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

Lab 3 Solution

##### *Jonathan Li, Capri Scharff*

***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 "failure" 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; successor 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 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");

continue;

}

if (sender == null) {

System.err.println("received packet failure");

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.out.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 is circle 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() {

Packet leavePacket = new Packet();

leavePacket.type = "leave";

leavePacket.senderInfo = myInfo;

leavePacket.tag = sendTag + 1;

sendTag++;

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

Packet response = new Packet();

while (!stopFlag){

try{

Thread.sleep(100);

}

catch(InterruptedException e){

continue;

}

}

Packet updatePred = new Packet();

updatePred.type = "update";

updatePred.succInfo = succInfo;

updatePred.tag = sendTag +1;

sendTag++;

updatePred.hashRange = new Pair<Integer, Integer>(predInfo.right,hashRange.right);

updatePred.send(sock,predecessor,debug);

Packet updateSucc = new Packet();

updateSucc.type = "update";

updateSucc.predInfo = predInfo;

updateSucc.tag = sendTag + 1;

sendTag++;

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

Set<String> keys = map.keySet();

Iterator<String> i = keys.iterator();

while (i.hasNext()){

String next = i.next().toString();

Packet transferPacket = new Packet();

transferPacket.type = "transfer";

transferPacket.key = next;

transferPacket.val = map.get(next);

transferPacket.tag = sendTag +1;

sendTag++;

transferPacket.send(sock,predecessor , debug);

}

cache.clear();

map.clear();

rteTbl.clear();

}

/\*\* 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;

predecessor = predInfo.left;

}

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 firstly 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,

\*

\* Send a packet with type join to predAdr, the server that will

\* be the current server's predecessor. Wait for the response packet

\* and if the response packet is a "success" packet from the

\*predecessor,

\* set the current server's predecessor, successor, and hashrange

\* with

\* the corresponding fields from the packet. Add the successor to the

\* current server's routing table.

\*/

public static void join(InetSocketAddress predAdr) {

Packet join\_packet = new Packet();

join\_packet.type = "join";

int tag = sendTag + 1;

sendTag++;

join\_packet.tag = tag;

join\_packet.send(sock, predAdr,debug);

Packet response = new Packet();

while(true){

InetSocketAddress remote\_address =

response.receive(sock, debug);

if(remote\_address.equals(predAdr)){

predInfo = response.predInfo;

predecessor = predAdr;

succInfo = response.succInfo;

hashRange = response.hashRange;

myInfo = new Pair<InetSocketAddress,

Integer>(myAdr,hashRange.left);

addRoute(succInfo);

break;

}

if(!response.check()){

System.out.println("DhtServer server failed to

join: " + response.reason);

return;

}

if(response.tag != join\_packet.tag ||

!response.type.equals("success")){

System.out.println("DhtServer join failed after

recieving packet: " + response.toString());

return;

}

}

}

/\*\* 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)

\*

\* Split the current server's hashrange in half, and pass the

\* upper half as the hashrange for the new server. Send a

\* "success" packet with the new server's hashrange, predecessor

\* information, and Successor information to the server that

\* sent the "join" packet. Update the current server's

\* hashrange and successor information. Update the current

\* server's previous successor with an update packet that

\* sets its new predecessor as the server that sent the

\* "join" packet.

\*/

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

//find the new hashRanges

InetSocketAddress oldSuccAdr = succInfo.left;

int oldBeginning = hashRange.left;

int oldEnd = hashRange.left+ (hashRange.right

hashRange.left)/2;

int newEnd = hashRange.right;

int newBegin = oldEnd + 1;

hashRange.left = oldBeginning;

hashRange.right = oldEnd;

//send the success packet to the server

// that requested to join

Packet successPacket = new Packet();

successPacket.type = "success";

successPacket.hashRange = new Pair<Integer, Integer>(newBegin,

newEnd);

successPacket.tag = sendTag +1;

sendTag++;

successPacket.predInfo = myInfo;

successPacket.succInfo = succInfo;

succInfo = new Pair<InetSocketAddress, Integer> (succAdr,

newBegin);

successPacket.send(sock,succAdr,debug);

//update the successor with its new predInfo

Packet updateSucc = new Packet();

updateSucc.type = "update";

updateSucc.predInfo = succInfo;

updateSucc.tag = sendTag + 1;

sendTag++;

updateSucc.send(sock, oldSuccAdr,debug);

addRoute(succInfo);

//transfer the key-value pairs to the

// new server and update the current

// server's map

Set<String> keys = map.keySet();

Iterator<String> i = keys.iterator();

while (i.hasNext() && newBegin <= newEnd){

String next = i.next().toString();

Packet transferPacket = new Packet();

transferPacket.tag = sendTag +1;

sendTag++;

transferPacket.type = "transfer";

transferPacket.key = next;

transferPacket.val = map.get(next);

transferPacket.send(sock, succAdr, debug);

map.remove(next);

newBegin++;

}

}

/\*\* Handle a get packet.

