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

Lab 5 Report – 140 Points

##### *Your name: Haiyu Wang, Haodong Huang*

***Part A***. **[20 points]**Paste a copy of the completed source code for the *Forwarder* class below. Highlight your changes by making them **bold**(you may omit sections of the original program that contain no added code). 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.

import java.io.\*;

import java.net.\*;

import java.util.\*;

import java.util.concurrent.\*;

/\*\* Forwarder for an overlay IP router.

\*

\* This class implements a basic packet forwarder for a simplified

\* overlay IP router. It runs as a separate thread.

\*

\* An application layer thread provides new packet payloads to be

\* sent using the provided send() method, and retrieves newly arrived

\* payloads with the receive() method. Each application layer payload

\* is sent as a separate packet, where each packet includes a protocol

\* field, a ttl, a source address and a destination address.

\*/

public class Forwarder implements Runnable {

private int myIp; // this node's ip address in overlay

private int debug; // controls amount of debugging output

private Substrate sub; // Substrate object for packet IO

private double now; // current time in ns

private final double sec = 1000000000; // # of ns in a second

// forwarding table maps contains (prefix, link#) pairs

private ArrayList<Pair<Prefix,Integer>> fwdTbl;

// queues for communicating with SrcSnk

private ArrayBlockingQueue<Packet> fromSrc;

private ArrayBlockingQueue<Packet> toSnk;

// queues for communicating with Router

private ArrayBlockingQueue<Pair<Packet,Integer>> fromRtr;

private ArrayBlockingQueue<Pair<Packet,Integer>> toRtr;

private Thread myThread;

private boolean quit;

/\*\* Initialize a new Forwarder object.

\* @param myIp is this node's IP address in the overlay network,

\* expressed as a raw integer.

\* @param sub is a reference to the Substrate object that this object

\* uses to handle the socket IO

\* @param debug controls the amount of debugging output

\*/

Forwarder(int myIp, Substrate sub, int debug) {

this.myIp = myIp; this.sub = sub; this.debug = debug;

// intialize forwarding table with a default route to link 0

fwdTbl = new ArrayList<Pair<Prefix,Integer>>();

fwdTbl.add(new Pair<Prefix,Integer>(new Prefix(0,0), 0));

// create queues for SrcSnk and Router

fromSrc = new ArrayBlockingQueue<Packet>(1000,true);

toSnk = new ArrayBlockingQueue<Packet>(1000,true);

fromRtr = new ArrayBlockingQueue<Pair<Packet,Integer>>(1000,true);

toRtr = new ArrayBlockingQueue<Pair<Packet,Integer>>(1000,true);

quit = false;

}

/\*\* Start the Forwarder running. \*/

public void start() throws Exception {

myThread = new Thread(this); myThread.start();

}

/\*\* Terminate the Forwarder. \*/

public void stop() throws Exception { quit = true; myThread.join(); }

/\*\* This is the main thread for the Forwarder object.

\*

\* The forwarder enables communication between Srcsnk, Router and

\* Substrate objects.

\*

\* It receives packets from Substrate, decides whether to pass the

\* packets to

\* local objects based on the destination address field and whether to

\* pass them to

\* Srcsnk or Router based on protocol field.

\*

\* It also reads new packets provided by Srcsnk or Router. While reading

\* packets from

\* Router, it sends them to Substrate directly. While reading packets

\* from Srcsnk, it

\* looks for the next hop based on its forwarding table using the

\* longest prefix match

\* rule and then sends them to Substrate.

\*/

public void run() {

now = 0; double t0 = System.nanoTime()/sec;

while (!quit) {

now = System.nanoTime()/sec - t0;

