###### *CSE 473 – Introduction to Computer Networks*

Lab 3 Report

##### *Katharine Thomas & Danny Munro*

***Part A (30 points).*** Place a copy of the source code of the functions in *DhtServer* to which you added any code or documentation; remember to include the documentation you added for the functions that required it. Highlight your changes by making them **bold**. Remember to also place a complete copy in the repository before you make your final commit. *Your* committed version should have no extraneous *print* statements.

/\*\* Server for simple distributed hash table that stores (key,value) strings.

\*

\* usage: DhtServer myIp numRoutes cfgFile [ cache ] [ debug ] [ predFile ]

\*

\* myIp is the IP address to use for this server's socket

\* numRoutes is the max number of nodes allowed in the DHT's routing table;

\* typically lg(numNodes)

\* cfgFile is the name of a file in which the server writes the IP

\* address and port number of its socket

\* cache is an optional argument; if present it is the literal string

\* "cache"; when cache is present, the caching feature of the

\* server is enabled; otherwise it is not

\* debug is an optional argument; if present it is the literal string

\* "debug"; when debug is present, a copy of every packet received

\* and sent is printed on stdout

\* predFile is an optional argument specifying the configuration file of

\* this node's predecessor in the DHT; this file is used to obtain

\* the IP address and port number of the precessor's socket,

\* allowing this node to join the DHT by contacting predecessor

\*

\* The DHT uses UDP packets containing ASCII text. Here's an example of the

\* UDP payload for a get request from a client.

\*

\* CSE473 DHTPv0.1

\* 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 :, 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 that is returned in the response; and

\* can be used by the client to match responses with requests; the ttl is

\* decremented by every DhtServer and if <0 causes the packet to be discarded.

\*

\* Possible responses to the above request include:

\*

\* CSE473 DHTPv0.1

\* type:success

\* key:dungeons

\* value:dragons

\* tag:12345

\* ttl:95

\*

\* or

\*

\* CSE473 DHTPv0.1

\* 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. In the same

\* way, the 'leave' type is used by the leaving server to circle around the

\* DHT asking other servers to delete it from their routing tables. The

\* 'transfer' type is used to transfer (key,value) pairs to a newly added

\* server. The 'update' type is used to update the predecessor, successor,

\* or hash range of another DHT server, usually when a join or leave even

\* happens.

\*

\* Other fields and their use are described briefly below

\* clientAdr 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 clientAdr:123.45.67.89:51349.

\* relayAdr 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.

\* hashRange is a pair of integers separated by a colon, specifying a range

\* of hash indices; it is included in the response to a 'join'

\* packet, to inform the new DHT server of the set of hash values

\* it is responsible for; it is also included in the update packet

\* to update the hash range a server is responsible for.

\* succInfo 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; it is also included in the update packet

\* to change the immediate successor of a DHT server; an example

\* of the format is succInfo:123.45.6.7:5678:987654321.

\* predInfo is also the IP address and port number of a server, followed

\* by its first hash index; this information is included in a join

\* packet to inform the successor DHT server of its new

\* predecessor; it is also included in update packets to update

\* the new predecessor of a server.

\* senderInfo 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 other servers.

\* It also used in leave packet to let other servers know the IP

\* address and port number information of the leaving server.

\*/

import java.io.\*;

import java.net.\*;

import java.util.\*;

import java.util.Map.Entry;

import java.lang.\*;

import sun.misc.Signal;

import sun.misc.SignalHandler;

public class DhtServer {

private static int numRoutes; // number of routes in routing table

private static boolean cacheOn; // enables caching when true

private static boolean debug; // enables debug messages when true

private static HashMap<String,String> map; // key/value pairs

private static HashMap<String,String> cache; // cached pairs

private static List<Pair<InetSocketAddress,Integer>> rteTbl;

private static DatagramSocket sock;

private static InetSocketAddress myAdr;

private static InetSocketAddress predecessor; // DHT predecessor

private static Pair<InetSocketAddress,Integer> myInfo;

private static Pair<InetSocketAddress,Integer> predInfo;

private static Pair<InetSocketAddress,Integer> succInfo; // successor

private static Pair<Integer,Integer> hashRange; // my DHT hash range

private static int sendTag; // tag for new outgoing packets

// flag for waiting leave message circle back

private static boolean stopFlag;

/\*\* Main method for DHT server.

\* Processes command line arguments, initializes data, joins DHT,

\* then starts processing requests from clients.

\*/

public static void main(String[] args) {

// process command-line arguments

if (args.length < 3) {

System.err.println("usage: DhtServer myIp numRoutes " +

"cfgFile [debug] [ predFile ] ");

System.exit(1);

}

numRoutes = Integer.parseInt(args[1]);

String cfgFile = args[2];

cacheOn = debug = false;

stopFlag = false;

String predFile = null;

for (int i = 3; i < args.length; i++) {

if (args[i].equals("cache")) cacheOn = true;

else if (args[i].equals("debug")) debug = true;

else predFile = args[i];

}

// open socket for receiving packets

// write ip and port to config file

// read predecessor's ip/port from predFile (if there is one)

InetAddress myIp = null; sock = null; predecessor = null;

try {

myIp = InetAddress.getByName(args[0]);

sock = new DatagramSocket(0,myIp);

BufferedWriter cfg =

new BufferedWriter(

new OutputStreamWriter(

new FileOutputStream(cfgFile),

"US-ASCII"));

cfg.write("" + myIp.getHostAddress() + " " +

sock.getLocalPort());

cfg.newLine();

cfg.close();

if (predFile != null) {

BufferedReader pred =

new BufferedReader(

new InputStreamReader(

new FileInputStream(predFile),

"US-ASCII"));

String s = pred.readLine();

String[] chunks = s.split(" ");

predecessor = new InetSocketAddress(

chunks[0],Integer.parseInt(chunks[1]));

}

} catch(Exception e) {

System.err.println("usage: DhtServer myIp numRoutes " +

"cfgFile [ cache ] [ debug ] " +

"[ predFile ] ");

System.exit(1);

}

myAdr = new InetSocketAddress(myIp,sock.getLocalPort());

// initialize data structures

map = new HashMap<String,String>();

cache = new HashMap<String,String>();

rteTbl = new LinkedList<Pair<InetSocketAddress,Integer>>();

// join the DHT (if not the first node)

hashRange = new Pair<Integer,Integer>(0,Integer.MAX\_VALUE);

myInfo = null;

succInfo = null;

predInfo = null;

if (predecessor != null) {

join(predecessor);

} else {

myInfo = new Pair<InetSocketAddress,Integer>(myAdr,0);

succInfo = new Pair<InetSocketAddress,Integer>(myAdr,0);

predInfo = new Pair<InetSocketAddress,Integer>(myAdr,0);

}

// start processing requests from clients

Packet p = new Packet();

Packet reply = new Packet();

InetSocketAddress sender = null;

sendTag = 1;

/\* this function will be called if there's a "TERM" or "INT"

\* captured by the signal handler. It simply execute the leave

\* function and leave the program.

\*/

SignalHandler handler = new SignalHandler() {

public void handle(Signal signal) {

leave();

System.exit(0);

}

};

//Signal.handle(new Signal("KILL"), handler); // capture kill -9 signal

Signal.handle(new Signal("TERM"), handler); // capture kill -15 signal

Signal.handle(new Signal("INT"), handler); // capture ctrl+c

while (true) {

try { sender = p.receive(sock,debug);

} catch(Exception e) {

System.err.println(

"received packet failure, couldn't not receive");

continue;

}

if (sender == null) {

System.err.println(

“received packet failure, sender == null");

continue;

}

if (!p.check()) {

reply.clear();

reply.type = "failure";

reply.reason = p.reason;

reply.tag = p.tag;

reply.ttl = p.ttl;

reply.send(sock,sender,debug);

continue;

}

handlePacket(p,sender);

}

}

/\*\* Hash a string, returning a 32 bit integer.

\* @param s is a string, typically the key from some get/put operation.

\* @return and integer hash value in the interval [0,2^31).

\*/

public static int hashit(String s) {

while (s.length() < 16) s += s;

byte[] sbytes = null;

try { sbytes = s.getBytes("US-ASCII");

} catch(Exception e) {

System.err.println("illegal key string");

System.exit(1);

}

int i = 0;

int h = 0x37ace45d;

while (i+1 < sbytes.length) {

int x = (sbytes[i] << 8) | sbytes[i+1];

h \*= x;

int top = h & 0xffff0000;

int bot = h & 0xffff;

h = top | (bot ^ ((top >> 16)&0xffff));

i += 2;

}

if (h < 0) h = -(h+1);

return h;

}

/\*\* Leave an existing DHT.

\*

\* Send a leave packet to it's successor and wait until stopFlag is

\* set to "true", which means leave packet has circled back.

\*

\* Send an update packet with the new hashRange and succInfo fields to

\* its predecessor, and sends an update packet with the predInfo

\* field to its successor.

\*

\* Transfers all keys and values to predecessor.

\* Clear all the existing cache, map and rteTbl information

\*/

public static void leave() {

**//leave packet to succ**

**Packet leavePacket = new Packet();**

**leavePacket.type = "leave";**

**leavePacket.tag = ++sendTag;**

**leavePacket.senderInfo = myInfo;**

**leavePacket.send(sock, succInfo.left, debug);**

**while (!stopFlag) {} //do nothing, just keep checking**

**//update packet to pred**

**Packet updatePacketPred = new Packet();**

**updatePacketPred.type = "update";**

**updatePacketPred.tag = ++sendTag;**

**updatePacketPred.succInfo = succInfo;**

**int left = predInfo.right;**

**int right = hashRange.right.intValue();**

**updatePacketPred.hashRange = new Pair<Integer, Integer>(left, right);**

**updatePacketPred.send(sock, predInfo.left, debug);**

**//update packet to succ**

**Packet updatePacketSucc = new Packet();**

**updatePacketSucc.type = "update";**

**updatePacketSucc.tag = ++sendTag;**

**updatePacketSucc.predInfo = predInfo;**

**updatePacketSucc.send(sock, succInfo.left, debug);**

**for (Map.Entry<String,String> entry : map.entrySet()) {**

**Packet transferPacket = new Packet();**

**transferPacket.type = "transfer";**

**transferPacket.tag = ++sendTag;**

**transferPacket.key = entry.getKey();**

**transferPacket.val = entry.getValue();**

**transferPacket.send(sock, predInfo.left, debug);**

**}**

**//clear map, cache, and rtetbl**

**map.clear();**

**cache.clear();**

**rteTbl.clear();**

**sock.close();**

**}**

/\*\* Handle a update packet from a prospective DHT node.

\* @param p is the received join packet

\* @param adr is the socket address of the host that

\*

\* The update message might contains infomation need update,

\* including predInfo, succInfo, and hashRange.

