EECS 489 PA2: DHT Search

This assignment is due on Mon 23 February 2015 at 10 pm.

Overview

You are to implement a simplified, Chord-like distributed hash table (DHT) in this programming assignment. The specification of this assignment relies heavily on the specifications for Labs 3 and 4. You are expected to be familiar with the specifications for those labs already. To differentiate the program for this assignment from that of Lab 4, we will call the one for this assignment, `dhtdb`. The assignment consists of the following graded tasks.

Graded tasks (100 points total)

1. Basic DHT construction (30 pts)
2. DHT construction with finger table (20 pts)
3. Image database load, search with Bloom Filter, and display (10 points)
4. Image search on the DHT, with caching (40 pts)
5. Writeup

Assumptions

We will make several simplifying assumptions in line with Labs 3 and 4, only the last two assumptions are new:

1. Aside from graceful teardown of the DHT, we will assume no node departure or failure. Once a node is part of the DHT, it will stay a part of the DHT until the whole DHT is torn down. So when a connection to a node is down, we can assume that the DHT is being torn down and simply close the connection.
2. Node join process does not fail. To assume otherwise would require a bit more complicated 2-phase commit join process.
3. No concurrent joins. Nodes are added one a time. The provided supported code will most likely work with concurrent joins, but it has not been tested for it. Consequently, until a node has completed its join process, it will interpret receiving a join packet from another node as an error.
4. Everytime a node needs to send a message, it opens a separate connection with the target node and immediately closes the connection once the message is sent. So there is no continuously open connections. The only exception is when performing an on-demand correction of DHT inconsistency due to node addition, as explained below.
5. The whole image database is available at each DHT node, but a node is only allowed to serve up images whose IDs are within its range in the identifier space or if it has the image "cached," as explained in the caching task below. This allows you to run multiple nodes from the same folder and all instances of the node will have access to the same "images" folder containing all the images.
6. Object ID size is 8 bits. To compute an object ID, we "fold" a 160-bit SHA1 value up into 8 bits. So the probability of IDs colliding become much higher. For the images, once we have a hit on the Bloom filter, we simply do a linear search of the database. A match requires matching both the image's ID and name, which also detect any hashing collision (false positive).

7. The image database has a fixed maximum size of `IMGDB_MAXDBSIZE`. Once this capacity is reached, we simply print out a message to inform the user that we're not adding more images, but the server continues to run otherwise.
8. Only one image is read into memory at any one time. Each time there is a search hit, the image will be read from file.
9. Once loaded, images are never removed. So, we don't have to worry about holes in the database or resetting the Bloom filter. When the ID range of a node changes, usually when its predecessor node changes, the whole image database is reloaded, its cache flushed, and its Bloom Filter recomputed.
10. Only one active search request is allowed per `dhtdb` node. If multiple `netimg` clients try to perform a search at a `dhtdb` node at the same time, queries subsequent to the first one are simply informed that the server is busy and have their connections closed.

Your Tasks

1. Basic DHT Constructions

This part of the assignment is covered by Lab 4. You may re-use both the support code and your code from Lab 4. We will use SHA1 to generate node and object IDs, but we will limit the hash key size to 8 bits. When a node is started without another node provided in the command line, it is the first node on the DHT and it will assume the full range of the identifier space. If the ID of the first node is 128 and a node with ID 132 joins the DHT next, the new node will assume ID range [129-132] and node 128's range becomes [133-255, 0-128]. If next a node with ID 64 joins the DHT, it will assume ID range [133-255, 0-64] and node 128's range becomes [65-128]. Since we fold SHA1's 160-bit hash keys up into 8 bits to create our IDs, we massively increase the probability of two IDs colliding. When a new node's ID collides with that of an existing node, we rejects the node and it simply obtains another port number and try to join again with a different ID. Description of the join protocol, along with the packet formats, and support code can be found in Lab 4.