\* @param p is a get packet

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

\*

\* checks the hash of the key in packet p to determine if

\* the key hash is within the server's hashrange. If the

\* hash is within the hashrange and the map contains the key,

\* return the packet with type "success" and the value. If

\* the hash is within the hashrange but the key is not in the

\* map, retun the packet with a type "no match". If the hash is

\* not in the hashrange, forward the packet.

\*

\* If caching is on and the key is contained in the caching

\* map, return the packet with type "success" and the cached value.

\*/

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 = myInfo;

} 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);

}

// if caching is true

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

p.val = cache.get(p.key);

p.type = "success";

if(p.relayAdr == null){

replyAdr = senderAdr;

}

else{

p.senderInfo = new Pair<InetSocketAddress,

Integer> (myAdr, right); //myInfo; //same as

replyAdr = p.relayAdr;

}

}

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

\*

\* Check whether the packet's key hash is in the server's hashrange.

\* If the hash is within the hashrange, add the key-value pair to the

\* server's map and send the packet with type "success". If the hash

\* is not within the hashrange, forward the packet.

\*

\* If caching is true and the key is in the cache map, remove the

\* corresponding key-value pair from the cache map.

\*/

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

InetSocketAddress replyAdr;

int hash = hashit(p.key);

int left = hashRange.left.intValue();

int right = hashRange.right.intValue();

// hash is within this server's hashRange

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

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

p.type = "success";

if (p.relayAdr == null){

p.send(sock, senderAdr, debug);

}

else {

p.senderInfo = myInfo;

p.send(sock, p.relayAdr, debug);

}

}

//hash is not within this server's hashRange

// and the packet was directly from the client

else if (p.relayAdr == null){

p.clientAdr = senderAdr;

p.relayAdr = myAdr;

forward(p, hash);

}

//hash is not within this server's hashRange

//and p was passed from another server

else{

forward(p, hash);

}

//if the key from the put request is

//in the cache, remove the cached pair

// that corresponds to that key

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

cache.remove(p.key);

}

}

/\*\* Handle a transfer packet.

\* @param p is a transfer packet

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

\*

\* check that the key hash is within the server's hashrange. If

\* the hash is within the hashrange, add the key-value pair to the

\* map.

\*/

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

int hash = hashit(p.key);

if(hash >= hashRange.left && hash <= hashRange.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

\*

\* Set the packet's senderInfo, relayAdr, and clientAdr to null.

\* If caching is true and the packet is of type "success", cache

\* the key-value pair in the cache map.

\*/

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

InetSocketAddress client\_address = p.clientAdr;

p.senderInfo = null;

p.relayAdr = null;

p.clientAdr = null;

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

p.key != null){

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

}

p.send(sock, client\_address, 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){

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

if(route.equals(newRoute)){

return;

}

}

rteTbl.add(newRoute);

if (rteTbl.size() > numRoutes){

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

if(!route.equals(succInfo)) {

rteTbl.remove(route);

break;

}

}

}

if (debug){

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

}

}

/\*\* 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){

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

if(route.equals(rmRoute)){

rteTbl.remove(rmRoute);

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> best = null;

int maxHash = hashRange.right;

// no wrap around

if (hashRange.right < hash){

// find the largest first hash that is not greater than

//hash

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

if (route.right > maxHash && route.right <=

hash){

maxHash = route.right;

best = route;

}

}

}

// does wrap around

else{

maxHash = 0;

// check for firstHashes less than (or equal to hash)

// i.e. servers on the other side of the wrap around

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

if (route.right <= hash && route.right >=

maxHash){

best = route;

maxHash = route.right;

}

}

// if no good servers found after the wrap around

// before hash, look for the largest firstHash

//i.e. the server closest to the wrap around

if (maxHash == 0 || best == null){

for (Pair<InetSocketAddress, Integer> route :

rteTbl){

if (route.right >= maxHash){

best = route;

maxHash = route.right;

}

}

}

}

p.send(sock, best.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("clientAdr")) {

chunks = right.split(":");

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

clientAdr = new InetSocketAddress(chunks[0],

Integer.parseInt(chunks[1]));

} 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("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);

}\*/// \*\*don't think its in the packet

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

chunks = right.split(":");

if (chunks.length != 3) {

return false;

}

String ip = chunks[0];

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

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

InetSocketAddress address = new

InetSocketAddress(ip, port\_number);

senderInfo = new

Pair<InetSocketAddress,Integer>(

address, hash);

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

chunks = right.split(":");

if (chunks.length != 2) {

return false;

}

int begin = Integer.parseInt(chunks[0]);

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

hashRange = new

Pair<Integer,Integer>(begin, end);

}

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

chunks = right.split(":");

if (chunks.length != 2) {

return false;

}

String hostname = chunks[0];

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

relayAdr = new InetSocketAddress(hostname,

port\_number);

}

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

key = right;

}

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

val = right;

}

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

tag = Integer.parseInt(right);

}

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

reason = right;

}

//

// add code for missing cases

//

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(tag != -1){

s.append("tag:");

s.append(tag);

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 (hashRange != null) {

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

s.append(":"); s.append(hashRange.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 (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 (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 *DhtClient* here.