\* And add the new Predecessor/Successor into the routing table.

\* If succInfo is updated, succInfo should be removed from

\* the routing table and the new succInfo should be added

\* into the new routing table.

\*/

public static void handleUpdate(Packet p, InetSocketAddress adr) {

if (p.predInfo != null){

predInfo = p.predInfo;

}

if (p.succInfo != null){

succInfo = p.succInfo;

addRoute(succInfo);

}

if (p.hashRange != null){

hashRange = p.hashRange;

}

}

/\* Handle a leave packet from a leaving DHT node.

\* @param p is the received join packet

\* @param adr is the socket address of the host that sent the leave packet

\*

\* If the leave packet is sent by this server, set the stopFlag.

\* Otherwise first send the received leave packet to its successor,

\* and then remove the routing entry with the senderInfo of the packet.

\*/

public static void handleLeave(Packet p, InetSocketAddress adr) {

if (p.senderInfo.equals(myInfo)){

stopFlag = true;

return;

}

// send the leave message to successor

p.send(sock, succInfo.left, debug);

//remove the senderInfo from route table

removeRoute(p.senderInfo);

}

/\* Join an existing DHT.

\* @param predAdr is the socket address of a server in the DHT,

\*

\*

**\* The server sends a join packet and waits for a response from**

**\* the DHT. The response should include information on the node's**

**\* successor and predecessor as well as a range of hash values for**

**\* which it is responsible. \*/**

**public static void join(InetSocketAddress predAdr) {**

**//send packet**

**Packet joinPacket = new Packet();**

**joinPacket.type = "join"; joinPacket.tag = ++sendTag;**

**joinPacket.send(sock, predAdr, debug);**

**//correctly identify reply packet**

**Packet replyPacket = new Packet();**

**while (true) {**

**InetSocketAddress replyAdr = replyPacket.receive(sock, debug);**

**if (replyAdr.equals(predAdr) &&**

**(replyPacket.tag == joinPacket.tag)){**

**break;**

**}**

**}**

**//check to make sure it's correctly formed**

**if (!replyPacket.check()) {**

**System.err.println("Error, server could not join DHT network: "**

**+ replyPacket.reason);**

**System.exit(1);**

**}**

**if (!replyPacket.type.equals("success") ||**

**replyPacket.hashRange == null || replyPacket.predInfo == null ||**

**replyPacket.succInfo == null) {**

**System.err.println("Error: Response packet was malformed. " +**

**"Type must be 'success' and hashRange, succInfo, and” + “predInfo must not be null. Packet received was: ");**

**System.err.println(replyPacket.toString());**

**System.exit(1);**

}

**//update info**

**hashRange = replyPacket.hashRange;**

**predInfo = replyPacket.predInfo;**

**succInfo = replyPacket.succInfo;**

**myInfo = new Pair<InetSocketAddress, Integer>(myAdr, hashRange.left);**

**addRoute(succInfo);**

**}**

/\*\* Handle a join packet from a prospective DHT node.

\* @param p is the received join packet

\* @param succAdr is the socket address of the host that

\* sent the join packet (the new successor)

\*

**\* Let host A be the node that wants to join**

**\* Let host B be the node that is processing the request**

**\* handleJoin accepts a join request from host A by sending**

**\* a reply packet containing**

**\* (a) type:success**

**\* (b) a hashRange field that contains the top half host B's**

**\* responsible hashes**

**\* (c) information on Host A's successor node**

**\* (d) information on Host A's predecessor node**

**\***

**\* It then sends Host A transfer packets containing the key/value**

**\* pairs it is now responsible for.**

**\***

**\* It then notifies its current successor that its new precessor**

**\* is Host A.**

**\*/**

**public static void handleJoin(Packet p, InetSocketAddress succAdr) {**

**//hash info: divide hash range value in half and send top to new node**

**int left = hashRange.left.intValue();**

**int right = hashRange.right.intValue();**

**int newRight = left + (right-left)/2;**

**hashRange.right = new Integer(newRight);**

**p.hashRange = new Pair<Integer, Integer>(newRight+1, right);**

**//pred info (current node's pred stays the same)**

**p.predInfo = new Pair<InetSocketAddress, Integer>(myAdr, left);**

**//succ info**

**p.succInfo = succInfo;**

**p.type = "success";**

**p.send(sock, succAdr, debug);**

**//transfer hashes**

**Packet transferPacket = new Packet();**

**transferPacket.type = "transfer";**

**Iterator<Entry<String, String>> it = map.entrySet().iterator();**

**while (it.hasNext()) {**

**Map.Entry<String, String> entry =**

**(Map.Entry<String, String>) it.next();**

**String key = (String) entry.getKey();**

**if (hashit(key) > newRight) {**

**transferPacket.tag = ++sendTag;**

**transferPacket.key = key;**

**transferPacket.val = (String) entry.getValue();**

**transferPacket.send(sock, succAdr, debug);**

**it.remove();**

**}**

**}**

**Pair<InetSocketAddress, Integer> newPacketInfo =**

**new Pair<InetSocketAddress, Integer>(succAdr, newRight+1);**

**//notify successor of new predecessor**

**Packet updatePacket = new Packet();**

**updatePacket.type = "update";**

**updatePacket.tag = ++sendTag;**

**updatePacket.predInfo = newPacketInfo;**

**updatePacket.send(sock, succInfo.left, debug);**

**//update current successor info**

**succInfo = newPacketInfo;**

**addRoute(succInfo);**

**}**

/\*\* Handle a get packet.

\* @param p is a get packet

\* @param senderAdr is the the socket address of the sender

\*

**\* If the key hashes into the values covered by the node, the node**

**\* responds to the request and sends a response to the relay node**

**\* (or the client, if it is the node to first receive the request)**

**\***

**\* If the key doesn't hash into the values the node is resposible for,**

**\* but caching is turned on, the node checks to see if the answer is**

**\* in its cache. If so, it handles the request like it were the**

**\* responsible node.**

**\***

**\* Otherwise, it forwards the request to the proper node.**

**\*/**

public static void handleGet(Packet p, InetSocketAddress senderAdr) {

InetSocketAddress replyAdr;

int hash = hashit(p.key);

int left = hashRange.left.intValue();

int right = hashRange.right.intValue();

if (left <= hash && hash <= right) {

// respond to request using map

if (p.relayAdr != null) {

replyAdr = p.relayAdr;

p.senderInfo = new

Pair<InetSocketAddress,Integer>(myAdr,left);

} else {

replyAdr = senderAdr;

}

if (map.containsKey(p.key)) {

p.type = "success"; p.val = map.get(p.key);

} else {

p.type = "no match";

}

p.send(sock,replyAdr,debug);

}

**else if (cacheOn && cache.containsKey(p.key)) {**

**p.type = "success"; p.val = cache.get(p.key);**

**if (p.relayAdr != null) {**

**replyAdr = p.relayAdr;**

**p.senderInfo = new**

**Pair<InetSocketAddress, Integer>(myAdr, left);**

**}**

**else {**

**replyAdr = senderAdr;**

**}**

**p.send(sock,replyAdr,debug);**

**}**

**else {**

// forward around DHT

if (p.relayAdr == null) {

p.relayAdr = myAdr; p.clientAdr = senderAdr;

}

forward(p,hash);

}

}

/\*\* Handle a put packet.

\* @param p is a put packet

\* @param senderAdr is the the socket address of the sender

\*

**\* If the key hashes to the range of responsible values for this node,**

**\* it puts the key/value pair into its map, adds its own information to**

**\* senderInfo and sends the response packet to either the relayAdr or**

**\* the client.**

**\***

**\* If it is not in the range the node is responsible for, it checks its**

**\* cache for the key. If the key exists it deletes the key/value pair. It**

**\* then forwards the packet to the appropriate node.**

\*/

**public static void handlePut(Packet p, InetSocketAddress senderAdr) {**

**int hash = hashit(p.key);**

**int left = hashRange.left.intValue();**

**int right = hashRange.right.intValue();**

**InetSocketAddress replyAdr = null;**

**if (left <= hash && hash <= right) {**

**// respond to request using map**

**if (p.relayAdr != null) {**

**replyAdr = p.relayAdr;**

**p.senderInfo = new**

**Pair<InetSocketAddress,Integer>(myAdr,left);**

**} else {**

**replyAdr = senderAdr;**

**}**

**if (p.val != null) {**

**map.put(p.key,p.val);**

**} else {**

**map.remove(p.key);**

**}**

**p.type = "success";**

**p.send(sock,replyAdr,debug);**

**}**

**else { //if not in hash**

**if (cacheOn && cache.containsKey(p.key)) {**

**cache.remove(p.key);**

**}**

**// forward around DHT**

**if (p.relayAdr == null) {**

**p.relayAdr = myAdr; p.clientAdr = senderAdr;**

**}**

**forward(p,hash);**

**}**

**}**

/\*\* Handle a transfer packet.

\* @param p is a transfer packet

\* @param senderAdr is the the address (ip:port) of the sender

\*

**\* If a key hashes to a range that it used to be responsible for, but**

**\* now this node is, it transfers the packet to this node, which adds**

**\* it to its map.**

**\*/**

**public static void handleXfer(Packet p, InetSocketAddress senderAdr) {**

**int hash = hashit(p.key);**

**int left = hashRange.left.intValue();**

**int right = hashRange.right.intValue();**

**if (left <= hash && hash <= right) {**

**map.put(p.key,p.val);**

**}**

**}**

/\*\* Handle a reply packet.

\* @param p is a reply packet, more specifically, a packet of type

\* "success", "failure" or "no match"

\* @param senderAdr is the the address (ip:port) of the sender

\*

**\* Forwards the packet on to the client after removing extraneous fields**

**\* Key/value pair is added to the local cache when type is "success".**

**\*/**

**public static void handleReply(Packet p, InetSocketAddress senderAdr) {**

**//if caching is on, add pair to cache**

**if (cacheOn && p.type.equals("success") && p.val != null**

**&& p.key != null) {**

**cache.put(p.key,p.val);**

**}**

**//get rid of extraneous fields and send to client**

**InetSocketAddress clientAdr = p.clientAdr;**

**p.clientAdr = null; p.relayAdr = null; p.senderInfo = null;**

**p.send(sock, clientAdr, debug);**

**}**

/\*\* Handle packets received from clients or other servers

\* @param p is a packet

\* @param senderAdr is the address (ip:port) of the sender

\*/

public static void handlePacket(Packet p, InetSocketAddress senderAdr) {

if (p.senderInfo != null & !p.type.equals("leave"))

addRoute(p.senderInfo);

if (p.type.equals("get")) {

handleGet(p,senderAdr);

} else if (p.type.equals("put")) {

handlePut(p, senderAdr);

} else if (p.type.equals("transfer")) {

handleXfer(p, senderAdr);

} else if (p.type.equals("success") ||

p.type.equals("no match") ||

p.type.equals("failure")) {

handleReply(p, senderAdr);

} else if (p.type.equals("join")) {

handleJoin(p, senderAdr);

} else if (p.type.equals("update")){

handleUpdate(p, senderAdr);

} else if (p.type.equals("leave")){

handleLeave(p, senderAdr);

}

}

/\*\* Add an entry to the route tabe.