2. DHT Construction with Finger Table

This part of the assignment has you generalize your Lab 4 code to use finger tables. The original Chord algorithm runs a periodic process to fix broken fingers. In this assignment, we do the fix-up on-demand, using the same mechanism used in Lab 4 to fix the predecessor info. A finger table allows a node to keep shortcuts/pointers to nodes at various distances away from itself. As with the original Chord paper, we keep a pointer to the closest node succeeding ID+2^*i*, where ID is the identifier of the current DHT node and *i* ranges from 0 to 7, assuming an 8-bit identifier space. I will refer to the finger table as `finger[]` henceforth. To keep the finger table up to date after node additions to the DHT, you may find it convenient to define finger table of size one larger than necessary to hold the predecessor information at the top (largest index) of the table such that `finger[0]` is the immediate successor node and `finger[DHTN_FINGERS]` is the immediate predecessor node. In addition to the finger table, it may be convenient to keep a separate finger identifier table to store the identifier associated with each finger (the ID+2^*i*'s above). I will refer to this table as `fID[]` henceforth. This way you don't have to recompute the ID+2^*i*'s everytime you need one. You can simply look it up in the `fID[]` table.

When a node first forms **or joins** a DHT, set all its fingers to point to itself. When a node accepts a joining node, it make the joining node its predecessor as in Lab 4. Similarly, the joining node gets as its successor the node accepting its join request; and its predecessor is the accepting node's old predecessor. (See the discussion on `fixup()` and `fixdn()` below for how the rest of the finger table is updated.) When

forwarding a join packet, instead of simply forwarding to the successor node, a node first finds the largest index, `j`, for which `fID[j]` is in the range `(current node's ID, joining node's ID]`, in modulo arithmetic, and forwards the join packet to `finger[j]`.

Let's look at an example (you may want to make this one of your first test cases). Say your node ID is 23 and thus your `fID[]` contains: 24, 25, 27, 31, 39, 55, 87, 151. The finger table does NOT necessarily record consecutive nodes on the identifier ring. In this example, we may have nodes with IDs 40, 43, 56. The finger table entry corresponding to fID 39 is node 40 and the entry for fID 55 is 56. If now a join request arrives with ID 42, you want to forward it to node 40 (fID 39), not node 56 (fID 55). If you send the request to node 56, you would have missed node 43, which in this case is the correct node to forward the join request to.

In the *O*(*N*) case (Lab 4), each node points to its *immediate* successor, with no potential for skipping unseen node(s) between one node and its successor, so we can simply forward the join request to the next node with an ID that is subsequent to that of the joining node. In the present case that uses the finger table, there is a possibility of unseen nodes and we must therefore forward the join request to the largest ID that still precedes that of the joining node in the finger table.

As in Lab 4, if the ID of the joining node is expected to be in the range ending at the recipient node's identifier range, set the `DHTM_ATLOC` bit in the type field of the join packet. If it turns out that the identifier range is no longer in the purview of the recipient node, it sends back a `DHTM_REDRT` packet along with its current predecessor. In Lab 4, when a node receives a `DHTM_REDRT` packet, it makes the node returned in the packet as its new successor and retry sending the join packet to the new successor etc. until it receives no more `DHTM_REDRT` packet. With a finger table, instead of saving the returned node as the new successor, we save it in `finger[j]`, where `j` is as computed above.

Finally, define two functions, I call them `dhtn::fixup(int idx)` and `dhtn::fixdn(int idx)`. Given index `idx`, `fixup()` walks "up" the finger table from `finger[idx+1]` to `finger[DHTN_FINGERS-1]`. For each entry, `k`, `idx` < `k` < `DHTN_FINGERS`, it checks whether `fID[k]` is within the range between the node's ID and the ID of `finger[idx]`. If so, it updates `finger[k]` to `finger[idx]`. It stops walking "up" the finger table as soon as the identifier range check above fails. Conversely, `fixdn()` walks "down" the finger table starting from `finger[idx-1]` to `finger[0]`. For each entry, `k`, `idx` > `k` >= 0, it checks whether the ID of `finger[idx]` is within the range between the `fID[k]` and the ID of `finger[k]`. If so, it updates the entry of `finger[k]` to be that of `finger[idx]`. Unlike the `fixup()` case, `fixdn()` doesn't stop walking "down" the finger table until it reaches entry 0 or when `fID[k] == finger[k].dhtn_ID`, to ensure that fingers in the middle of the finger table will be updated with newly learned predecessor. We stop walking down the finger table when `fID[k] == finger[k].dhtn_ID` because in this case the entry is already correct but `ID_inrange()` always returns true when given the same ID as the beginning and end of range.

