

IMPORTANT:

On this lab, you have the option to work with a lab partner of your choosing. When choosing a partner, you are strongly advised to choose a partner whose ability is similar to your own. One way to assess this is by looking at your overall “score” in the class up to this point. If you and your prospective partner have very different scores, you should think twice before agreeing to work together on the lab. When partners have mismatched abilities, neither gets very much out of the collaboration. The stronger partner ends up doing the bulk of the work, while the weaker partner sits back and watches. It may seem contradictory, but two weaker partners will actually learn more from each other than they will if they are paired with stronger partners (and the point of these labs is for you to learn as much as possible).

IF YOU DECIDE TO WORK ON THE LAB WITH A PARTNER, YOU NEED TO RECORD YOUR TEAM BY WEDNESDAY 10/07/2015 NOON. SEND ME AN EMAIL (WITH COPY TO YOUR PARTNER) WITH THE NAMES AND STUDENT IDs OF THE TWO PARTNERS.

Review and follow the general instructions included in the Lab 1 writeup.

In this lab, you will be implementing a server for a distributed hash table. While some of the code will be provided, you will be doing much of the implementation (and all of the testing). The DHT stores (*key,value*) pairs, where the keys and values are strings. Clients add pairs to the DHT using *put* operations and retrieve them using *get* operations. The DHT uses a circular organization, similar to the one described in Kurose and Ross. It also uses “shortcuts” to improve routing efficiency and caches pairs when it can.

The protocol uses UDP packets and all information is represented as text. An example of the UDP payload for a *get* request is shown below.

```
CSE473 DHTPv0.2015
type:get
key:dungeons
tag:12345
ttl:100
```

The first line is just an identifying string that is required in every DHT packet. The remaining lines all start with a keyword and a colon, usually followed by some additional text. Here, the *type* field specifies that this is a *get* request; the *key* field specifies the key to be looked up; the *tag* is a client-specified tag (must be an integer) that is returned in the response; it can be used by the client to match responses with requests; the *ttl* is set by the client to a positive integer and is decremented by every *DhtServer* and if <0, causes the packet to be discarded.

Possible responses to the above request include:

```
CSE473 DHTPv0.2015
type:success
key:dungeons
value:dragons
tag:12345
ttl:95
```

or

```
CSE473 DHTPv0.2015
type:no match
key:dungeons
tag:12345
ttl:95
```

Put requests are formatted similarly, but in this case the client typically specifies a *value* field (omitting the value field causes the pair with the specified key to be removed).

The packet type “*failure*” is used to indicate an error of some sort; in this case, the “*reason*” field provides an explanation of the failure. The “*join*” type is used by a server to join an existing DHT. The “*transfer*” type is used to transfer (*key,value*) pairs to a newly added server.

Other fields and their use are described briefly below

- | | |
|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>clientAdr</i> | is used to specify the IP address and port number of the client that sent a particular request; it is added to a request packet by the first server to receive the request, before forwarding the packet to another node in the DHT; an example of the format is <code>clientAdr:123.45.67.89:51349</code> . |
| <i>relayAdr</i> | is used to specify the IP address and port number of the first server to receive a request packet from the client; it is added to the packet by the first server before forwarding the packet. |
| <i>hashRange</i> | is a pair of integers separated by a colon, specifying a range of hash indices; it is included in the response to a “ <i>join</i> ” packet, to inform the new DHT server of the set of hash values it is responsible for. |
| <i>succInfo</i> | is the IP address and port number of a server, followed by its first hash index; this information is included in the response to a join packet to inform the new DHT server about its immediate successor; an example of the format is <code>succInfo:123.45.6.7:5678:987654321</code> . |
| <i>senderInfo</i> | is the IP address and port number of a DHT server, followed by its first hash index; this information is sent by a DHT to provide routing information that can be used by another. |

For the purposes of this lab, you may assume that packets are never lost and that clients and servers are all cooperative (that is, you need not protect against malicious behavior). You may also assume that servers never leave the DHT and that they never fail.