\* @param newRoute is a pair (addr,hash) where addr is the socket

\* address for some server and hash is the first hash in that

\* server's range

\*

**\* If the number of entries in the table exceeds the max**

**\* number allowed, the first entry that does not refer to**

**\* the successor of this server, is removed.**

**\* If debug is true and the set of stored routes does change,**

**\* print the string "rteTbl=" + rteTbl. (IMPORTANT)**

**\*/**

**public static void addRoute(Pair<InetSocketAddress,Integer> newRoute){**

**//check if already in route**

**for (Pair<InetSocketAddress, Integer> entry : rteTbl) {**

**if (entry.equals(newRoute)){**

**return;**

**}**

**}**

**//add to table**

**rteTbl.add(newRoute);**

**if (rteTbl.size() <= numRoutes) {**

**if (debug) System.out.println("rteTble=" + rteTbl);**

**return;**

**}**

**//if too many routes, delete one**

**for (Pair<InetSocketAddress, Integer> entry : rteTbl) {**

**if (!entry.equals(succInfo)) {**

**removeRoute(entry);**

**break;**

**}**

**}**

**return;**

**}**

/\*\* Remove an entry from the route tabe.

\* @param rmRoute is the route information for some server

\* need to be removed from route table

\*

**\* If the route information exists in current entries, remove it.**

**\* Otherwise, do nothing.**

**\* If debug is true and the set of stored routes does change,**

**\* print the string "rteTbl=" + rteTbl. (IMPORTANT)**

**\*/**

**public static void removeRoute(Pair<InetSocketAddress,Integer> rmRoute){**

**//check if in map**

**Pair<InetSocketAddress,Integer> node = null;**

**for (Pair<InetSocketAddress, Integer> entry : rteTbl) {**

**if (entry.equals(rmRoute)){**

**node = entry;**

**break;**

**}**

**}**

**//if node in map, remove it**

**if (node != null) {**

**rteTbl.remove(node);**

**if (debug) {**

**System.out.println("rteTbl=" + rteTbl);**

**}**

**}**

**return;**

**}**

/\*\* Forward a packet using the local routing table.

\* @param p is a packet to be forwarded

\* @param hash is the hash of the packet's key field

\*

**\* This method selects a server from its route table that is**

**\* "closest" to the target of this packet (based on hash).**

**\* If firstHash is the first hash in a server's range, then**

**\* we seek to minimize the difference hash-firstHash, where**

**\* the difference is interpreted modulo the range of hash values.**

**\* IMPORTANT POINT - handle "wrap-around" correctly.**

**\* Once a server is selected, p is sent to that server.**

**\*/**

**public static void forward(Packet p, int hash) {**

**Pair<InetSocketAddress,Integer> closestNode = null;**

**int smallestDiff = Integer.MAX\_VALUE;**

**for (Pair<InetSocketAddress,Integer> node : rteTbl) {**

**int firstHash = node.right;**

**int hashDiff = hash - firstHash;**

**//wrap around**

**if (hashDiff < 0) {**

**hashDiff += Integer.MAX\_VALUE;**

**++hashDiff;**

**}**

**if (hashDiff < smallestDiff || closestNode == null) {**

**closestNode = node;**

**smallestDiff = hashDiff;**

**}**

**}**

**p.send(sock, closestNode.left, debug);**

**}**

}

***Part B (10 points).*** Place a copy of the source code of the functions in *Packet* where you added code and comments; highlight your changes by making them **bold*.*** Include a complete copy in the repository before you make your final commit. *Your* committed version should have no extraneous *print* statements.

import java.io.\*;

import java.net.\*;

import java.util.\*;

/\*\* Class for working with DHT packets. \*/

public class Packet {

// packet fields - note: all are public

public String type; // packet type

public int ttl; // time-to-live

public String key; // DHT key string

public String val; // DHT value string

public String reason; // reason for a failure

public InetSocketAddress clientAdr; // address of original client

public InetSocketAddress relayAdr; // address of first DHT server

public int tag; // tag used to identify packet

public Pair<Integer,Integer> hashRange; // range of hash values

public Pair<InetSocketAddress,Integer> senderInfo;// address, first hash

public Pair<InetSocketAddress,Integer> succInfo; // address, first hash

public Pair<InetSocketAddress,Integer> predInfo; // address, first hash

/\*\* Constructor, initializes fields to default values. \*/

public Packet() { clear(); }

/\*\* Initialize all packet fields.

\* Initializes all fields with a standard initial value

\* or makes them undefined.

\*/

public void clear() {

type = null; ttl = 100; key = null; val = null;

reason = null; clientAdr = null; relayAdr = null;

tag = -1; hashRange = null;

senderInfo = null; succInfo = null; predInfo = null;

}

/\*\* Pack attributes defining packet fields into buffer.

\* Fails if the packet type is undefined or if the resulting

\* buffer exceeds the allowed length of 1400 bytes.

\* @return null on failure, otherwise a byte array

\* containing the packet payload.

\*/

public byte[] pack() {

if (type == null) return null;

byte[] buf;

try { buf = toString().getBytes("US-ASCII");

} catch(Exception e) { return null; }

if (buf.length > 1400) return null;

return buf;

}

/\*\* Unpack attributes defining packet fields from buffer.

\* @param buf is a byte array containing the DHT packet

\* (or if you like, the payload of a UDP packet).

\* @param bufLen is the number of valid bytes in buf

\*/

public boolean unpack(byte[] buf, int bufLen) {

// convert buf to a string

String s;

try { s = new String(buf,0,bufLen,"US-ASCII");

} catch(Exception e) { return false; }

// divide into lines and check the first line

String[] lines = s.split("\n");

if (!lines[0].equals("CSE473 DHTPv0.1")) return false;

//process remaining lines

for (int i = 1; i < lines.length; i++) {

String[] chunks = lines[i].split(":",2);

if (chunks.length != 2) return false;

// process the line

String left = chunks[0];

String right = chunks[1];

if (left.equals("type")) {

type = right;

} else if (left.equals("ttl")) {

ttl = Integer.parseInt(right);

**} else if (left.equals("key")) {**

**key = right;**

**} else if (left.equals("val")) {**

**val = right;**

**} else if (left.equals("reason")) {**

**reason = right;**

} else if (left.equals("clientAdr")) {

chunks = right.split(":");

if (chunks.length != 2) return false;

clientAdr = new InetSocketAddress(chunks[0],

Integer.parseInt(chunks[1]));

**} else if (left.equals("relayAdr")) {**

**chunks = right.split(":");**

**if (chunks.length != 2) return false;**

**relayAdr = new InetSocketAddress(chunks[0],**

**Integer.parseInt(chunks[1]));**

**} else if (left.equals("tag")){**

**tag = Integer.parseInt(right);**

**} else if (left.equals("hashRange")) {**

**chunks = right.split(":");**

**hashRange = new Pair<Integer,Integer>(**

**Integer.parseInt(chunks[0]),**

**Integer.parseInt(chunks[1]));**

**} else if (left.equals("predInfo")) {**

**chunks = right.split(":");**

**if (chunks.length != 3) return false;**

**String ip = chunks[0];**

**int port = Integer.parseInt(chunks[1]);**

**int hash = Integer.parseInt(chunks[2]);**

**predInfo = new**

**Pair<InetSocketAddress,Integer>(**

**new InetSocketAddress(ip,port),hash);**

} else if (left.equals("succInfo")) {

chunks = right.split(":");

if (chunks.length != 3) return false;

String ip = chunks[0];

int port = Integer.parseInt(chunks[1]);

int hash = Integer.parseInt(chunks[2]);

succInfo = new

Pair<InetSocketAddress,Integer>(

new InetSocketAddress(ip,port),hash);

**} else if (left.equals("senderInfo")) {**

**chunks = right.split(":");**

**if (chunks.length != 3) return false;**

**String ip = chunks[0];**

**int port = Integer.parseInt(chunks[1]);**

**int hash = Integer.parseInt(chunks[2]);**

**senderInfo = new**

**Pair<InetSocketAddress,Integer>(**

**new InetSocketAddress(ip,port),hash);**

**}**

else {

// ignore lines that don't match defined field

}

}

return true;

}

/\*\* Basic validity checking for received packets.

\* @return true on success, false on failure;

\* on failure, place an explanatory String in the reason field

\* of the packet

\*/

public boolean check() {

reason = null;

if (type == null) {

reason = "every packet must include a type";

return false;

} else if ((type.equals("get") || type.equals("put")) &&

(key == null || tag == -1)) {

reason = "gets and puts require key and tag";

return false;

}

return true;

}

/\*\* Create String representation of packet.

\* The resulting String is produced using the defined

\* attributes and is formatted with one field per line,

\* allowing it to be used as the actual buffer contents.

\*/

public String toString() {

StringBuffer s = new StringBuffer("CSE473 DHTPv0.1\n");

if (type != null) {

s.append("type:"); s.append(type); s.append("\n");

}

if (key != null) {

s.append("key:"); s.append(key); s.append("\n");

}

**if (val != null) {**

**s.append("val:"); s.append(val); s.append("\n");**

**}**

**if (reason != null) {**

**s.append("reason:"); s.append(reason); s.append("\n");**

**}**

**if (clientAdr != null) {**

**s.append("clientAdr:");**

**s.append(clientAdr.getAddress().getHostAddress());**

**s.append(":"); s.append(clientAdr.getPort());**

**s.append("\n");**

**}**

if (relayAdr != null) {

s.append("relayAdr:");

s.append(relayAdr.getAddress().getHostAddress());

s.append(":"); s.append(relayAdr.getPort());

s.append("\n");

}

**if (tag != -1) {**

**s.append("tag:"); s.append(tag); s.append("\n");**

**}**

if (hashRange != null) {

s.append("hashRange:"); s.append(hashRange.left);

s.append(":"); s.append(hashRange.right);

s.append("\n");

}

if (senderInfo != null) {

s.append("senderInfo:");

s.append(senderInfo.left.getAddress().getHostAddress());

s.append(":"); s.append(senderInfo.left.getPort());

s.append(":"); s.append(senderInfo.right);

s.append("\n");

}

**if (succInfo != null) {**

**s.append("succInfo:");**

**s.append(succInfo.left.getAddress().getHostAddress());**

**s.append(":"); s.append(succInfo.left.getPort());**

**s.append(":"); s.append(succInfo.right);**

**s.append("\n");**

**}**

**if (predInfo != null) {**

**s.append("predInfo:");**

**s.append(predInfo.left.getAddress().getHostAddress());**

**s.append(":"); s.append(predInfo.left.getPort());**

**s.append(":"); s.append(predInfo.right);**

**s.append("\n");**

**}**

if (ttl != -1) {

s.append("ttl:"); s.append(ttl); s.append("\n");

}

return s.toString();

}

/\*\* Send the packet to a specified destination.