**boolean nothing = true;**

**if(sub.incoming()) {**

**// if the Substrate has an incoming packet**

**Pair<Packet,Integer> p = sub.receive();**

**if(p.left.destAdr == myIp) {**

**// if it's addressed to this overlay router,**

**// send to the SrcSnk or the Router**

**if(p.left.protocol == 1) toSnk.offer(p.left);**

**else toRtr.offer(p);**

**}**

**else {**

**// forward it to the next hop**

**int lnk = lookup(p.left.destAdr);**

**if(lnk >= 0 && sub.ready(lnk)) {**

**sub.send(p.left, lnk);**

**}**

**}**

**nothing = false;**

**}**

**if(fromRtr.size() > 0) {**

**// else if we have a packet from the Router to send**

**// send it to the Substrate**

**try {**

**Pair<Packet, Integer> p = fromRtr.take();**

**if(p != null && sub.ready(p.right)) {**

**sub.send(p.left, p.right);**

**nothing = false;**

**}**

**} catch(Exception e) {**

**System.err.println(e);**

**System.exit(1);**

**}**

**}**

**if(fromSrc.size() > 0) {**

**// else if we have a payload from the SrcSnk to send**

**// lookup the outgoing link using dest IP address**

**// format a packet containing the payload and**

**// pass it to the Substrate**

**try {**

**Packet p = fromSrc.take();**

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

**int lnk = lookup(p.destAdr);**

**if(lnk >= 0 && sub.ready(lnk)) {**

**sub.send(p, lnk);**

**nothing = false;**

**}**

**}**

**} catch(Exception e) {**

**System.err.println(e);**

**System.exit(1);**

**}**

**}**

**if(nothing) {**

**// else, nothing to do, so take a nap**

**try {**

**Thread.sleep(1);**

**} catch(Exception e) {**

**System.err.println("Forwarder:run: "**

**+ "can't sleep " + e);**

**System.exit(1);**

**}**

**}**

**}**

**}**

/\*\* Add a route to the forwarding table.

\*

\* @param nuPrefix is a prefix to be added

\* @param nuLnk is the number of the link on which to forward

\* packets matching the prefix

\*

\* If the table already contains a route with the specified

\* prefix, the route is updated to use nuLnk. Otherwise,

\* a route is added.

\*

\* If debug>0, print the forwarding table when done

\*/

public synchronized void addRoute(Prefix nuPrefix, int nuLnk) {

**// if table contains an entry with the same prefix,**

**// just update the link**

**for(int i = 0; i < fwdTbl.size(); i++) {**

**if(fwdTbl.get(i).left.equals(nuPrefix)) {**

**fwdTbl.get(i).right = nuLnk;**

**if(debug > 0) printTable();**

**return;**

**}**

**}**

**// otherwise add an entry**

**fwdTbl.add(new Pair<Prefix, Integer>(nuPrefix, nuLnk));**

**if(debug > 0) printTable();**

}

/\*\* Print the contents of the forwarding table. \*/

public synchronized void printTable() {

String s = String.format("Forwarding table (%.3f)\n",now);

for (Pair<Prefix,Integer> rte : fwdTbl)

s += String.format("%s %s\n", rte.left, rte.right);

System.out.println(s);

}

/\*\* Lookup route in fwding table.

\*

\* @param ip is an integer representing an IP address to lookup

\* @return nextHop link number or -1, if no matching entry.

\*/

private synchronized int lookup(int ip) {

**int bestL = -1, lnk = -1;**

**for(Pair<Prefix,Integer> entry : fwdTbl) {**

**if(entry.left.matches(ip) && entry.left.leng > bestL) {**

**bestL = entry.left.leng;**

**lnk = entry.right;**

**}**

**}**

**return lnk;**

}

/\*\* Send a message to another overlay host.

\* @param message is a string to be sent to the peer

\*/

public void send(String payload, String destAdr) {

Packet p = new Packet();

p.srcAdr = myIp; p.destAdr = Util.string2ip(destAdr);

p.protocol = 1; p.ttl = 100;

p.payload = payload;

try {

fromSrc.put(p);

} catch(Exception e) {

System.err.println("Forwarder:send: put exception" + e);

System.exit(1);

}

}

/\*\* Test if Forwarder is ready to send a message.

\* @return true if Forwarder is ready

\*/

public boolean ready() { return fromSrc.remainingCapacity() > 0; }

/\*\* Get an incoming message.