Here are some more details of the server's operation. When a server receives a *get* or *put* from a client, it first hashes the key to determine if it is the "responsible server" for this request. If it is, then it responds directly to the client. If it is not, it adds a *clientAdr* field to the packet containing the client's socket address. It also adds a *relayAdr* field containing its own socket address. Then, it forwards the packet around the DHT. When a server receives a *get* or *put* from another server, it behaves similarly. If the server determines that it is not the responsible server, it simply forwards the packet towards the responsible server. If it determines that it is the responsible server, it performs the operation and then converts the packet to a response packet, and sends it to the *relay server* using the *relayAdr* field in the packet. Before sending the response, the server also adds a *senderInfo* field to the packet containing its own IP address and port number and the first hash index in its range (this field is used to establish shortcut routes, as discussed below). Note that a server can recognize a packet coming from a client, since it will not include a *relayAdr* field.

When a server gets a response packet (type = "success", "failure" or "no match") from another server, it assumes that it is the relay server, and forwards the packet on to the client, using the address in the *clientAdr* field of the packet. Before doing so, it removes the *clientAdr*, *relayAdr* and *senderInfo* fields from the packet. It also uses the *senderInfo* field to add a shortcut route to its routing table. If the response is a "success" packet, it also stores the (*key,value*) pair in its local *cache*.

The server uses a routing table when deciding where to forward a packet. For this lab, the routing table will be a simple list containing tuples of the form (*nextHopAddress*, *firstHash*) where *nextHopAddress* is the IP address and port number of another server, and *firstHash* is the first hash index in the range for which that server is responsible. When forwarding a packet, we try to forward it to the server that is closest to the target of the operation. We do this by comparing the hash index of the packet's *key* to the *firstHash* values of the entries in the routing table, and selecting the server that comes closest to the target. Routes are added to the routing table opportunistically. Whenever a packet is received that contains a *senderInfo* field, this information is added to the routing table. However, if this would cause the number of routing table entries to exceed a specified limit, the new entry will replace one of the old ones. However, note that the entry for the *successor* of a node should never be replaced.

A server joins the DHT by sending a *join* packet to an existing server, then waiting for a response. The response will normally be a "success" packet, with a *hashRange* field and a *succInfo* field. This tells the new server what range of hash indices it is responsible for and who its successor is. The new server records this information and initializes its routing table to include an entry for its successor. When a server receives a *join* packet, it splits its hash range in half, giving the "top" half of the range to its new successor. After responding to a *join* request, the responding server also sends a series of *transfer* packets to the new server. Each of these contains a (*key,value*) that the new server is now responsible for. The original server also removes these pairs from its own local map. When a server receives a transfer packet, it updates its local map, but does not send a reply.

Some final details. If a server receives a *get* packet that it is not responsible for, it checks to see if it has a matching *key* in its local cache. If so, it responds to the request as though it were the responsible server. When a server receives a *put* that it is not responsible for, it checks for the *key* in its cache. If there is a pair with that *key* in the cache, it removes it.

The repository contains a partial implementation of *DhtServer* and another file containing a partial implementation of a *Packet* class. You will need to study the provided code first, then fill in the missing parts. Think carefully about the interactions among servers and make sure you understand how all the “moving parts” work together. You will also need to write a *DhtClient*. This program will take from 3 to 5 command line arguments. The first is the IP address of the interface that the client should bind to its own datagram socket. The second is the name of a configuration file containing the IP address and port number used by a *DhtServer* (each server writes such a file when it starts up). The third is an operation (“*get*” or “*put*”) and the remaining arguments specify the *key* and/or *value* for the operation. These may be omitted. Your client should not do any error checking. Leave that to the server. The client should enable the debug flag for the *Packet.send()* and *Packet.receive()* methods. This will allow you to see every packet that the client sends or receives.

The repository also includes a *README* file. Read it.

A word of advice. Start with a limited version of *DhtServer*. Specifically implement only what you need for a single node DHT and test the interactions with the client using this configuration (remember to include the *debug* argument when you run the server). Then, expand to a two node DHT. Make sure that the *join* works correctly. Then go to three nodes and start testing puts and gets. Don’t bother with short-cut routes or caching in your initial testing, as these features make the behavior of the DHT more complicated and harder to understand. Once you are sure that the basic stuff works correctly, go ahead and add short-cut routes. Then add caching at the very end.

The provided lab report template contains additional instructions and a number of questions for you to answer. Your lab report for this lab will end up being quite long. It may take you multiple sessions to generate the data you need for your report. When you have your code working and you are ready to generate results for Parts D, E and F, you may need to break them up into different sessions. Part D should be fairly short. Parts E and F will each take some time.

Start early and read all the material before your start.