\* Packs the various packet fields into a buffer

\* before sending. Does no validity checking.

\* @param sock is the socket on which the packet is sent

\* @param dest is the socket address of the destination

\* debug is a flag; if true, the packet is printed before it is sent

\* @return true on success, false on failure

\*/

public boolean send(DatagramSocket sock, InetSocketAddress dest,

boolean debug) {

if (debug) {

System.out.println("" + sock.getLocalSocketAddress() +

" sending packet to " + dest + "\n" +

toString());

System.out.flush();

}

byte[] buf = pack();

if (buf == null) return false;

DatagramPacket pkt = new DatagramPacket(buf, buf.length);

pkt.setSocketAddress(dest);

try { sock.send(pkt); } catch(Exception e) { return false; }

return true;

}

/\*\* Get the next packet on the socket.

\*

\* Receives the next datagram from the socket and

\* unpacks it.

\* @param sock is the socket on which the packet is received

\* @param debug is a flag; if it is true, the received

\* packet is printed

\* @return the sender's socket address on success and null on failure

\*/

public InetSocketAddress receive(DatagramSocket sock, boolean debug) {

clear();

byte[] buf = new byte[2000];

DatagramPacket pkt = new DatagramPacket(buf, buf.length);

try {

sock.receive(pkt);

} catch(Exception e) {

System.out.println("receive exception: " + e);

return null;

}

if (!unpack(buf,pkt.getLength())) {

System.out.println("error while unpacking packet");

return null;

}

ttl--;

if (debug) {

System.out.println(sock.getLocalSocketAddress() +

" received packet from " +

pkt.getSocketAddress() + "\n" + toString());

System.out.flush();

}

if (ttl < 0) return null;

return (InetSocketAddress) pkt.getSocketAddress();

}

}

***Part C (10 points).*** Place a copy of your source code for *SimpleDhtClient* here.

import java.io.BufferedReader;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.UnsupportedEncodingException;

import java.net.DatagramSocket;

import java.net.InetAddress;

import java.net.InetSocketAddress;

import java.net.UnknownHostException;

/\*\*

\* Client sends requests to DHT servers in the form of inserting, updating

\* or deleting (via a "put" request) or retrieving (via a "get" request) key-

\* value pairs on the server's hashmap.

\*

\* usage: DhtClient myIp cfgFile operation [ key ] [ value ]

\*

\* Takes 3-5 arguments:

\*

\* myIp: the IP address associated with this client

\*

\* cfgFile: a file containing the IP address and port number of the server to

\* which this client plans to make a request

\*

\* operation: either a "put" if inserting/updating/deleting information, or a

\* "get" if retrieving information.

\*

\* key: the key of the map entry

\*

\* value: the value of the map entry

\*/

public class DhtClient{

private static int sendTag = 12344;

//Indices for the run arguments

public static final int IP\_INDEX = 0;

public static final int CONFIG\_INDEX = 1;

public static final int OPERATION\_INDEX = 2;

public static final int KEY\_INDEX = 3;

public static final int VALUE\_INDEX = 4;

//Valid numbers of run arguments

public static final int MIN\_ARGS = 3;

public static final int MIN\_ARGS\_PLUS\_KEY = 4;

public static final int MAX\_ARGS = 5;

//The indices of the information read from the cfg file,

//once delimited into a string array

public static final int CONFIG\_IP = 0;

public static final int CONFIG\_PORT = 1;

public static final int EPHEMERAL\_PORT = 0;

public static final String DEFAULT\_CFG\_NAME = "cfg";

public static final int EXIT = 1;

public static final int DEFAULT\_TTL = 100;

public static final boolean DEBUG = true;

public static void main(String[] args){

// process command-line arguments

if (args.length < MIN\_ARGS) {

System.err.println("usage: DhtClient myIp cfgFile " +

"operation [key] [ value ] ");

System.exit(EXIT);

}

String op = args[OPERATION\_INDEX];

String key = null;

String val = null;

if (args.length > KEY\_INDEX) {

key = args[KEY\_INDEX];

}

if (args.length > VALUE\_INDEX) {

val = args[VALUE\_INDEX];

}

// open socket for sending/receiving packets

// read ip and port from config file

InetSocketAddress destAdr = null;

DatagramSocket sock = null;

BufferedReader inBuf;

try {

InetAddress myIp = InetAddress.getByName(args[IP\_INDEX]);

sock = new DatagramSocket(0, myIp);

inBuf = new BufferedReader(new InputStreamReader(

new FileInputStream(args[CONFIG\_INDEX]),"US-ASCII"));

String[] info = inBuf.readLine().split(" ");

InetAddress serverAdr = InetAddress.getByName(info[CONFIG\_IP]);

int serverPort = Integer.parseInt(info[CONFIG\_PORT]);

destAdr = new InetSocketAddress(serverAdr, serverPort);

} catch (NumberFormatException e) {

System.err.println("Config file contains invalid port");

e.printStackTrace();

} catch (UnsupportedEncodingException | FileNotFoundException e) {

System.err.println("Config file not found");

e.printStackTrace();

} catch (UnknownHostException e){

System.err.println("Invalid or unknown host");

e.printStackTrace();

} catch (IOException e) {

System.err.println("Config file is misformatted");

e.printStackTrace();

}

//create packet with request to send to server

//Debug option is always on for sending and receiving

Packet outPkt = new Packet();

outPkt.type = op;

outPkt.key = key;

outPkt.val = val;

outPkt.tag = ++sendTag;

outPkt.send(sock, destAdr, DEBUG);

//create packet to receive response from server

Packet inPkt = new Packet();

inPkt.receive(sock,DEBUG);

}

}

***Part D (10 points).*** Use the provided *script0* to test your client and server on a single computer. Of course, you will first need to compile your java code. If you prefer, you can do this using Eclipse, but you may find it simpler to just type

javac \*.java

in the lab3 directory where your java files are stored. We are using a signal handling API so servers can announce they are leaving before they exit. This will incur some compilation warnings, but you do not need to worry about the ones mentioning “*Signal”* or *“SignalHandler”*. When you test using *script0*, note that this script uses just a single server, so it does not test many of the features of your DHT, but it will allow you to check a significant fraction of the code. You may do this testing on any Unix (including MacOS) or Linux computer (shell.cec.wustl.edu or onl.wustl.edu). Go to the *test0* directory and read *script0* to make sure you understand what it does, then type

./script0 > out

to run it. Check the output file carefully. When you are satisfied that things are working correctly, paste a copy of the output below. **Commit the output file and the log file in your *test0* directory to your SVN repository.**

put foo bar

/127.0.0.1:62714 sending packet to /127.0.0.1:62713

CSE473 DHTPv0.1

type:put

key:foo

val:bar

tag:12345

ttl:100

/127.0.0.1:62714 received packet from /127.0.0.1:62713

CSE473 DHTPv0.1

type:success

key:foo

val:bar

tag:12345

ttl:98

put who hah

/127.0.0.1:62715 sending packet to /127.0.0.1:62713

CSE473 DHTPv0.1

type:put

key:who

val:hah

tag:12345

ttl:100

/127.0.0.1:62715 received packet from /127.0.0.1:62713

CSE473 DHTPv0.1

type:success

key:who

val:hah

tag:12345

ttl:98

get foo

/127.0.0.1:62716 sending packet to /127.0.0.1:62713

CSE473 DHTPv0.1

type:get

key:foo

tag:12345

ttl:100

/127.0.0.1:62716 received packet from /127.0.0.1:62713

CSE473 DHTPv0.1

type:success

key:foo

val:bar

tag:12345

ttl:98

get who

/127.0.0.1:62717 sending packet to /127.0.0.1:62713

CSE473 DHTPv0.1

type:get

key:who

tag:12345

ttl:100

/127.0.0.1:62717 received packet from /127.0.0.1:62713

CSE473 DHTPv0.1

type:success

key:who

val:hah

tag:12345

ttl:98

get goodbye

/127.0.0.1:62718 sending packet to /127.0.0.1:62713

CSE473 DHTPv0.1

type:get

key:goodbye

tag:12345

ttl:100

/127.0.0.1:62718 received packet from /127.0.0.1:62713

CSE473 DHTPv0.1

type:no match

key:goodbye

tag:12345

ttl:98

get

/127.0.0.1:62719 sending packet to /127.0.0.1:62713

CSE473 DHTPv0.1

type:get

tag:12345

ttl:100

/127.0.0.1:62719 received packet from /127.0.0.1:62713

CSE473 DHTPv0.1

type:failure

reason:gets and puts require key and tag

tag:12345

ttl:98

get bar

/127.0.0.1:62720 sending packet to /127.0.0.1:62713

CSE473 DHTPv0.1

type:get

key:bar

tag:12345

ttl:100

/127.0.0.1:62720 received packet from /127.0.0.1:62713

CSE473 DHTPv0.1

type:no match

key:bar

tag:12345

ttl:98

put foo toast is tasty

/127.0.0.1:62721 sending packet to /127.0.0.1:62713

CSE473 DHTPv0.1

type:put

key:foo

val:toast is tasty

tag:12345

ttl:100

/127.0.0.1:62721 received packet from /127.0.0.1:62713

CSE473 DHTPv0.1

type:success

key:foo

val:toast is tasty

tag:12345

ttl:98

get foo

/127.0.0.1:62722 sending packet to /127.0.0.1:62713

CSE473 DHTPv0.1

type:get

key:foo

tag:12345

ttl:100

/127.0.0.1:62722 received packet from /127.0.0.1:62713

CSE473 DHTPv0.1

type:success

key:foo

val:toast is tasty

tag:12345

ttl:98

***Part E (20 points).*** In this part, you are to use the provided *script1* (in the *test1* directory) to test your DHT on a single computer. This script uses four servers, so it will exercise the routing features of your DHT. In the questions that follow, we will refer to the servers by number. The first server that is started is number 0. Its successor in the DHT (after all servers have been started) is number 1. The next is number 2, and so forth. Read the *script1* file and make sure you understand what it does. Notice that each server produces a log file labeled with its number. Now, type

./script1 1 > out1

to run it. Note that this version limits the servers to a single route, so there are no shortcut routes at this point. When you are satisfied that your results are correct, paste the initial and last portion of the *out1* file below. Specifically, include everything up through the first “get who” sequence (including the reply for “get who”) and last four operations. Commit the output and log files to your repository.