\* @return next message

\*/

public Pair<String,String> receive() {

Packet p = null;

try {

p = toSnk.take();

} catch(Exception e) {

System.err.println("Forwarder:send: take exception" +e);

System.exit(1);

}

return new Pair<String,String>(

p.payload,Util.ip2string(p.srcAdr));

}

/\*\* Test for the presence of an incoming message.

\* @return true if there is an incoming message

\*/

public boolean incoming() { return toSnk.size() > 0; }

// the following methods are used by the Router

/\*\* Send a message to another overlay Router.

\* @param p is a packet to be sent to another overlay node

\* @param lnk is the number of the link the packet should be

\* forwarded on

\*/

public void sendPkt(Packet p, int lnk) {

Pair<Packet,Integer> pp = new Pair<Packet,Integer>(p,lnk);

try {

fromRtr.put(pp);

} catch(Exception e) {

System.err.println("Forwarder:sendPkt: cannot write"

+ " to fromRtr " + e);

System.exit(1);

}

// debug for print pkt

if (debug > 2) printPkt(p, lnk, 0);

}

/\*\* Test if Forwarder is ready to send a packet from Router.

\* @return true if Forwarder is ready

\*/

public boolean ready4pkt() { return fromRtr.remainingCapacity() > 0; }

/\*\* Get an incoming packet.

\* @return a Pair containing the next packet for the Router,

\* including the number of the link on which it arrived

\*/

public Pair<Packet,Integer> receivePkt() {

Pair<Packet,Integer> pp = null;

try {

pp = toRtr.take();

} catch(Exception e) {

System.err.println("Forwarder:receivePkt: cannot read"

+ " from toRtr " + e);

System.exit(1);

}

return pp;

}

/\*\* Test for the presence of an incoming packet for Router.

\* @return true if there is an incoming packet

\*/

public boolean incomingPkt() { return toRtr.size() > 0; }

public void printPkt(Packet p, int lnk, int inout){

// incoming pkt

String s;

if (inout == 1)

s = String.format("Receive");

else

s = String.format("Send");

s += String.format("Pkt from %s to %s through lnk %d\n",

Util.ip2string(p.srcAdr), Util.ip2string(p.destAdr), lnk);

s += String.format("%s\n", p.payload);

System.out.println(s);

}

}

***Part B***. **[30 points]**Paste a copy of the completed source code for the *Router* class below. Highlight your changes by making them **bold**(you may omit sections of the original program that contain no added code). 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.

import java.io.\*;

import java.net.\*;

import java.util.\*;

import java.util.concurrent.\*;

/\*\* Router module for an overlay router.

\*

\* This class implements a simplified overlay IP router. It runs as a

\* separate thread.

\*

\* The router maintains routing information in its own routing table by

\* sending

\* hello packets and advertisement packets to its neighbors periodically.

\* Link information is maintained through sending and receiving hello

\* packets.

\*

\* When the router receives a packet from the forwarder, it handles the

\* packet

\* based on the type of the packet which is written in the payload.

\* Basically there are 4 types of packets to handle:

\* hello, reply to hello, advertisement, failure advertisement.

\*

\* When there are changes of validity or output link in any route in the

\* routing table,

\* the router updates the forwarder's forwarding table correspondingly.

\*/

public class Router implements Runnable {

private Thread myThread; // thread that executes run() method

private int myIp; // ip address in the overlay

private String myIpString; // String representation

private ArrayList<Prefix> pfxList; // list of prefixes to advertise

private ArrayList<NborInfo> nborList; // list of info about neighbors

private class LinkInfo { // class used to record link information

public int peerIp; // IP address of peer in overlay net

public double cost; // in seconds

public boolean gotReply; // flag to detect hello replies

public int helloState; // set to 3 when hello reply

received

// decremented whenever hello reply

// is not received; when 0, link is down

// link cost statistics

public int count;

public double totalCost;

public double minCost;

public double maxCost;

LinkInfo() {

cost = 0; gotReply = true; helloState = 3;

count = 0; totalCost = 0; minCost = 10; maxCost = 0;

}

}

private ArrayList<LinkInfo> lnkVec; // indexed by link number

private class Route { // routing table entry

public Prefix pfx; // destination prefix for route

public double timestamp; // time this route was generated

public double cost; // cost of route in ns

public LinkedList<Integer> path;// list of router IPs;