Everytime any of the finger table entry is updated, including when the first (successor) or the last (predecessor) entry is updated due to the receipt of a `DHTM_JOIN`, `DHTM_WLCM`, or `DHTM_REDRT` message, always call the `dhtn::fixup()` and/or `dhtn::fixdn()` functions as appropriate. This task should take on the order of 30 lines of code.

3. Image Database Load, Search with Bloom Filter, and Display

This part of the assignment is covered by Lab 3. You may re-use both the support code and your code from Lab 3. In Lab 3, you worked on a server that manages a database of images. The server is given a range of IDs for which it is responsible. If the filename of an image hashes to an ID within this range, the server enters the image's filename into its Bloom Filter. The server then listens for connections from clients. Once a client connects, it sends a query for an image file. The server looks up the image queried

in its Bloom Filter. If the Bloom Filter lookup returns a positive result, the server then searches its database for the image and, if found, returns the image to the client for display. Further description of this task and its support code can be found in Lab 3.

As in PA1, `dhtdb` listens to two sockets: the image socket to handle the image query from client, and the DHT socket to communicate with other `dhtdb`. It is convenient to make the image socket a member of the `imgdb` class and the DHT socket a member of the `dhtn` class. As in Lab 4, the ID range of a node changes everytime it gets a new predecessor. Be sure to reload a node's image database and reset its cache and Bloom Filter everytime its predecessor changes, by calling `imgdb::reloaddb(begin, end)`.

4. Image Search on the DHT, with Caching

In this task, we search the DHT for images that are not cached and that are outside the identifier range of the local node. We will be using the same search function used to construct the DHT with finger table. As part of the search process, we may need to fix up the finger table if it has been broken since it was last constructed. When the node the client connects to receives a positive search result, it "caches" the image. Since each node on the DHT has access to the full database of images, when a node "caches" an image, it simply loads into its database the filename of the image and enters the image's ID into its Bloom Filter. The next time the same image is queried, either locally or remotely, the image will be in its Bloom Filter and it can serve up the image instead of forwarding the search further.

When a `dhtdb` receives an image query from a `netimg` client, it first searches its local database and cache for the image. If the requested image is not found, it creates a `DHTM_SRCH` packet with itself as the originator of the query stored in the query message (`dhts_msg.dhtm_node`), then it computes the ID of the requested image name and stores it in the `dhts_imgID` field of the query packet, and copies the requested image name into the `dhts_name` field of the query packet. It then calls `dhtn::forward()` to forward the packet along onto the DHT. If you're using the support code from Labs 3 and 4, you may want to split this task between the `imgdb::handleqry()` and a `dhtn` search function of your own. Forwarding a `DHTM_SRCH` message follows exactly the same logic as forwarding a `DHTM_JOIN` message. The `dhtn::forward()` in Lab 4 support code allows it to be used to forward both types of packet. When forwarding a join packet, you want to use the ID of the joining node to determine where to forward the packet. When forwarding a search message, you want to use the ID of the queried image to determine where to forward the packet. When constructing the search packet, don't forget to set the version and type fields appropriately. We allow only one outstanding search at each `dhtdb` node at any time. You would need to enforce this. This task should take about 20 lines of new or modified code. If you've decided not to use the support code, your search message MUST follow the `dhtsrch_t` packet format defined in `dhtn.h`.

Next you need to update the `dhtn::handlepkt()` function to recognize `DHTM_SRCH` packet. Upon receiving a `DHTM_SRCH` packet, you first check your local cache/database for the image. If the image is found, you send a `DHTM_RPLY` back to the query originator and you're done. If the image is not in cache, you check whether the image's ID is within your range. If so, the image doesn't exist, and you send back a `DHTM_MISS` to the query originator and you're done. If the image is not in cache and its ID is not in your range and the node forwarding the search to you is not expecting the ID to be within your range, you simply forward the search using `dhtn::forward()`. However, if the queried image's ID is not within your identifier range, but the node forwarding the search message expected it to be within your range, you must send back a `DHTM_REDRT` message, same as the case for handling join packets. To iterate: if a queried image is found in cache, or it is in your range but there's no such image, you simply send a `DHTM_RPLY` or `DHTM_MISS` back to the query originator. You don't forward the search message further and you don't need to fix any existing inconsistencies in your finger table (if you do, you could end up sending multiple

replies to the query originator). This task should take no more than 40 lines of code. [NOTE: in the `dhtn.h` released with Lab4, there's a comment that refers to `DHTM_SRCH` and `DHTM_RPLY` as `DHTM_QUERY` and `DHTM_REPLY` respectively. Sorry.]