put foo bar

/127.0.0.1:63576 sending packet to /127.0.0.1:55649

CSE473 DHTPv0.1

type:put

key:foo

val:bar

tag:12345

ttl:100

/127.0.0.1:63576 received packet from /127.0.0.1:55649

CSE473 DHTPv0.1

type:success

key:foo

val:bar

tag:12345

ttl:98

put who hah

/127.0.0.1:59233 sending packet to /127.0.0.1:64131

CSE473 DHTPv0.1

type:put

key:who

val:hah

tag:12345

ttl:100

/127.0.0.1:59233 received packet from /127.0.0.1:64131

CSE473 DHTPv0.1

type:success

key:who

val:hah

tag:12345

ttl:94

put junk mail

/127.0.0.1:56452 sending packet to /127.0.0.1:49751

CSE473 DHTPv0.1

type:put

key:junk

val:mail

tag:12345

ttl:100

/127.0.0.1:56452 received packet from /127.0.0.1:49751

CSE473 DHTPv0.1

type:success

key:junk

val:mail

tag:12345

ttl:95

put blue moose

/127.0.0.1:63976 sending packet to /127.0.0.1:59792

CSE473 DHTPv0.1

type:put

key:blue

val:moose

tag:12345

ttl:100

/127.0.0.1:63976 received packet from /127.0.0.1:59792

CSE473 DHTPv0.1

type:success

key:blue

val:moose

tag:12345

ttl:98

get foo

/127.0.0.1:58070 sending packet to /127.0.0.1:49751

CSE473 DHTPv0.1

type:get

key:foo

tag:12345

ttl:100

/127.0.0.1:58070 received packet from /127.0.0.1:49751

CSE473 DHTPv0.1

type:success

key:foo

val:bar

tag:12345

ttl:96

get who

/127.0.0.1:57983 sending packet to /127.0.0.1:59792

CSE473 DHTPv0.1

type:get

key:who

tag:12345

ttl:100

/127.0.0.1:57983 received packet from /127.0.0.1:59792

CSE473 DHTPv0.1

type:success

key:who

val:hah

tag:12345

ttl:95

get blue

/127.0.0.1:51601 sending packet to /127.0.0.1:55649

CSE473 DHTPv0.1

type:get

key:blue

tag:12345

ttl:100

/127.0.0.1:51601 received packet from /127.0.0.1:55649

CSE473 DHTPv0.1

type:success

key:blue

val:moose

tag:12345

ttl:95

get blue

/127.0.0.1:62984 sending packet to /127.0.0.1:55649

CSE473 DHTPv0.1

type:get

key:blue

tag:12345

ttl:100

/127.0.0.1:62984 received packet from /127.0.0.1:55649

CSE473 DHTPv0.1

type:success

key:blue

val:moose

tag:12345

ttl:96

get foo

/127.0.0.1:51957 sending packet to /127.0.0.1:49751

CSE473 DHTPv0.1

type:get

key:foo

tag:12345

ttl:100

/127.0.0.1:51957 received packet from /127.0.0.1:49751

CSE473 DHTPv0.1

type:success

key:foo

val:toast is tasty

tag:12345

ttl:96

get junk

/127.0.0.1:63550 sending packet to /127.0.0.1:64131

CSE473 DHTPv0.1

type:get

key:junk

tag:12345

ttl:100

/127.0.0.1:63550 received packet from /127.0.0.1:64131

CSE473 DHTPv0.1

type:success

key:junk

val:mail

tag:12345

ttl:98

By examining the *out1* file, determine the port number used by the server that holds the (*key*,*value*) pair (*blue*, *moose*). What’s the ttl of the packet returned to client?

*Port number: 59792*

*ttl = 98*

Note the last eight *get* operations in the *out1* file before server 2 exits the DHT. Based on the *ttls* of the reply packets, determine each server’s successor. For this question, identify the servers by their port numbers, and also provide the *ttls.*

*49751 = cgf3 ttl: 94 successor = 55649*

*55649 = cfg0 ttl: 95 successor = 64131*

*64131 = cgf1 ttl: 96 successor = 59792*

*59792 = cfg2 ttl: 98 successor = 49751*

For the last two “get blue” operations, they are requesting the same server. Why do they get different *ttls*?

*The first time cfg0 requests “get blue”, server 2 holds the answer, so it needs to hop through cfg1 to get there, decrementing ttl from cfg0🡪cfg1 and from cfg1🡪cfg2.*

*The second time, cfg2 has left the DHT and has transferred all of its key/value pairs to cfg1, so cfg0 only needs to hop to cfg1, decrementing only from cfg0🡪cfg1.*

*Therefore, it makes sense that for the second “get blue” request, the ttl is 1 greater than it was the first time – there was one less hop to make.*

Paste the initial portion of the *log1\_2* file below (everything up through the first “*get blue*” operation and response).

/127.0.0.1:59792 sending packet to /127.0.0.1:55649

CSE473 DHTPv0.1

type:join

tag:1

ttl:100

/127.0.0.1:59792 received packet from /127.0.0.1:55649

CSE473 DHTPv0.1

type:success

tag:1

hashRange:1073741824:2147483647

succInfo:127.0.0.1:55649:0

predInfo:127.0.0.1:55649:0

ttl:98

rteTble=[(/127.0.0.1:55649,0)]

/127.0.0.1:59792 received packet from /127.0.0.1:55649

CSE473 DHTPv0.1

type:update

tag:3

predInfo:127.0.0.1:64131:536870912

ttl:99

/127.0.0.1:59792 received packet from /127.0.0.1:49751

CSE473 DHTPv0.1

type:join

tag:1

ttl:99

/127.0.0.1:59792 sending packet to /127.0.0.1:49751

CSE473 DHTPv0.1

type:success

tag:1

hashRange:1610612736:2147483647

succInfo:127.0.0.1:55649:0

predInfo:127.0.0.1:59792:1073741824

ttl:99

/127.0.0.1:59792 sending packet to /127.0.0.1:55649

CSE473 DHTPv0.1

type:update

tag:2

predInfo:127.0.0.1:49751:1610612736

ttl:100

rteTbl=[(/127.0.0.1:49751,1610612736)]

/127.0.0.1:59792 received packet from /127.0.0.1:64131

CSE473 DHTPv0.1

type:put

key:who

val:hah

clientAdr:127.0.0.1:53556

relayAdr:127.0.0.1:64131

tag:12345

ttl:98

/127.0.0.1:59792 sending packet to /127.0.0.1:49751

CSE473 DHTPv0.1

type:put

key:who

val:hah

clientAdr:127.0.0.1:53556

relayAdr:127.0.0.1:64131

tag:12345

ttl:98

/127.0.0.1:59792 received packet from /127.0.0.1:56746

CSE473 DHTPv0.1

type:put

key:blue

val:moose

tag:12345

ttl:99

/127.0.0.1:59792 sending packet to /127.0.0.1:56746

CSE473 DHTPv0.1

type:success

key:blue

val:moose

tag:12345

ttl:99

/127.0.0.1:59792 received packet from /127.0.0.1:59716

CSE473 DHTPv0.1

type:get

key:who

tag:12345

ttl:99

/127.0.0.1:59792 sending packet to /127.0.0.1:49751

CSE473 DHTPv0.1

type:get

key:who

clientAdr:127.0.0.1:59716

relayAdr:127.0.0.1:59792

tag:12345

ttl:99

/127.0.0.1:59792 received packet from /127.0.0.1:55649

CSE473 DHTPv0.1

type:success

key:who

val:hah

clientAdr:127.0.0.1:59716

relayAdr:127.0.0.1:59792

tag:12345

senderInfo:127.0.0.1:55649:0

ttl:96

rteTbl=[(/127.0.0.1:49751,1610612736)]

/127.0.0.1:59792 sending packet to /127.0.0.1:59716

CSE473 DHTPv0.1

type:success

key:who

val:hah

tag:12345

ttl:96

/127.0.0.1:59792 received packet from /127.0.0.1:64131

CSE473 DHTPv0.1

type:get

key:blue

clientAdr:127.0.0.1:65384

relayAdr:127.0.0.1:64131

tag:12345

ttl:98

/127.0.0.1:59792 sending packet to /127.0.0.1:64131

CSE473 DHTPv0.1

type:success

key:blue

val:moose

clientAdr:127.0.0.1:65384

relayAdr:127.0.0.1:64131

tag:12345

senderInfo:127.0.0.1:59792:1073741824

ttl:98

Approximately how many values are in the hash range of server number 1 when it joins the DHT? How many are in its range after the last server has joined the DHT? How many are in its range after server number 2 leaves the DHT?

*Server 1 has a hash range of approximately 500 million (1/4 of possible java integer values) when it joins the DHT.*

*The hash range for server 1 stays the same after server 3 joins (server 3 gets the top half of server 2’s hash range, which doesn’t affect server 1).*

*After server 2 leaves, it gives all of its hash range to server 1, so server 1 then has approximately 1 billion values in its hash range (1/2 of the possible java integer values).*

Type the command “cat ../cfg[0-3]” and paste the output below. Note that the port numbers shown here are those used by your servers in the order 0, 1, 2, 3.

127.0.0.1 55649

127.0.0.1 64131

127.0.0.1 59792

127.0.0.1 49751

Type the command “grep ttl:9 out1” and paste a copy of the output below. Note that this shows the *ttls* in the returned packets, allowing you to infer the number of hops that each packet took on its way through the DHT and back.

ttl:98

ttl:94

ttl:95

ttl:98

ttl:96

ttl:95

ttl:98

ttl:96

ttl:98

ttl:94

ttl:95

ttl:98

ttl:94

ttl:96

ttl:98

ttl:95

ttl:94

ttl:96

ttl:98

ttl:95

ttl:96

ttl:96

ttl:98

Find the first *get* operation that took the longest number of hops before returning to the client. What were the key and value of the returned pair?

*The first get operation with the longest number of hops (ttl=94) was for “bar”, which didn’t exist as a key so it returned NO MATCH.*

List the servers that the packet passed through, using the server numbers 0, 1, 2, 3.