// destination at end of list

public int outLink; // outgoing link for this route

public boolean valid; //indicate the valid of the route

**public String toString() {**

**String res = String.format("%s %.3f %.3f", pfx, timestamp, cost);**

**for(int path\_ip : path) res += " " + Util.ip2string(path\_ip);**

**return res;**

**}**

}

private ArrayList<Route> rteTbl; // routing table

private Forwarder fwdr; // reference to Forwarder object

private double now; // current time in ns

private static final double sec = 1000000000; // ns per sec

private int debug; // controls debugging output

private boolean quit; // stop thread when true

private boolean enFA; // link failure advertisement enable

/\*\* Initialize a new Router object.

\*

\* @param myIp is an integer representing the overlay IP address of

\* this node in the overlay network

\* @param fwdr is a reference to the Forwarder object through which

\* the Router sends and receives packets

\* @param pfxList is a list of prefixes advertised by this router

\* @param nborList is a list of neighbors of this node

\*

\* @param debug is an integer that controls the amount of debugging

\* information that is to be printed

\*/

Router(int myIp, Forwarder fwdr, ArrayList<Prefix> pfxList,

ArrayList<NborInfo> nborList, int debug, boolean enFA) {

this.myIp = myIp; this.myIpString = Util.ip2string(myIp);

this.fwdr = fwdr; this.pfxList = pfxList;

this.nborList = nborList; this.debug = debug;

this.enFA = enFA;

lnkVec = new ArrayList<LinkInfo>();

for (NborInfo nbor : nborList) {

LinkInfo lnk = new LinkInfo();

lnk.peerIp = nbor.ip;

lnk.cost = nbor.delay;

lnkVec.add(lnk);

}

rteTbl = new ArrayList<Route>();

quit = false;

}

/\*\* Instantiate and start a thread to execute run(). \*/

public void start() {

myThread = new Thread(this); myThread.start();

}

/\*\* Terminate the thread. \*/

public void stop() throws Exception { quit = true; myThread.join(); }

/\*\* This is the main thread for the Router object.

\*

\* The router sends hello packets to all of its neighbors every 1 second

\* and sends advertisement every 10 seconds.

\*

\* It also receives packets from the forwarder and handles them based on

\* the value of type field written in the payload. During the handling

\* process,

\* the router updates its routing table and the forwarder's forwarding

\* table

\* whenever there are appropriate changes.

\*

\* After the stop signal arrives, the main loop terminates and print

\* link status information.

\*/

public void run() {

double t0 = System.nanoTime()/sec;

now = 0;

double helloTime, pvSendTime;

helloTime = pvSendTime = now;

**while (!quit) {**

**now = System.nanoTime()/sec - t0;**

**boolean nothing = true;**

**if(now - helloTime >= 1.0) {**

**// if it's time to send hello packets, do it**

**sendHellos();**

**helloTime = now;**

**nothing = false;**

**}**

**if(now - pvSendTime >= 10.0) {**

**// else if it's time to send advertisements, do it**

**sendAdverts();**

**pvSendTime = now;**

**nothing = false;**

**}**

**if(fwdr.incomingPkt()) {**

**// else if the forwarder has an incoming packet**

**// to be processed, retrieve it and process it**

**handleIncoming();**

**nothing = false;**

**}**

**if(nothing) {**

**// else nothing to do, so take a nap**

**try {**

**Thread.sleep(1);**

**} catch(Exception e) {**

**System.err.println("Router:run: "**

**+ "can't sleep " + e);**

**System.exit(1);**

**}**

**}**

**}**

String s = String.format("Router link cost statistics\n" +

"%8s %8s %8s %8s %8s\n","peerIp","count","avgCost",

"minCost","maxCost");

for (LinkInfo lnk : lnkVec) {

if (lnk.count == 0) continue;

s += String.format("%8s %8d %8.3f %8.3f %8.3f\n",

Util.ip2string(lnk.peerIp), lnk.count,

lnk.totalCost/lnk.count,

lnk.minCost, lnk.maxCost);

}

System.out.println(s);

}

/\*\* Lookup route in routing table.