/\*

This is a simple DHT Client program.

The program takes between 3-5 command line arguements:

The first is the IP address used to bind the DatagramSocket to.

The second is the name of the configuration file.

The third is the operation("get" or "put").

The fourth is an optional key and the fifth an optional value.

usage: java DhtClient IP\_address cfg\_file operation [key] [value]

The client does no error handling.

It sends and recieves packets usingthe packet send and recieve interfaces.

\*/

import java.io.BufferedReader;

import java.io.FileInputStream;

import java.io.IOException;

import java.io.InputStreamReader;

import java.net.DatagramSocket;

import java.net.InetAddress;

import java.net.InetSocketAddress;

public class DhtClient {

public static void main(String[] args) throws IOException {

//reads from the command line and sets the values

if(args.length < 3) {

System.out.println("Incorrect Usage");

return;

}

String client\_IP\_address = args[0];

String config\_file = args[1];

String operation = args[2];

String key = null;

String value = null;

if(args.length > 3) {

key = args[3];

if(args.length > 4) {

value = args[4];

}

}

//creates IP addresses and objects to create the socket and

//read in

//the config file

InetAddress local\_IP = InetAddress.getByName(

client\_IP\_address);

DatagramSocket datagram\_socket = new DatagramSocket(0,

local\_IP);

FileInputStream file\_stream\_reader = new FileInputStream(

config\_file);

InputStreamReader reader = new

InputStreamReader(file\_stream\_reader,

"US-ASCII");

BufferedReader read\_config\_file = new BufferedReader(reader);

//reads and parses the config file for operations, keys,

//values,

//port numbers, and hostname

String data = read\_config\_file.readLine();

String[] data\_split = data.split(" ");

String hostname = data\_split[0];

int port\_number = Integer.parseInt(data\_split[1]);

InetAddress server\_IP = InetAddress.getByName(hostname);

InetSocketAddress server\_address= new

InetSocketAddress(server\_IP, port\_number);

Packet send\_packet = new Packet();

//sets the send packet values

send\_packet.type = operation;

send\_packet.key = key;

send\_packet.val = value;

send\_packet.tag = 12345;

//sends through the Packet class send interface

send\_packet.send(datagram\_socket, server\_address, true);

//receives the response

Packet receive\_packet = new Packet();

receive\_packet.receive(datagram\_socket, true);

file\_stream\_reader.close();

reader.close();

read\_config\_file.close();

}

}

***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, *e.g.,*

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) or the Linux Lab (linuxlab.seas.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 repository.**