*The client sent the request to server 3, which followed the path:: client🡪 3🡪 0🡪 1🡪 2🡪 3🡪 client*

Now, re-rerun script1 by typing

./script1 2 > out2

Paste the initial part of the *out2* file below (everything up through the first “*get who*” operation and the last four). Note that this allows shortcut routes, so you should expect that at least some of the packets will require fewer hops to reach the target server. Commit the output and log files to your repository.

put foo bar

/127.0.0.1:56486 sending packet to /127.0.0.1:51127

CSE473 DHTPv0.1

type:put

key:foo

val:bar

tag:12345

ttl:100

/127.0.0.1:56486 received packet from /127.0.0.1:51127

CSE473 DHTPv0.1

type:success

key:foo

val:bar

tag:12345

ttl:98

put who hah

/127.0.0.1:63663 sending packet to /127.0.0.1:58189

CSE473 DHTPv0.1

type:put

key:who

val:hah

tag:12345

ttl:100

/127.0.0.1:63663 received packet from /127.0.0.1:58189

CSE473 DHTPv0.1

type:success

key:who

val:hah

tag:12345

ttl:95

put junk mail

/127.0.0.1:51857 sending packet to /127.0.0.1:50593

CSE473 DHTPv0.1

type:put

key:junk

val:mail

tag:12345

ttl:100

/127.0.0.1:51857 received packet from /127.0.0.1:50593

CSE473 DHTPv0.1

type:success

key:junk

val:mail

tag:12345

ttl:95

put blue moose

/127.0.0.1:52702 sending packet to /127.0.0.1:61756

CSE473 DHTPv0.1

type:put

key:blue

val:moose

tag:12345

ttl:100

/127.0.0.1:52702 received packet from /127.0.0.1:61756

CSE473 DHTPv0.1

type:success

key:blue

val:moose

tag:12345

ttl:98

get foo

/127.0.0.1:54139 sending packet to /127.0.0.1:50593

CSE473 DHTPv0.1

type:get

key:foo

tag:12345

ttl:100

/127.0.0.1:54139 received packet from /127.0.0.1:50593

CSE473 DHTPv0.1

type:success

key:foo

val:bar

tag:12345

ttl:96

get who

/127.0.0.1:51424 sending packet to /127.0.0.1:61756

CSE473 DHTPv0.1

type:get

key:who

tag:12345

ttl:100

/127.0.0.1:51424 received packet from /127.0.0.1:61756

CSE473 DHTPv0.1

type:success

key:who

val:hah

tag:12345

ttl:96

get blue

/127.0.0.1:55395 sending packet to /127.0.0.1:51127

CSE473 DHTPv0.1

type:get

key:blue

tag:12345

ttl:100

/127.0.0.1:55395 received packet from /127.0.0.1:51127

CSE473 DHTPv0.1

type:success

key:blue

val:moose

tag:12345

ttl:96

get blue

/127.0.0.1:53975 sending packet to /127.0.0.1:51127

CSE473 DHTPv0.1

type:get

key:blue

tag:12345

ttl:100

/127.0.0.1:53975 received packet from /127.0.0.1:51127

CSE473 DHTPv0.1

type:success

key:blue

val:moose

tag:12345

ttl:96

get foo

/127.0.0.1:64150 sending packet to /127.0.0.1:50593

CSE473 DHTPv0.1

type:get

key:foo

tag:12345

ttl:100

/127.0.0.1:64150 received packet from /127.0.0.1:50593

CSE473 DHTPv0.1

type:success

key:foo

val:toast is tasty

tag:12345

ttl:96

get junk

/127.0.0.1:55924 sending packet to /127.0.0.1:58189

CSE473 DHTPv0.1

type:get

key:junk

tag:12345

ttl:100

/127.0.0.1:55924 received packet from /127.0.0.1:58189

CSE473 DHTPv0.1

type:success

key:junk

val:mail

tag:12345

ttl:98

Type the command “grep ttl:9 out2” and paste the output below.

ttl:98

ttl:95

ttl:95

ttl:98

ttl:96

ttl:96

ttl:98

ttl:96

ttl:98

ttl:95

ttl:96

ttl:98

ttl:96

ttl:96

ttl:98

ttl:96

ttl:96

ttl:96

ttl:98

ttl:96

ttl:96

ttl:96

ttl:98

Type the command “cat ../cfg[0-3]” and paste the output below.

127.0.0.1 51127

127.0.0.1 58189

127.0.0.1 61756

127.0.0.1 50593

Type the command “grep rteTbl log2\_[0-3]” and paste the output below.

log2\_0:rteTble=[(/127.0.0.1:61756,1073741824)]

log2\_0:rteTble=[(/127.0.0.1:61756,1073741824), (/127.0.0.1:58189,536870912)]

log2\_0:rteTbl=[(/127.0.0.1:58189,536870912)]

log2\_1:rteTble=[(/127.0.0.1:61756,1073741824)]

log2\_1:rteTble=[(/127.0.0.1:61756,1073741824), (/127.0.0.1:51127,0)]

log2\_1:rteTbl=[(/127.0.0.1:51127,0)]

log2\_1:rteTble=[(/127.0.0.1:51127,0), (/127.0.0.1:50593,1610612736)]

log2\_2:rteTble=[(/127.0.0.1:51127,0)]

log2\_2:rteTble=[(/127.0.0.1:51127,0), (/127.0.0.1:50593,1610612736)]

log2\_3:rteTble=[(/127.0.0.1:51127,0)]

log2\_3:rteTble=[(/127.0.0.1:51127,0), (/127.0.0.1:58189,536870912)]

log2\_3:rteTbl=[(/127.0.0.1:51127,0), (/127.0.0.1:61756,1073741824)]

log2\_3:rteTbl=[(/127.0.0.1:51127,0)]

List each server still in the DHT. For each server, list all of the servers in the DHT it still has routes to when the script finishes.

*Server 0: Server 1*

*Server 1: Server 0, Server 3*

*Server 3: Server 0*

Type the command “grep -B4 -A4 key:bar log2\_[0-3]” and paste the output below.

log2\_1-

log2\_1-/127.0.0.1:58189 received packet from /127.0.0.1:50593

log2\_1-CSE473 DHTPv0.1

log2\_1-type:get

log2\_1:key:bar

log2\_1-clientAdr:127.0.0.1:65426

log2\_1-relayAdr:127.0.0.1:50593

log2\_1-tag:12345

log2\_1-ttl:98

log2\_1-

log2\_1-/127.0.0.1:58189 sending packet to /127.0.0.1:61756

log2\_1-CSE473 DHTPv0.1

log2\_1-type:get

log2\_1:key:bar

log2\_1-clientAdr:127.0.0.1:65426

log2\_1-relayAdr:127.0.0.1:50593

log2\_1-tag:12345

log2\_1-ttl:98

--

log2\_2-

log2\_2-/127.0.0.1:61756 received packet from /127.0.0.1:58189

log2\_2-CSE473 DHTPv0.1

log2\_2-type:get

log2\_2:key:bar

log2\_2-clientAdr:127.0.0.1:65426

log2\_2-relayAdr:127.0.0.1:50593

log2\_2-tag:12345

log2\_2-ttl:97

log2\_2-

log2\_2-/127.0.0.1:61756 sending packet to /127.0.0.1:50593

log2\_2-CSE473 DHTPv0.1

log2\_2-type:no match

log2\_2:key:bar

log2\_2-clientAdr:127.0.0.1:65426

log2\_2-relayAdr:127.0.0.1:50593

log2\_2-tag:12345

log2\_2-senderInfo:127.0.0.1:61756:1073741824

--

log2\_3-

log2\_3-/127.0.0.1:50593 received packet from /127.0.0.1:65426

log2\_3-CSE473 DHTPv0.1

log2\_3-type:get

log2\_3:key:bar

log2\_3-tag:12345

log2\_3-ttl:99

log2\_3-

log2\_3-/127.0.0.1:50593 sending packet to /127.0.0.1:58189

log2\_3-CSE473 DHTPv0.1

log2\_3-type:get

log2\_3:key:bar

log2\_3-clientAdr:127.0.0.1:65426

log2\_3-relayAdr:127.0.0.1:50593

log2\_3-tag:12345

log2\_3-ttl:99

log2\_3-

log2\_3-/127.0.0.1:50593 received packet from /127.0.0.1:61756

log2\_3-CSE473 DHTPv0.1

log2\_3-type:no match

log2\_3:key:bar

log2\_3-clientAdr:127.0.0.1:65426

log2\_3-relayAdr:127.0.0.1:50593

log2\_3-tag:12345

log2\_3-senderInfo:127.0.0.1:61756:1073741824

--

log2\_3-rteTbl=[(/127.0.0.1:51127,0), (/127.0.0.1:61756,1073741824)]

log2\_3-/127.0.0.1:50593 sending packet to /127.0.0.1:65426

log2\_3-CSE473 DHTPv0.1

log2\_3-type:no match

log2\_3:key:bar

log2\_3-tag:12345

log2\_3-ttl:96

log2\_3-

log2\_3-/127.0.0.1:50593 received packet from /127.0.0.1:59426

Use the output to determine the sequence of servers that the “*get bar*” packet passed through. List them below, in the order that they handled the packet.

*3🡪1🡪2🡪3*

Now, re-rerun script1 once more by typing

./script1 2 cache >out2c

This enables the caching feature. Paste the *final* portion of the *out2c* file below (starting with the second “*get foo*”). Commit the output and log files to your repository.