\*

\* @param pfx is IP address prefix to be looked up.

\* @return a reference to the Route that matches the prefix or null

\*/

private Route lookupRoute(Prefix pfx) {

**for(Route r : rteTbl) {**

**if(r.pfx.equals(pfx)) return r;**

**}**

**return null;**

}

/\*\* Add a route to the routing table.

\*

\* @param rte is a route to be added to the table; no check is

\* done to make sure this route does not conflict with an existing

\* route

\*/

private void addRoute(Route r) {

**Route rte = new Route();**

**rte.pfx = new Prefix(r.pfx.toString());**

**rte.timestamp = now;**

**rte.cost = r.cost;**

**rte.path = new LinkedList<Integer>();**

**for(int i : r.path) rte.path.add(i);**

**rte.outLink = r.outLink;**

**rte.valid = r.valid;**

**rteTbl.add(rte);**

}

/\*\* Update a route in the routing table.

\*

\* @param rte is a reference to a route in the routing table.

\* @param nuRte is a reference to a new route that has the same

\* prefix as rte

\* @return true if rte is modified, else false

\*

\* This method replaces certain fields in rte with fields

\* in nuRte. Specifically,

\*

\* if nuRte has a link field that refers to a disabled

\* link, ignore it and return false

\*

\* else, if the route is invalid, then update the route

\* and return true,

\*

\* else, if both routes have the same path and link,

\* then the timestamp and cost fields of rte are updated

\*

\* else, if nuRte has a cost that is less than .9 times the

\* cost of rte, then all fields in rte except the prefix fields

\* are replaced with the corresponding fields in nuRte

\*

\* else, if nuRte is at least 20 seconds newer than rte

\* (as indicated by their timestamps), then all fields of

\* rte except the prefix fields are replaced

\*

\* else, if the link field for rte refers to a link that is

\* currently disabled, replace all fields in rte but the

\* prefix fields

\*/

private boolean updateRoute(Route rte, Route nuRte) {

**if(lnkVec.get(nuRte.outLink).helloState == 0) return false;**

**if(!rte.valid) {**

**rte.cost = nuRte.cost;**

**rte.path = new LinkedList<Integer>();**

**for(int i : nuRte.path) rte.path.add(i);**

**rte.outLink = nuRte.outLink;**

**rte.valid = nuRte.valid;**

**rte.timestamp = now;**

**return true;**

**}**

**if(rte.path.equals(nuRte.path) && rte.outLink == nuRte.outLink) {**

**rte.cost = nuRte.cost;**

**rte.timestamp = now;**

**return true;**

**}**

**if(nuRte.cost < 0.9 \* rte.cost**

**|| nuRte.timestamp > 20 + rte.timestamp**

**|| lnkVec.get(rte.outLink).helloState == 0){**

**rte.cost = nuRte.cost;**

**rte.path = new LinkedList<Integer>();**

**for(int i : nuRte.path) rte.path.add(i);**

**rte.outLink = nuRte.outLink;**

**rte.valid = nuRte.valid;**

**rte.timestamp = now;**

**return true;**

**}**

**return false;**

}

/\*\* Send hello packet to all neighbors.

\*

\* First check for replies. If no reply received on some link,

\* update the link status by subtracting 1. If that makes it 0,

\* the link is considered down, so we mark all routes using

\* that link as invalid. Also, if certain routes are marked as

\* invalid, we will need to print the table if debug larger

\* than 1, and we need to send failure advertisement by

\* calling sendFailureAdvert if failure advertisement is enable.