put foo bar

/127.0.0.1:60178 sending packet to /127.0.0.1:60796

CSE473 DHTPv0.1

type:put

key:foo

val:bar

tag:12345

ttl:100

/127.0.0.1:60178 received packet from /127.0.0.1:60796

CSE473 DHTPv0.1

type:success

key:foo

val:bar

tag:12345

ttl:98

put who hah

/127.0.0.1:42318 sending packet to /127.0.0.1:60796

CSE473 DHTPv0.1

type:put

key:who

val:hah

tag:12345

ttl:100

/127.0.0.1:42318 received packet from /127.0.0.1:60796

CSE473 DHTPv0.1

type:success

key:who

val:hah

tag:12345

ttl:98

get foo

/127.0.0.1:58753 sending packet to /127.0.0.1:60796

CSE473 DHTPv0.1

type:get

key:foo

tag:12345

ttl:100

/127.0.0.1:58753 received packet from /127.0.0.1:60796

CSE473 DHTPv0.1

type:success

key:foo

val:bar

tag:12345

ttl:98

get who

/127.0.0.1:46502 sending packet to /127.0.0.1:60796

CSE473 DHTPv0.1

type:get

key:who

tag:12345

ttl:100

/127.0.0.1:46502 received packet from /127.0.0.1:60796

CSE473 DHTPv0.1

type:success

key:who

val:hah

tag:12345

ttl:98

get goodbye

/127.0.0.1:42351 sending packet to /127.0.0.1:60796

CSE473 DHTPv0.1

type:get

key:goodbye

tag:12345

ttl:100

/127.0.0.1:42351 received packet from /127.0.0.1:60796

CSE473 DHTPv0.1

type:no match

key:goodbye

tag:12345

ttl:98

get

/127.0.0.1:39293 sending packet to /127.0.0.1:60796

CSE473 DHTPv0.1

type:get

tag:12345

ttl:100

/127.0.0.1:39293 received packet from /127.0.0.1:60796

CSE473 DHTPv0.1

type:failure

tag:12345

reason:gets and puts require key and tag

ttl:98

get bar

/127.0.0.1:49384 sending packet to /127.0.0.1:60796

CSE473 DHTPv0.1

type:get

key:bar

tag:12345

ttl:100

/127.0.0.1:49384 received packet from /127.0.0.1:60796

CSE473 DHTPv0.1

type:no match

key:bar

tag:12345

ttl:98

put foo toast is tasty

/127.0.0.1:46396 sending packet to /127.0.0.1:60796

CSE473 DHTPv0.1

type:put

key:foo

val:toast is tasty

tag:12345

ttl:100

/127.0.0.1:46396 received packet from /127.0.0.1:60796

CSE473 DHTPv0.1

type:success

key:foo

val:toast is tasty

tag:12345

ttl:98

get foo

/127.0.0.1:51528 sending packet to /127.0.0.1:60796

CSE473 DHTPv0.1

type:get

key:foo

tag:12345

ttl:100

/127.0.0.1:51528 received packet from /127.0.0.1:60796

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

/127.0.0.1:56098 sending packet to /127.0.0.1:56094

CSE473 DHTPv0.1

type:put

key:foo

val:bar

tag:12345

ttl:100

/127.0.0.1:56098 received packet from /127.0.0.1:56094

CSE473 DHTPv0.1

type:success

key:foo

val:bar

tag:12345

ttl:98

put who hah

/127.0.0.1:56099 sending packet to /127.0.0.1:56096

CSE473 DHTPv0.1

type:put

key:who

val:hah

tag:12345

ttl:100

/127.0.0.1:56099 received packet from /127.0.0.1:56096

CSE473 DHTPv0.1

type:success

key:who

val:hah

tag:12345

ttl:94

put junk mail

/127.0.0.1:56100 sending packet to /127.0.0.1:56097

CSE473 DHTPv0.1

type:put

key:junk

val:mail

tag:12345

ttl:100

/127.0.0.1:56100 received packet from /127.0.0.1:56097

CSE473 DHTPv0.1

type:success

key:junk

val:mail

tag:12345

ttl:95

put blue moose

/127.0.0.1:56101 sending packet to /127.0.0.1:56095

CSE473 DHTPv0.1

type:put

key:blue

val:moose

tag:12345

ttl:100

/127.0.0.1:56101 received packet from /127.0.0.1:56095

CSE473 DHTPv0.1

type:success

key:blue

val:moose

tag:12345

ttl:98

get foo

/127.0.0.1:56102 sending packet to /127.0.0.1:56097

CSE473 DHTPv0.1

type:get

key:foo

tag:12345

ttl:100

/127.0.0.1:56102 received packet from /127.0.0.1:56097

CSE473 DHTPv0.1

type:success

key:foo

val:bar

tag:12345

ttl:96

get who

/127.0.0.1:56103 sending packet to /127.0.0.1:56095

CSE473 DHTPv0.1

type:get

key:who

tag:12345

ttl:100

/127.0.0.1:56103 received packet from /127.0.0.1:56095

CSE473 DHTPv0.1

type:success

key:who

val:hah

tag:12345

ttl:95

get blue

/127.0.0.1:56114 sending packet to /127.0.0.1:56097

CSE473 DHTPv0.1

type:get

key:blue

tag:12345

ttl:100

/127.0.0.1:56114 received packet from /127.0.0.1:56097

CSE473 DHTPv0.1

type:success

key:blue

val:moose

tag:12345

ttl:94

get blue

/127.0.0.1:56115 sending packet to /127.0.0.1:56096

CSE473 DHTPv0.1

type:get

key:blue

tag:12345

ttl:100

/127.0.0.1:56115 received packet from /127.0.0.1:56096

CSE473 DHTPv0.1

type:success

key:blue

val:moose

tag:12345

ttl:96

get blue

/127.0.0.1:56116 sending packet to /127.0.0.1:56095

CSE473 DHTPv0.1

type:get

key:blue

tag:12345

ttl:100

/127.0.0.1:56116 received packet from /127.0.0.1:56095

CSE473 DHTPv0.1

type:success

key:blue

val:moose

tag:12345

ttl:98

get blue

/127.0.0.1:56117 sending packet to /127.0.0.1:56094

CSE473 DHTPv0.1

type:get

key:blue

tag:12345

ttl:100

/127.0.0.1:56117 received packet from /127.0.0.1:56094

CSE473 DHTPv0.1

type:success

key:blue

val:moose

tag:12345

ttl:95

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 for server that holds (blue, moose) = 56096*

*The ttl of the packet returned to the client is 96 for the first “get blue” request.*

*The ttl of the packet returned to the client is 98 for the first “put blue moose” request.*

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.*

*Server 0’s port number: 56094*

*Server 1’s port number: 56096*

*Server 2’s port number: 56095*

*Server 3’s port number: 56097*

*The ttls of the packets returned to the client are: 94, 96, 98, 95, 94, 96, 98, 95*

*Server 1’s successor is server 2, server 2’s successor is server 3, server 3’s successor is server 0, and server 0’s successor is server 1.*