get foo

/127.0.0.1:53276 sending packet to /127.0.0.1:53261

CSE473 DHTPv0.1

type:get

key:foo

tag:12345

ttl:100

/127.0.0.1:53276 received packet from /127.0.0.1:53261

CSE473 DHTPv0.1

type:success

key:foo

val:toast is tasty

tag:12345

ttl:98

get blue

/127.0.0.1:53277 sending packet to /127.0.0.1:53264

CSE473 DHTPv0.1

type:get

key:blue

tag:12345

ttl:100

/127.0.0.1:53277 received packet from /127.0.0.1:53264

CSE473 DHTPv0.1

type:success

key:blue

val:moose

tag:12345

ttl:96

get blue

/127.0.0.1:53278 sending packet to /127.0.0.1:53263

CSE473 DHTPv0.1

type:get

key:blue

tag:12345

ttl:100

/127.0.0.1:53278 received packet from /127.0.0.1:53263

CSE473 DHTPv0.1

type:success

key:blue

val:moose

tag:12345

ttl:98

get blue

/127.0.0.1:53279 sending packet to /127.0.0.1:53262

CSE473 DHTPv0.1

type:get

key:blue

tag:12345

ttl:100

/127.0.0.1:53279 received packet from /127.0.0.1:53262

CSE473 DHTPv0.1

type:success

key:blue

val:moose

tag:12345

ttl:98

get blue

/127.0.0.1:53280 sending packet to /127.0.0.1:53261

CSE473 DHTPv0.1

type:get

key:blue

tag:12345

ttl:100

/127.0.0.1:53280 received packet from /127.0.0.1:53261

CSE473 DHTPv0.1

type:success

key:blue

val:moose

tag:12345

ttl:96

get blue

/127.0.0.1:53281 sending packet to /127.0.0.1:53264

CSE473 DHTPv0.1

type:get

key:blue

tag:12345

ttl:100

/127.0.0.1:53281 received packet from /127.0.0.1:53264

CSE473 DHTPv0.1

type:success

key:blue

val:moose

tag:12345

ttl:98

get blue

/127.0.0.1:53282 sending packet to /127.0.0.1:53263

CSE473 DHTPv0.1

type:get

key:blue

tag:12345

ttl:100

/127.0.0.1:53282 received packet from /127.0.0.1:53263

CSE473 DHTPv0.1

type:success

key:blue

val:moose

tag:12345

ttl:98

get blue

/127.0.0.1:53283 sending packet to /127.0.0.1:53262

CSE473 DHTPv0.1

type:get

key:blue

tag:12345

ttl:100

/127.0.0.1:53283 received packet from /127.0.0.1:53262

CSE473 DHTPv0.1

type:success

key:blue

val:moose

tag:12345

ttl:98

get blue

/127.0.0.1:53284 sending packet to /127.0.0.1:53261

CSE473 DHTPv0.1

type:get

key:blue

tag:12345

ttl:100

/127.0.0.1:53284 received packet from /127.0.0.1:53261

CSE473 DHTPv0.1

type:success

key:blue

val:moose

tag:12345

ttl:98

get blue

/127.0.0.1:53285 sending packet to /127.0.0.1:53261

CSE473 DHTPv0.1

type:get

key:blue

tag:12345

ttl:100

/127.0.0.1:53285 received packet from /127.0.0.1:53261

CSE473 DHTPv0.1

type:success

key:blue

val:moose

tag:12345

ttl:98

get foo

/127.0.0.1:53286 sending packet to /127.0.0.1:53264

CSE473 DHTPv0.1

type:get

key:foo

tag:12345

ttl:100

/127.0.0.1:53286 received packet from /127.0.0.1:53264

CSE473 DHTPv0.1

type:success

key:foo

val:bar

tag:12345

ttl:98

get junk

/127.0.0.1:53287 sending packet to /127.0.0.1:53263

CSE473 DHTPv0.1

type:get

key:junk

tag:12345

ttl:100

/127.0.0.1:53287 received packet from /127.0.0.1:53263

CSE473 DHTPv0.1

type:success

key:junk

val:mail

tag:12345

ttl:98

Type the command “grep ttl:9 out2c” and paste the output below.

ttl:98

ttl:95

ttl:95

ttl:98

ttl:96

ttl:96

ttl:98

ttl:96

ttl:98

ttl:95

ttl:96

ttl:98

ttl:96

ttl:98

ttl:98

ttl:96

ttl:98

ttl:98

ttl:98

ttl:98

ttl:98

ttl:98

ttl:98

Just before server 2 starts to leave the DHT network, are there are any servers that do not have the pair (*blue*, *moose*) in their cache? If so, which ones. In either case, how do you know?

*Every server has the pair (blue, moose) cached, since each one, when given a request from a client for the pair, handles the packet such that the TTL of the packet is 98 once received by the client. This suggests that for each server queried, the packet need only go to that server and back to get the pair.*

***Part F (30 points).*** In this part, you will test your DHT in *onl* using multiple servers. Use the provided *onl* configuration file. Create a directory *473/lab3* that contains all the files in the lab3 directory from the repository. It must be this specific directory structure. Also, include copies of all the class files. Go to the *test2* directory, read *script2* to make sure you understand what it does. When you’re ready, type

./script2 1 > out1

Note that it starts eight servers, but that two of the servers are started only after some *puts* and *gets* have been performed. Type “cat ../cfg[0-7]” and paste the output below. Commit the output and log files to your repository.

192.168.7.1 39243

192.168.6.1 39553

192.168.3.2 56842

192.168.2.5 49523

192.168.2.4 56443

192.168.2.3 35973

192.168.1.1 32877

192.168.5.2 47423

Now, type “grep rteTbl log1\_[0-7]” and paste the output below.

log1\_0:rteTble=[(/192.168.2.4:56443,1073741824)]

log1\_0:rteTbl=[(/192.168.3.2:56842,536870912)]

log1\_0:rteTbl=[(/192.168.3.2:56842,536870912)]

log1\_0:rteTbl=[(/192.168.3.2:56842,536870912)]

log1\_0:rteTbl=[(/192.168.3.2:56842,536870912)]

log1\_0:rteTbl=[(/192.168.6.1:39553,268435456)]

log1\_1:rteTble=[(/192.168.3.2:56842,536870912)]

log1\_2:rteTble=[(/192.168.2.4:56443,1073741824)]

log1\_2:rteTbl=[(/192.168.2.5:49523,805306368)]

log1\_2:rteTbl=[(/192.168.2.5:49523,805306368)]

log1\_2:rteTbl=[(/192.168.2.5:49523,805306368)]

log1\_2:rteTbl=[(/192.168.2.5:49523,805306368)]

log1\_2:rteTbl=[(/192.168.2.5:49523,805306368)]

log1\_2:rteTbl=[(/192.168.2.5:49523,805306368)]

log1\_2:rteTbl=[(/192.168.2.5:49523,805306368)]

log1\_2:rteTbl=[(/192.168.2.5:49523,805306368)]

log1\_3:rteTble=[(/192.168.2.4:56443,1073741824)]

log1\_3:rteTbl=[(/192.168.2.4:56443,1073741824)]

log1\_3:rteTbl=[(/192.168.2.4:56443,1073741824)]

log1\_3:rteTbl=[(/192.168.2.4:56443,1073741824)]

log1\_4:rteTble=[(/192.168.7.1:39243,0)]

log1\_4:rteTbl=[(/192.168.1.1:32877,1610612736)]

log1\_4:rteTbl=[(/192.168.1.1:32877,1610612736)]

log1\_4:rteTbl=[(/192.168.1.1:32877,1610612736)]

log1\_4:rteTbl=[(/192.168.1.1:32877,1610612736)]

log1\_4:rteTbl=[(/192.168.2.3:35973,1342177280)]

log1\_5:rteTble=[(/192.168.1.1:32877,1610612736)]

log1\_5:rteTbl=[(/192.168.1.1:32877,1610612736)]

log1\_6:rteTble=[(/192.168.7.1:39243,0)]

log1\_6:rteTbl=[(/192.168.5.2:47423,1879048192)]

log1\_6:rteTbl=[(/192.168.5.2:47423,1879048192)]

log1\_6:rteTbl=[(/192.168.5.2:47423,1879048192)]

log1\_6:rteTbl=[(/192.168.5.2:47423,1879048192)]

log1\_7:rteTble=[(/192.168.7.1:39243,0)]

Are the final route values consistent with the contents of the configuration file? Explain why they are consistent, or if they are not, explain any discrepancies.

*The port number in each server’s last routing table is the number of the server immediately after it – i.e. they all form a circle such that 0 🡪 1 🡪 2 🡪 3 🡪 4 🡪 5 🡪 6 🡪 7 🡪 0, exactly as instructed in script2*

Next, type “grep ttl.9 out1” and paste the output below.

ttl:98

ttl:92

ttl:95

ttl:93

ttl:96

ttl:98

ttl:94

ttl:94

ttl:93

ttl:95

ttl:96

ttl:94

ttl:93

ttl:94

ttl:96

ttl:95

ttl:98

ttl:94

ttl:94

ttl:96

ttl:98

ttl:95

ttl:98

ttl:95

ttl:95

ttl:95

ttl:98

ttl:91

ttl:98

ttl:94

ttl:96

ttl:98

ttl:93

Did any of the *get*/*put* requests get routed to all 8 servers? If not, what was the largest number of servers to handle any request? How many were handled by four or more servers?

*No request went to all 8 servers – evidenced by the lack of a packet with a TTL of 90. 13 requests were handled by 4 or more servers (i.e. had a TTL of <= 94).*

Type “grep –B15 ttl.91 out1” and paste the output below.

get slim

/192.168.4.2:40619 sending packet to /192.168.1.1:32877

CSE473 DHTPv0.1

type:get

key:slim

tag:12345

ttl:100

/192.168.4.2:40619 received packet from /192.168.1.1:32877

CSE473 DHTPv0.1

type:success

key:slim

val:jim

tag:12345

ttl:91

Type the command “grep -B3 -A4 transfer log1\_0” and paste the output below.

/192.168.7.1:39243 sending packet to /192.168.6.1:39553

CSE473 DHTPv0.1

type:transfer

key:flip

val:flop

tag:4

ttl:100

/192.168.7.1:39243 sending packet to /192.168.6.1:39553

CSE473 DHTPv0.1

type:transfer

key:who

val:hah

tag:5

ttl:100

Explain the output.

*When server 1 joins, the top half of server 0’s hashes are transferred to one (specifically, the hashes of flip and who are 428313143 and 425842601, respectively, and the initial hash range of 0 is [0,* *~500 million], but when server 1 joins, server 1 takes on the hash range [~268 million,* *~500 million]).*

Now, we’re going to re-run script2 using more routes. Type

./script2 3 > out3

Type “cat ../cfg[0-7]” and paste the output below. Commit the output and log files to your repository.

192.168.7.1 45710

192.168.6.1 42545

192.168.3.2 58834

192.168.2.5 42617

192.168.2.4 42114

192.168.2.3 38247

192.168.1.1 32781

192.168.5.2 54264

Now, type “grep rteTbl log3\_[0-7]” and paste the output below.

log3\_0:rteTble=[(/192.168.2.4:42114,1073741824)]

log3\_0:rteTble=[(/192.168.2.4:42114,1073741824), (/192.168.3.2:58834,536870912)]

log3\_0:rteTble=[(/192.168.2.4:42114,1073741824), (/192.168.3.2:58834,536870912), (/192.168.6.1:42545,268435456)]

log3\_1:rteTble=[(/192.168.3.2:58834,536870912)]

log3\_2:rteTble=[(/192.168.2.4:42114,1073741824)]

log3\_2:rteTble=[(/192.168.2.4:42114,1073741824), (/192.168.2.5:42617,805306368)]

log3\_2:rteTble=[(/192.168.2.4:42114,1073741824), (/192.168.2.5:42617,805306368), (/192.168.7.1:45710,0)]

log3\_2:rteTbl=[(/192.168.2.5:42617,805306368), (/192.168.7.1:45710,0), (/192.168.5.2:54264,1879048192)]

log3\_2:rteTbl=[(/192.168.2.5:42617,805306368), (/192.168.5.2:54264,1879048192), (/192.168.2.4:42114,1073741824)]

log3\_2:rteTbl=[(/192.168.2.5:42617,805306368), (/192.168.2.4:42114,1073741824), (/192.168.1.1:32781,1610612736)]

log3\_3:rteTble=[(/192.168.2.4:42114,1073741824)]

log3\_3:rteTble=[(/192.168.2.4:42114,1073741824), (/192.168.7.1:45710,0)]

log3\_3:rteTble=[(/192.168.2.4:42114,1073741824), (/192.168.7.1:45710,0), (/192.168.1.1:32781,1610612736)]

log3\_4:rteTble=[(/192.168.7.1:45710,0)]

log3\_4:rteTble=[(/192.168.7.1:45710,0), (/192.168.1.1:32781,1610612736)]

log3\_4:rteTble=[(/192.168.7.1:45710,0), (/192.168.1.1:32781,1610612736), (/192.168.2.3:38247,1342177280)]

log3\_5:rteTble=[(/192.168.1.1:32781,1610612736)]

log3\_5:rteTble=[(/192.168.1.1:32781,1610612736), (/192.168.7.1:45710,0)]

log3\_6:rteTble=[(/192.168.7.1:45710,0)]

log3\_6:rteTble=[(/192.168.7.1:45710,0), (/192.168.5.2:54264,1879048192)]

log3\_6:rteTble=[(/192.168.7.1:45710,0), (/192.168.5.2:54264,1879048192), (/192.168.2.4:42114,1073741824)]

log3\_7:rteTble=[(/192.168.7.1:45710,0)]

Draw a picture of the eight servers arranged in a circle (label them 0-7). Draw an arrow from server *i* to server *j* if *i* has a direct route to *j* at the end of the run.