\*/

public void sendHellos() {

**int lnk = 0;**

**for (LinkInfo lnkInfo : lnkVec) {**

**if(!lnkInfo.gotReply && lnkInfo.helloState > 0) {**

**// if no reply to the last hello, subtract 1 from**

**// link status if it's not already 0**

**lnkInfo.helloState--;**

**if(lnkInfo.helloState == 0) {**

**// go through the routes to check routes**

**// that contain the failed link**

**boolean changed = false;**

**for(Route r : rteTbl) {**

**if(r.outLink == lnk && r.valid) {**

**changed = true;**

**r.valid = false;**

**fwdr.addRoute(r.pfx, -1);**

**}**

**}**

**// print routing table if debug is enabled**

**// and valid field of route is changed**

**if(debug > 0 && changed) printTable();**

**// send link failure advertisement if enFA is enabled**

**// and valid field of route is changed**

**if(enFA && changed) sendFailureAdvert(lnk);**

**}**

**}**

**// send new hello, after setting gotReply to false**

**lnkInfo.gotReply = false;**

**Packet p = new Packet();**

**p.srcAdr = myIp; p.destAdr = lnkInfo.peerIp;**

**p.protocol = 2; p.ttl = 100;**

**p.payload = String.format("RPv0\n**

**type: hello\ntimestamp: %.3f\n", now);**

**if(fwdr.ready4pkt()) fwdr.sendPkt(p, lnk);**

**lnk++;**

**}**

}

/\*\* Send initial path vector to each of our neighbors. \*/

public void sendAdverts() {

**for(Prefix pfx : pfxList) {**

**extend(-1, String.format("RPv0\ntype:**

**+advert\npathvec: %s %.3f %.3f %s",**

**pfx.toString(), now, 0.0, myIpString));**

}

}

/\*\* Send link failure advertisement to all available neighbors

\*

\* @param failedLnk is the number of link on which is failed.

\*

\*/

public void sendFailureAdvert(int failedLnk){

**int failIp = lnkVec.get(failedLnk).peerIp;**

**String failIpString = Util.ip2string(failIp);**

**for (int lnk = 0; lnk < nborList.size(); lnk++) {**

**if (lnkVec.get(lnk).helloState == 0) continue;**

**Packet p = new Packet();**

**p.protocol = 2; p.ttl = 100;**

**p.srcAdr = myIp;**

**p.destAdr = lnkVec.get(lnk).peerIp;**

**p.payload = String.format("RPv0\ntype: fadvert\n"**

**+ "linkfail: %s %s %.3f %s\n",**

**myIpString, failIpString, now, myIpString);**

**fwdr.sendPkt(p,lnk);**

**}**

}

/\*\* Retrieve and process packet received from Forwarder.

\*

\* For hello packets, we simply echo them back.

\* For replies to our own hello packets, we update costs.

\* For advertisements, we update routing state and propagate

\* as appropriate.

\*/

public void handleIncoming() {

**// parse the packet payload**

**Pair<Packet,Integer> pp = fwdr.receivePkt();**

**Packet p = pp.left; int lnk = pp.right;**

**String[] lines = p.payload.split("\n");**

**if (!lines[0].equals("RPv0")) return;**

**String[] chunks = lines[1].split(":");**

**if (!chunks[0].equals("type")) return;**

**String type = chunks[1].trim();**

**// if it's an route advert, call handleAdvert**

**if(type.equals("advert")) handleAdvert(lines, lnk);**

**// if it's a link failure advert, call handleFailureAdvert**

**else if(type.equals("fadvert")) handleFailureAdvert(lines, lnk);**

**// if it's a hello, echo it back**

**else if(type.equals("hello")) {**

**p.destAdr = p.srcAdr;**

**p.srcAdr = myIp;**

**p.payload = String.format("RPv0\ntype: hello2u\ntimestamp: %s\n",**

**lines[2].split(":")[1].trim());**

**fwdr.sendPkt(p, lnk);**

**}**

**else {**

**// else it's a reply to a hello packet**

**// use timestamp to determine round-trip delay**

**double RTT = (now –**

**Double.parseDouble(lines[2].split(":")[1].trim())) / 2;**

**// use this to update the link cost using exponential**

**// weighted moving average method**

**lnkVec.get(lnk).cost = lnkVec.get(lnk).cost \* 0.9 + RTT \* 0.1;**

**// also, update link cost statistics**

**lnkVec.get(lnk).count++;**

**lnkVec.get(lnk).totalCost += RTT;**

**lnkVec.get(lnk).minCost = Math.min(lnkVec.get(lnk).minCost, RTT);**

**lnkVec.get(lnk).maxCost = Math.max(lnkVec.get(lnk).maxCost, RTT);**

**// also, set gotReply to true**

**lnkVec.get(lnk).gotReply = true;**

**lnkVec.get(lnk).helloState = 3;**

**}**

}

/\*\* Handle an advertisement received from another router.