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

*The ttl’s are different because server 2 leaves the DHT between the two “get blue” requests. The first “get blue” request goes from server 0 to server 1 to server 2 which contains the key, value pair, and then goes to server 0 and then back to the client with a TTL of 95.*

*After server 2 leaves, server 2 transfers all its key-value pairs to server 1. The request goes to server 0, server 1(which has the key-value pair), back to server 0 and back to the client with a TTL of 96.*

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

/127.0.0.1:56095 sending packet to /127.0.0.1:56094

CSE473 DHTPv0.1

type:join

tag:1

ttl:100

/127.0.0.1:56095 received packet from /127.0.0.1:56094

CSE473 DHTPv0.1

type:success

tag:2

hashRange:1073741824:2147483647

succInfo:127.0.0.1:56094:0

ttl:99

rteTbl=[(/127.0.0.1:56094,0)]

/127.0.0.1:56095 received packet from /127.0.0.1:56094

CSE473 DHTPv0.1

type:update

tag:5

ttl:99

/127.0.0.1:56095 received packet from /127.0.0.1:56097

CSE473 DHTPv0.1

type:join

tag:1

ttl:99

/127.0.0.1:56095 sending packet to /127.0.0.1:56097

CSE473 DHTPv0.1

type:success

tag:2

hashRange:1610612736:2147483647

succInfo:127.0.0.1:56094:0

ttl:100

/127.0.0.1:56095 sending packet to /127.0.0.1:56094

CSE473 DHTPv0.1

type:update

tag:3

ttl:100

rteTbl=[(/127.0.0.1:56097,1610612736)]

/127.0.0.1:56095 received packet from /127.0.0.1:56096

CSE473 DHTPv0.1

type:put

key:who

val:hah

tag:12345

clientAdr:127.0.0.1:56099

relayAdr:127.0.0.1:56096

ttl:98

/127.0.0.1:56095 sending packet to /127.0.0.1:56097

CSE473 DHTPv0.1

type:put

key:who

val:hah

tag:12345

clientAdr:127.0.0.1:56099

relayAdr:127.0.0.1:56096

ttl:98

/127.0.0.1:56095 received packet from /127.0.0.1:56101

CSE473 DHTPv0.1

type:put

key:blue

val:moose

tag:12345

ttl:99

/127.0.0.1:56095 sending packet to /127.0.0.1:56101

CSE473 DHTPv0.1

type:success

key:blue

val:moose

tag:12345

ttl:99

/127.0.0.1:56095 received packet from /127.0.0.1:56103

CSE473 DHTPv0.1

type:get

key:who

tag:12345

ttl:99

/127.0.0.1:56095 sending packet to /127.0.0.1:56097

CSE473 DHTPv0.1

type:get

key:who

tag:12345

clientAdr:127.0.0.1:56103

relayAdr:127.0.0.1:56095

ttl:99

/127.0.0.1:56095 received packet from /127.0.0.1:56094

CSE473 DHTPv0.1

type:success

key:who

val:hah

tag:12345

clientAdr:127.0.0.1:56103

relayAdr:127.0.0.1:56095

senderInfo:127.0.0.1:56094:0

ttl:96

rteTbl=[(/127.0.0.1:56097,1610612736)]

/127.0.0.1:56095 sending packet to /127.0.0.1:56103

CSE473 DHTPv0.1

type:success

key:who

val:hah

tag:12345

ttl:96

/127.0.0.1:56095 received packet from /127.0.0.1:56096

CSE473 DHTPv0.1

type:get

key:blue

tag:12345

clientAdr:127.0.0.1:56105

relayAdr:127.0.0.1:56096

ttl:98

/127.0.0.1:56095 sending packet to /127.0.0.1:56096

CSE473 DHTPv0.1

type:success

key:blue

val:moose

tag:12345

clientAdr:127.0.0.1:56105

relayAdr:127.0.0.1:56096

senderInfo:127.0.0.1:56095:1073741824

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?

*The hash range of server 1 when it joins the DHT is [536870912, 1073741823] and it is the same after server3, the last server, joins the DHT. At both of these points, no key-value pairs have been put into the DHT so no server 1 has no key-value pairs. After server 2 leaves the DHT, it inherits all of server 2’s key value pairs and hash range so it would contain the range [536870912, 1610612735] and it would contain the key-value pairs: blue:moose, junk:mail.*

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 56094

127.0.0.1 56096

127.0.0.1 56095

127.0.0.1 56097

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

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 key:bar type:no match were the ones returned in the return pair.*

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

*The packet passed through 3, 0, 1, 2 and back to 3 and then to the client. It passes through all the servers.*