If the image is found, you receive back a `DHTM_RPLY` packet, otherwise you receive back a `DHTM_MISS` packet. If you receive a `DHTM_MISS` packet, you send back to the client a `imsg_t` packet with `im_found` set to `NETIMG_NFOUND`, using `imgdb::sendimg()`. Unlike in PA1, a `DHTM_RPLY` packet doesn't actually transfer any image between `dhtdb` nodes, instead you should think of it simply as a "permission" for a `dhtdb` node to load an image from the images folder to its database and to subsequently serve that image as if it were part of its database. When you receive a `DHTM_RPLY` packet, you can call `imgdb::loadimg()` to load the image into your database, which is acting as your cache, and then call `imgdb::readimg()` to read the image into memory. To send the image to the client querying the image, use `imgdb::marshall_image()` and `dhtn::sendimg()` as is done in `imgdb::handleqry()` for an image found locally. Finally, don't forget to close the socket connecting the client and allow your `dhtdb` node to serve the next client. Given the member variables and methods the handling of `DHTM_RPLY` and `DHTM_MISS` packets need to access, you may want to make these two handler functions members of `imgdb` class. This task should take on the order of 35 lines of new code.

Testing Your Code

The description for the second task above contains a test case you can use to test your finger table implementation. You may want to extend the `dhtn::printIDs()` function to print out your finger table also. Watch how your join packet is forwarded between existing nodes and make sure it is doing what you're expecting it to. It may help to draw the ring you're trying to build first. Then as you add each node, double check that its join packet is being forwarded correctly, and where there are finger table inconsistenies, they are being corrected; and check that the node finally attaches at the right location on the identifier ring. Once you're convinced that your ring construction works, image search should just work :-). Query a node that doesn't have an image locally and watch the search packet propagates correctly on your ring. Try querying the same image at the same node again to test your cache. Then try querying another node that doesn't hold the image but has the first node in its finger table and see if caching is working properly in this case also.

Submission Guidelines

As with PA1, to incorporate publicly available code in your solution is considered cheating in this course. To pass off the implementation of an algorithm as that of another is also considered cheating. For example, the assignment asks you to implement Bloom Filter and you turn in a working program that does a simple search without using Bloom Filter and you do not inform us about it, it will be considered cheating. If you can not implement a required algorithm, you *must* inform the teaching staff when turning in your assignment, e.g., by documenting it in your writeup. As with PA1, if you have not been able to complete Lab 3 and would like the solution so that you can complete this assignment, you may choose to forfeit the 10 points associated with it and obtain a solution from us. Similarly for Lab 4, for 30 points.

Test your compilation and build on CAEN eecs489 hosts! Your submission must compile and run **without** errors on CAEN eecs489 hosts. Code that does not compile on CAEN eecs489 hosts will be heavily penalized. Your solution must either work with the provided `Makefile` from Lab4 or you must provide a `Makefile` that works on CAEN eecs489 hosts. Your code should interoperate with the provided `refnetimg` and `refdhtdb` binaries in the Course Folder.

Create a writeup in **text format** that discusses:

1. Your platform and its version - Linux, Mac OS X, or Windows.
2. Anything about your implementation that is noteworthy.
3. Feedback on the assignment.
4. Name the file writeup-*uniqname*.txt.
   For example, the person with uniqname *tarukmakto* would create *writeup-tarukmakto.txt*.

Your "*PA2 files*" then consists of your `writeup-uniqname.txt`, and your source codes.

To turn in your PA2:

1. Submit the SHA1's of your *PA2 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 *PA2 files* by pointing your web browser to Course folder and navigate to your `pa2` folder under your uniqname. Or you can `scp` the files to your `pa2` folder on IFS: `/afs/umich.edu/class/eecs487/w15/FOLDERS/<uniqname>/pa2/`.
   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 of 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.

General

The General Advice section from PA1 applies. Please review it if you haven't read it or would like to refresh your memory.