Note that some servers have more “incoming routes” than others. Explain why this happens.

*Since every routing table is allowed to have a maximum of three servers, and each server must have its successor in its routing table, any routing table with three servers referenced must have received at least two requests for keys (either get or put) that don’t hash to its (or its successor’s) range. Similarly, any routing table with two elements has received one request for a key not in its (or its successor’s) range, and a table with one element has received none.*

Next, type “grep ttl.9 out3” and paste the output below.

**ttl:9**8

**ttl:9**5

**ttl:9**6

**ttl:9**5

**ttl:9**6

**ttl:9**8

**ttl:9**6

**ttl:9**6

**ttl:9**6

**ttl:9**6

**ttl:9**6

**ttl:9**6

**ttl:9**4

**ttl:9**6

**ttl:9**6

**ttl:9**5

**ttl:9**8

**ttl:9**6

**ttl:9**6

**ttl:9**6

**ttl:9**8

**ttl:9**6

**ttl:9**8

**ttl:9**6

**ttl:9**5

**ttl:9**6

**ttl:9**8

**ttl:9**5

**ttl:9**8

**ttl:9**5

**ttl:9**6

**ttl:9**8

**ttl:9**5

What was the largest number of servers to handle any request? How many were handled by three or more or more servers? Compare these results to those you got earlier and comment on the differences.

*The smallest ttl value is 94, meaning that this was the request that involved the most servers (4 servers).*

*There were 8 requests handled by 3+ servers (determined by requests with a final ttl of 95 or less).*

*In this test run, the routing table had a size of 3 rather than 1, so generally, the requests will be handled by fewer servers because they will be forwarded to the correct server more quickly. Earlier, we had*

Type “grep -B15 ttl.95 out3” and paste the output below.

put who hah

/192.168.4.2:53985 sending packet to /192.168.3.2:48697

CSE473 DHTPv0.1

type:put

key:who

val:hah

tag:12345

ttl:100

/192.168.4.2:53985 received packet from /192.168.3.2:48697

CSE473 DHTPv0.1

type:success

key:who

val:hah

tag:12345

**ttl:95**

--

get who

/192.168.4.2:56272 sending packet to /192.168.2.5:60323

CSE473 DHTPv0.1

type:get

key:who

tag:12345

ttl:100

/192.168.4.2:56272 received packet from /192.168.2.5:60323

CSE473 DHTPv0.1

type:success

key:who

val:hah

tag:12345

**ttl:95**

--

put political follies

/192.168.4.2:45310 sending packet to /192.168.3.2:48697

CSE473 DHTPv0.1

type:put

key:political

val:follies

tag:12345

ttl:100

/192.168.4.2:45310 received packet from /192.168.3.2:48697

CSE473 DHTPv0.1

type:success

key:political

val:follies

tag:12345

**ttl:95**

--

get fantasy

/192.168.4.2:49346 sending packet to /192.168.2.5:60323

CSE473 DHTPv0.1

type:get

key:fantasy

tag:12345

ttl:100

/192.168.4.2:49346 received packet from /192.168.2.5:60323

CSE473 DHTPv0.1

type:success

key:fantasy

val:football

tag:12345

**ttl:95**

--

get slim

/192.168.4.2:48220 sending packet to /192.168.1.1:38650

CSE473 DHTPv0.1

type:get

key:slim

tag:12345

ttl:100

/192.168.4.2:48220 received packet from /192.168.1.1:38650

CSE473 DHTPv0.1

type:success

key:slim

val:jim

tag:12345

**ttl:95**

--

get chocolate

/192.168.4.2:41518 sending packet to /192.168.2.3:56412

CSE473 DHTPv0.1

type:get

key:chocolate

tag:12345

ttl:100

/192.168.4.2:41518 received packet from /192.168.2.3:56412

CSE473 DHTPv0.1

type:success

key:chocolate

val:fudge

tag:12345

**ttl:95**

--

get fantasy

/192.168.4.2:45245 sending packet to /192.168.3.2:48697

CSE473 DHTPv0.1

type:get

key:fantasy

tag:12345

ttl:100

/192.168.4.2:45245 received packet from /192.168.3.2:48697

CSE473 DHTPv0.1

type:success

key:fantasy

val:football

tag:12345

ttl:95

Look at the last *get* operation performed by the script. Which server is the packet sent to by the client?

*IP address 192.168.3.2 corresponds to Server 2 (h3x2).*

Use the log files to determine the sequence of servers that this packet passes through. List those servers below, in order.

*2🡪4🡪6🡪2*

*192.168.3.2:48697🡪 192.168.2.4:37890 🡪 192.168.1.1:38650 🡪 192.168.3.2:48697*

Look at the “route diagram” you made earlier. Is the path used by the packet consistent with your route diagram? If not, explain any discrepancy.

*In the route diagram, it was shown that server 2 has a direct route to server 6, naively suggesting that server 2 should have sent this packet directly to server 6 rather than relaying it through server 4. However, when server 2 forwarded this get request, its routing table consisted of servers 3, 4, and 7. Server 6 replaced server 7* ***because*** *of server 6’s response to server 2. Therefore the routing of this packet makes sense (the rest of the path is consistent with what is written on the diagram).*Now, we are going to re-run script2 with single routes, but with caching enabled. Type

script2 1 cache >out1c

Next, type “grep ttl.9 out1c” and paste the output below. Commit the output and log files to your repository.

**ttl:9**8

**ttl:9**2

**ttl:9**5

**ttl:9**3

**ttl:9**6

**ttl:9**8

**ttl:9**4

**ttl:9**4

**ttl:9**6

**ttl:9**5

**ttl:9**6

**ttl:9**4

**ttl:9**3

**ttl:9**4

**ttl:9**6

**ttl:9**5

**ttl:9**8

**ttl:9**6

**ttl:9**5

**ttl:9**6

**ttl:9**8

**ttl:9**5

**ttl:9**8

**ttl:9**8

**ttl:9**5

**ttl:9**8

**ttl:9**8

**ttl:9**5

**ttl:9**8

**ttl:9**6

**ttl:9**8

**ttl:9**8

**ttl:9**6

What was the largest number of servers to handle any request? How many were handled by three or more or more servers?

*The most servers to handle a request was 6 servers (making a final ttl of 92). There were 14 requests handled by 3+ servers (with ttl’s <=95).*

Compare these results to the results for the first two cases (no cache, 1 route and 3 routes) and comment on the differences.

*No caching with one route had 20 requests with 3+ servers and a most handled request passing through 7 servers. No caching with 3 routes had 8 requests with 3+ servers and a most handled request passing through 4 servers. This makes sense, since adding multiple routes (temporarily) increases efficiency for select* ***hash ranges****, whereas caching only will increase efficiency if the same* ***key*** *is called multiple times.*

Type “grep -B15 ttl.95 out1c” and paste the output below.

get foo

/192.168.4.2:45828 sending packet to /192.168.1.1:48878

CSE473 DHTPv0.1

type:get

key:foo

tag:12345

ttl:100

/192.168.4.2:45828 received packet from /192.168.1.1:48878

CSE473 DHTPv0.1

type:success

key:foo

val:bar

tag:12345

**ttl:95**

--

put flim flam

/192.168.4.2:38427 sending packet to /192.168.3.2:50012

CSE473 DHTPv0.1

type:put

key:flim

val:flam

tag:12345

ttl:100

/192.168.4.2:38427 received packet from /192.168.3.2:50012

CSE473 DHTPv0.1

type:success

key:flim

val:flam

tag:12345

**ttl:95**

--

put political follies

/192.168.4.2:51099 sending packet to /192.168.3.2:50012

CSE473 DHTPv0.1

type:put

key:political

val:follies

tag:12345

ttl:100

/192.168.4.2:51099 received packet from /192.168.3.2:50012

CSE473 DHTPv0.1

type:success

key:political

val:follies

tag:12345

**ttl:95**

--

get flip

/192.168.4.2:60256 sending packet to /192.168.2.4:56316

CSE473 DHTPv0.1

type:get

key:flip

tag:12345

ttl:100

/192.168.4.2:60256 received packet from /192.168.2.4:56316

CSE473 DHTPv0.1

type:success

key:flip

val:flop

tag:12345

**ttl:95**

--

get chocolate

/192.168.4.2:33260 sending packet to /192.168.1.1:48878

CSE473 DHTPv0.1

type:get

key:chocolate

tag:12345

ttl:100

/192.168.4.2:33260 received packet from /192.168.1.1:48878

CSE473 DHTPv0.1

type:success

key:chocolate

val:fudge

tag:12345

**ttl:95**

--

get fantasy

/192.168.4.2:37259 sending packet to /192.168.2.5:34885

CSE473 DHTPv0.1

type:get

key:fantasy

tag:12345

ttl:100

/192.168.4.2:37259 received packet from /192.168.2.5:34885

CSE473 DHTPv0.1

type:success

key:fantasy

val:football

tag:12345

**ttl:95**

--

get slim

/192.168.4.2:41434 sending packet to /192.168.1.1:48878

CSE473 DHTPv0.1

type:get

key:slim

tag:12345

ttl:100

/192.168.4.2:41434 received packet from /192.168.1.1:48878

CSE473 DHTPv0.1

type:success

key:slim

val:jim

tag:12345

**ttl:95**

Look at the last *get* operation performed by the script. Use the log files to determine the sequence of servers that this packet passes through. List those servers below, in order.

*Last get request is for fantasy football.*

*2🡪 3🡪2*

*192.168.3.2:50012 🡪 192.168.2.5:34885 🡪 192.168.3.2:50012*

Compare this to the result for earlier case of no cache and three routes. Does the request go all the way the server that is responsible for this (*key*,*value*) pair, or does some intermediate server respond, using the contents of its cache?

*No, the request does not make it all the way to the server responsible for its hash value. It is intercepted by server 3, which has the requested key/value pair in its cache and replies directly back to server 2 who responds to the client.*