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

/127.0.0.1:51361 sending packet to /127.0.0.1:51357

CSE473 DHTPv0.1

type:put

key:foo

val:bar

tag:12345

ttl:100

/127.0.0.1:51361 received packet from /127.0.0.1:51357

CSE473 DHTPv0.1

type:success

key:foo

val:bar

tag:12345

ttl:98

put who hah

/127.0.0.1:51362 sending packet to /127.0.0.1:51359

CSE473 DHTPv0.1

type:put

key:who

val:hah

tag:12345

ttl:100

/127.0.0.1:51362 received packet from /127.0.0.1:51359

CSE473 DHTPv0.1

type:success

key:who

val:hah

tag:12345

ttl:94

put junk mail

/127.0.0.1:51363 sending packet to /127.0.0.1:51360

CSE473 DHTPv0.1

type:put

key:junk

val:mail

tag:12345

ttl:100

/127.0.0.1:51363 received packet from /127.0.0.1:51360

CSE473 DHTPv0.1

type:success

key:junk

val:mail

tag:12345

ttl:95

put blue moose

/127.0.0.1:51364 sending packet to /127.0.0.1:51358

CSE473 DHTPv0.1

type:put

key:blue

val:moose

tag:12345

ttl:100

/127.0.0.1:51364 received packet from /127.0.0.1:51358

CSE473 DHTPv0.1

type:success

key:blue

val:moose

tag:12345

ttl:98

get foo

/127.0.0.1:51365 sending packet to /127.0.0.1:51360

CSE473 DHTPv0.1

type:get

key:foo

tag:12345

ttl:100

/127.0.0.1:51365 received packet from /127.0.0.1:51360

CSE473 DHTPv0.1

type:success

key:foo

val:bar

tag:12345

ttl:96

get who

/127.0.0.1:51366 sending packet to /127.0.0.1:51358

CSE473 DHTPv0.1

type:get

key:who

tag:12345

ttl:100

/127.0.0.1:51366 received packet from /127.0.0.1:51358

CSE473 DHTPv0.1

type:success

key:who

val:hah

tag:12345

ttl:95

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

ttl:98

ttl:94

ttl:95

ttl:98

ttl:96

ttl:95

ttl:98

ttl:96

ttl:98

ttl:95

ttl:95

ttl:98

ttl:95

ttl:96

ttl:98

ttl:96

ttl:95

ttl:96

ttl:98

ttl:96

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

127.0.0.1 56094

127.0.0.1 56096

127.0.0.1 56095

127.0.0.1 56097

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

log1\_0:rteTbl=[(/127.0.0.1:56095,1073741824)]

log1\_0:rteTbl=[(/127.0.0.1:56096,536870912)]

log1\_0:rteTbl=[(/127.0.0.1:56096,536870912)]

log1\_0:rteTbl=[(/127.0.0.1:56096,536870912)]

log1\_1:rteTbl=[(/127.0.0.1:56095,1073741824)]

log1\_1:rteTbl=[(/127.0.0.1:56095,1073741824)]

log1\_2:rteTbl=[(/127.0.0.1:56094,0)]

log1\_2:rteTbl=[(/127.0.0.1:56097,1610612736)]

log1\_2:rteTbl=[(/127.0.0.1:56097,1610612736)]

log1\_2:rteTbl=[(/127.0.0.1:56097,1610612736)]

log1\_3:rteTbl=[(/127.0.0.1:56094,0)]

log1\_3:rteTbl=[(/127.0.0.1:56094,0)]

log1\_3:rteTbl=[(/127.0.0.1:56094,0)]

log1\_3:rteTbl=[(/127.0.0.1:56094,0)]

log1\_3:rteTbl=[(/127.0.0.1:56094,0)]

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.

*Servers 0, 1, 3*

*Server 0 has route to 1, server 1 has routes to 3 and 0, server 3 has routes to 0 and 1.*

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

log2\_0-

log2\_0-/127.0.0.1:51357 received packet from /127.0.0.1:51360

log2\_0-CSE473 DHTPv0.1

log2\_0-type:get

log2\_0:key:bar

log2\_0-tag:12345

log2\_0-clientAdr:127.0.0.1:51370

log2\_0-relayAdr:127.0.0.1:51360

log2\_0-ttl:98

log2\_0-

log2\_0-/127.0.0.1:51357 sending packet to /127.0.0.1:51358

log2\_0-CSE473 DHTPv0.1

log2\_0-type:get

log2\_0:key:bar

log2\_0-tag:12345

log2\_0-clientAdr:127.0.0.1:51370

log2\_0-relayAdr:127.0.0.1:51360

log2\_0-ttl:98

--

log2\_2-

log2\_2-/127.0.0.1:51358 received packet from /127.0.0.1:51357

log2\_2-CSE473 DHTPv0.1

log2\_2-type:get

log2\_2:key:bar

log2\_2-tag:12345

log2\_2-clientAdr:127.0.0.1:51370

log2\_2-relayAdr:127.0.0.1:51360

log2\_2-ttl:97

log2\_2-

log2\_2-/127.0.0.1:51358 sending packet to /127.0.0.1:51360

log2\_2-CSE473 DHTPv0.1

log2\_2-type:no match

log2\_2:key:bar

log2\_2-tag:12345

log2\_2-clientAdr:127.0.0.1:51370

log2\_2-relayAdr:127.0.0.1:51360

log2\_2-ttl:97

--

log2\_3-

log2\_3-/127.0.0.1:51360 received packet from /127.0.0.1:51370

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-/127.0.0.1:51360 sending packet to /127.0.0.1:51357

