.
3. When the sender's successor information has become inconsistent due to node addition.
4. When a `DHTM_JOIN` message must be forwarded along the DHT.

Please see the comments in `dhtn.cpp:dhtn::handlejoin()` for what you need to do for each of the above cases. Note that in the second case, the current node's ID range is split with the joining node. You will need to call `imgdb::reloaddb()` to reload the image database to account for the new ID range (see the comments in `dhtn::handlejoin()`). This task takes no more than 30-40 lines of code. You will need your

`hash.cpp:ID_inrange()` code from Lab 3.

Task 2

For the fourth case in Task 1, you should call `dhtn::forward()` to handle the message forwarding. The `dhtn::forward()` function is where we implement the "on-demand" repair of the DHT identifier ring inconsistencies arising from node additions. The original Chord algorithm relies on a periodic process to fix such inconsistencies. We do it on-demand instead. Again, please see the comments in `dhtn.cpp:dhtn::forward()` for what you need to do to implement this function. This task takes no more than 15-20 lines of code.

In completing both tasks, you may want to use the function `socks.cpp:socks_clntinit()` (please see associate comments in the code). You may also find `socks_close()` useful if you want to maintain Windows portability to your code.

Testing Your Code

The provided `dhtn` is linked with the `imgdb` class so you can use `netimg` from Lab 3 to query it for an image. To simplify experimentation, you can let all instances of your `dhtn` to share the same `images` folder, but each `dhtn` instance should load to its image database only those images whose IDs fall within its purview. We are not implementing search on the distributed hash table in this lab, so only query for images within a `dhtn`'s ID range may return an image. In your test, you can run `netimg` multiple times, each time connecting to a different `dhtn`, requesting images within and outside the node's ID range.

Since we don't have search implemented, after each node addition, the successor information on earlier nodes may not get corrected such that when you hit '`p`' you will see inconsistent successor information. This is alright. It will be corrected in PA2. However, for each new node you add, entering '`p`' at that newly joined node should show you the correct successor node. Only after additional nodes have joined the network is the successor information of existing nodes allowed to be inconsistent. While the successor information is allowed to become inconsistent, the predecessor information must stay consistent at all times. So if you hit '`p`' on each node, you should be able to reconstruct the correct identifier ring by following the predecessor node information at each node.

Support Code

The support code is available as lab4.tgz in the Course Folder. The support code contains only three files: a `Makefile`, `dhtn.h`, and `dhtn.cpp`. To build the `dhtn` program, you need your files from Lab 3. You could simply copy over these new files from Lab 4 to your Lab 3 folder. If you want to save the `Makefile` from Lab 3, please take precaution to save it before you copy over and overwrite it with the new `Makefile`. As with Lab 3, only those who have completed PA1, or inform us that they are not going to complete PA1, will get access to the support code because the support code reveals the solution to parts of PA1.

You can also find `refdhtn` in the course FILES folder, which, as usual, was compiled on CAEN eecs489 hosts running GNU/Linux, so don't try to run it on Debian, Ubuntu, Mac OS X, or Windows machines. The support code has been compiled and tested on Linux and Mac OS X and Windows. As with Lab 3, on Windows, you'd need to install the OpenSSL library (see the relevant section of the *Building Socket Program* course note for installation and usage instructions.)

For the provided support code, we assume you have working `ID_inrange()` and `searchdb()` functions

from Lab 3. If you didn't manage to get these functions to work in Lab 3, you can get the solutions for 10 of your PA2 points.

Submission Instructions

Test your compilation on CAEN eecs489 hosts! Your submission must compile and run **without** errors on CAEN eecs489 hosts.

Your "*Lab4 files*" comprises your `dhtn.cpp` file.

To turn in your Lab4:

1. Submit the SHA1's of your *Lab4 files* on the CTools Assignments page. Once you've sent in your SHA1's, don't make any more changes to the files, or your SHA1's will become invalid.
2. Upload your *Lab4 files* by pointing your web browser to Course folder and navigate to your `lab4` folder under your uniqname. Or you can `scp` the files to your `lab4` folder on IFS: `/afs/umich.edu/class/eecs487/w15/FOLDERS/<uniqname>/lab4/`.
   This path is accessible from any machine you've logged into using your ITCS (`umich.edu`) password. Please report any problems to ITCS.
3. Keep your own backup copy! Don't make any more changes to the files once you've submitted your final SHA1's.

The timestamp on your SHA1 submission on CTools' Assignments page will be your time of submission. If this is past the deadline, your submission will be considered late. You are allowed multiple "submissions" without late-policy implications as long as you respect the deadline. CTools keeps only your last submission.

Do NOT turn in an archival (`.zip` or `.tgz`) file, instead please turn in your solution files individually. Turn in ONLY the files you have modified. Do not turn in support code we provided that you haven't modified. Do not turn in any binary files (object files, executables, or images) with your assignment.

Do remove all `printf()`'s or `cout`'s and `cerr`'s and any other logging statements you've added for debugging purposes. You should debug using a debugger, not with `printf()`'s. If we can't understand the output of your code, you will get zero point.