log2\_3-CSE473 DHTPv0.1

log2\_3-type:get

log2\_3:key:bar

log2\_3-tag:12345

log2\_3-clientAdr:127.0.0.1:51370

log2\_3-relayAdr:127.0.0.1:51360

log2\_3-ttl:99

log2\_3-

log2\_3-/127.0.0.1:51360 received packet from /127.0.0.1:51358

log2\_3-CSE473 DHTPv0.1

log2\_3-type:no match

log2\_3:key:bar

log2\_3-tag:12345

log2\_3-clientAdr:127.0.0.1:51370

log2\_3-relayAdr:127.0.0.1:51360

log2\_3-ttl:96

log2\_3-

log2\_3-/127.0.0.1:51360 sending packet to /127.0.0.1:51370

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-/127.0.0.1:51360 received packet from /127.0.0.1:51358

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.

It went through servers 3 to 0 to 2 to 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

/127.0.0.1:51390 sending packet to /127.0.0.1:51385

CSE473 DHTPv0.1

type:get

key:foo

tag:12345

ttl:100

/127.0.0.1:51390 received packet from /127.0.0.1:51385

CSE473 DHTPv0.1

type:success

key:foo

val:bar

tag:12345

ttl:96

get who

/127.0.0.1:51391 sending packet to /127.0.0.1:51383

CSE473 DHTPv0.1

type:get

key:who

tag:12345

ttl:100

/127.0.0.1:51391 received packet from /127.0.0.1:51383

CSE473 DHTPv0.1

type:success

key:who

val:hah

tag:12345

ttl:95

get goodbye

/127.0.0.1:51392 sending packet to /127.0.0.1:51382

CSE473 DHTPv0.1

type:get

key:goodbye

tag:12345

ttl:100

/127.0.0.1:51392 received packet from /127.0.0.1:51382

CSE473 DHTPv0.1

type:no match

key:goodbye

tag:12345

ttl:98

get blue

/127.0.0.1:51393 sending packet to /127.0.0.1:51384

CSE473 DHTPv0.1

type:get

key:blue

tag:12345

ttl:100

/127.0.0.1:51393 received packet from /127.0.0.1:51384

CSE473 DHTPv0.1

type:success

key:blue

val:moose

tag:12345

ttl:96

get

/127.0.0.1:51394 sending packet to /127.0.0.1:51383

CSE473 DHTPv0.1

type:get

tag:12345

ttl:100

/127.0.0.1:51394 received packet from /127.0.0.1:51383

CSE473 DHTPv0.1

type:failure

tag:12345

reason:gets and puts require key and tag

ttl:98

get bar

/127.0.0.1:51395 sending packet to /127.0.0.1:51385

CSE473 DHTPv0.1

type:get

key:bar

tag:12345

ttl:100

/127.0.0.1:51395 received packet from /127.0.0.1:51385

CSE473 DHTPv0.1

type:no match

key:bar

tag:12345

ttl:95

put foo toast is tasty

/127.0.0.1:51396 sending packet to /127.0.0.1:51383

CSE473 DHTPv0.1

type:put

key:foo

val:toast is tasty

tag:12345

ttl:100

/127.0.0.1:51396 received packet from /127.0.0.1:51383

CSE473 DHTPv0.1

type:success

key:foo

val:toast is tasty

tag:12345

ttl:95

get foo

/127.0.0.1:51397 sending packet to /127.0.0.1:51382

CSE473 DHTPv0.1

type:get

key:foo

tag:12345

ttl:100

/127.0.0.1:51397 received packet from /127.0.0.1:51382

CSE473 DHTPv0.1

type:success

key:foo

val:toast is tasty

tag:12345

ttl:98

get blue

/127.0.0.1:51398 sending packet to /127.0.0.1:51385

CSE473 DHTPv0.1

type:get

key:blue

tag:12345

ttl:100

/127.0.0.1:51398 received packet from /127.0.0.1:51385

CSE473 DHTPv0.1

type:success

key:blue

val:moose

tag:12345

ttl:95

get blue

/127.0.0.1:51399 sending packet to /127.0.0.1:51384

CSE473 DHTPv0.1

type:get

key:blue

tag:12345

ttl:100

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

ttl:98

ttl:94

ttl:95

ttl:98

ttl:96

ttl:95

ttl:98

ttl:96

ttl:98

ttl:95

ttl:95

ttl:98

ttl:95

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?

*Checking the log file of each server, if the server returns a success packet with the pair key-value pair (blue, moose) back to the client, then it has the pair.*

*Server 1, server 2, server 3 have it in their caches.*

***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 40168

192.168.6.1 53140

192.168.3.2 56509

192.168.2.5 56952

192.168.2.4 44935

192.168.2.3 60384

192.168.1.1 38047

192.168.5.2 54061

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

log1\_0:rteTbl=[(/192.168.2.4:44935,1073741824)]

log1\_0:rteTbl=[(/192.168.3.2:56509,536870912)]

log1\_0:rteTbl=[(/192.168.6.1:53140,268435456)]

log1\_1:rteTbl=[(/192.168.3.2:56509,536870912)]

log1\_2:rteTbl=[(/192.168.2.4:44935,1073741824)]

log1\_2:rteTbl=[(/192.168.2.5:56952,805306368)]

log1\_2:rteTbl=[(/192.168.2.5:56952,805306368)]

log1\_3:rteTbl=[(/192.168.2.4:44935,1073741824)]

log1\_3:rteTbl=[(/192.168.2.4:44935,1073741824)]

log1\_3:rteTbl=[(/192.168.2.4:44935,1073741824)]

log1\_4:rteTbl=[(/192.168.7.1:40168,0)]

log1\_4:rteTbl=[(/192.168.1.1:38047,1610612736)]

log1\_4:rteTbl=[(/192.168.1.1:38047,1610612736)]

log1\_4:rteTbl=[(/192.168.1.1:38047,1610612736)]

log1\_4:rteTbl=[(/192.168.1.1:38047,1610612736)]

log1\_4:rteTbl=[(/192.168.2.3:60384,1342177280)]

log1\_5:rteTbl=[(/192.168.1.1:38047,1610612736)]

log1\_6:rteTbl=[(/192.168.7.1:40168,0)]

log1\_6:rteTbl=[(/192.168.5.2:54061,1879048192)]

log1\_6:rteTbl=[(/192.168.5.2:54061,1879048192)]

log1\_6:rteTbl=[(/192.168.5.2:54061,1879048192)]

log1\_7:rteTbl=[(/192.168.7.1:40168,0)]

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.

*They are consistent because after each join, the routing table adds the new server to the routing table along with its correctly divided hash range which takes the upper half of the predecessor’s hash range as shown in the rteTbl. The one inconsistency that shows up is the 192.168.7.1:40168,0 which may be due to a bug in our code.*

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**

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?

*None were forwarded to all 8 servers. The largest number of servers to handle any request was 6 because the TTL on the packet was 93, and the original TTL twice, once upon entry into the DHT, and once upon receiving the response and forwarding it back to the client.*

*There are 10 requests that are handled by four or more servers.*

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

No output

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

rteTbl=[(/192.168.6.1:53140,268435456)]

/192.168.7.1:40168 sending packet to /192.168.6.1:53140

CSE473 DHTPv0.1

type:transfer

key:chocolate

val:fudge

tag:8

ttl:100

Explain the output.

*This is a transfer packet when server 1 joins, and server 0 starts transferring key-value pairs to server 1.*

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 36859

192.168.6.1 53140

192.168.3.2 49473

192.168.2.5 39504

192.168.2.4 45918

192.168.2.3 60384

192.168.1.1 38875

192.168.5.2 50878

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

log3\_0:rteTbl=[(/192.168.2.4:45918,1073741824)]

log3\_0:rteTbl=[(/192.168.2.4:45918,1073741824), (/192.168.2.4:45918,536870912)]

log3\_3:rteTbl=[(/192.168.2.5:56952,805306368)]

log3\_4:rteTbl=[(/192.168.7.1:36859,0)]

log3\_7:rteTbl=[(/192.168.5.2:54061,1879048192)]

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 routes are added to the routing table when the server receives packets from other servers that are not of type “leave”, if the server processes more requests (receives more requests from the client, has more relevant keys, ect.) then it will have more “incoming routes” than other servers.*

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

ttl:98

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 largest number of servers to handle any request was 1 in our case. It was the same in both cases.*

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

Nothing

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

*The client sends the packet to server 2.*

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

*Server 2, server3, server 2*

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.

*It is not consistent because there were issues towards the latter few requests.*

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:98

ttl:92

ttl:95

ttl:93

ttl:96

ttl:98

ttl:94

ttl:94

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

*7 servers. 6 requests were handled by 3 or more servers.*

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

*More were handled by fewer servers than no cache, 1 route, and 3 route.*

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

get foo

/192.168.4.2:55008 sending packet to /192.168.1.1:48200

CSE473 DHTPv0.1

type:get

key:foo

tag:12345

ttl:100

/192.168.4.2:55008 received packet from /192.168.1.1:48200

CSE473 DHTPv0.1

type:success

key:foo

val:bar

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.

*The request should go into server 2, then to 3 and then back to 2 and back to the client.*

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?

*The request does not go all the way to the responsible server because intermediary servers use the contents stored in their cache for the key,value pair and return it directly to the server that handled the initial request from the client.*