\*

\* @param lines is a list of lines that defines the packet;

\* the first two lines have already been processed at this point

\*

\* @param lnk is the number of link on which the packet was received

\*/

private void handleAdvert(String[] lines, int lnk) {

**// example path vector line**

**// pathvec: 1.2.0.0/16 345.678 .052 1.2.0.1 1.2.3.4**

**String[] chunks = lines[2].split(":")[1].trim().split(" ");**

**// Form a new route, with cost equal to path vector cost**

**// plus the cost of the link on which it arrived.**

**Route nuRoute = new Route();**

**nuRoute.pfx = new Prefix(chunks[0]);**

**nuRoute.timestamp = Double.parseDouble(chunks[1]);**

**nuRoute.cost = Double.parseDouble(chunks[2]) + lnkVec.get(lnk).cost;**

**// Parse the path vector line.**

**// If there is loop in path vector, ignore this packet.**

**nuRoute.path = new LinkedList<Integer>();**

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

**int nuIp = Util.string2ip(chunks[i]);**

**if(myIp == nuIp) return;**

**nuRoute.path.add(nuIp);**

**}**

**nuRoute.outLink = lnk;**

**nuRoute.valid = true;**

**// Look for a matching route in the routing table**

**Route r = lookupRoute(nuRoute.pfx);**

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

**boolean path\_same = r.path.equals(nuRoute.path);**

**boolean link\_same = (r.outLink == nuRoute.outLink);**

**if(updateRoute(r, nuRoute)) {**

**// and update as appropriate; whenever an update**

**// changes the path, print the table if debug>0;**

**if(!path\_same && debug > 0) printTable();**

**// whenever an update changes the output link,**

**// update the forwarding table as well.**

**if(!link\_same) fwdr.addRoute(r.pfx, r.outLink);**

**// If the new route changed the routing table,**

**// extend the path vector and send it to other neighbors**

**nuRoute.path.add(0, myIp);**

**extend(lnk, String.format("RPv0\ntype:**

**+advert\npathvec: %s\n", nuRoute.toString()));**

**}**

**}**

**else {**

**addRoute(nuRoute);**

**printTable();**

**fwdr.addRoute(nuRoute.pfx, nuRoute.outLink);**

**nuRoute.path.add(0, myIp);**

**extend(lnk, String.format("RPv0\ntype: advert\npathvec: %s\n",**

**nuRoute.toString()));**

**}**

}

/\*\* Handle the failure advertisement received from another router.

\*

\* @param lines is a list of lines that defines the packet;

\* the first two lines have already been processed at this point

\*

\* @param lnk is the number of link on which the packet was received

\*/

private void handleFailureAdvert(String[] lines, int lnk) {

**// example path vector line**

**// fadvert: 1.2.0.1 1.3.0.1 345.678 1.4.0.1 1.2.0.1**

**// meaning link 1.2.0.1 to 1.3.0.1 is failed**

**String[] chunks = lines[2].split(":")[1].trim().split(" ");**

**String path = "";**

**// Parse the path vector line.**

**// If there is loop in path vector, ignore this packet.**

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

**if(myIpString == chunks[i]) return;**

**path += " " + chunks[i];**

**}**

**int firstIp = Util.string2ip(chunks[0]);**

**int secondIp = Util.string2ip(chunks[1]);**

**boolean changed = false;**

**// go through routes to check if it contains the link**

**// set the route as invalid (false) if it does**

**for(Route r : rteTbl) {**

**for(int i = 0; i < r.path.size(); i++) {**

**if(i + 1 < r.path.size() && r.path.get(i) ==**

**firstIp && r.path.get(i+1) == secondIp) {**

**// update the time stamp if route is changed**

**r.timestamp = now;**

**r.valid = false;**

**changed = true;**

**fwdr.addRoute(r.pfx, -1);**

**}**

**}**

**}**

**if(changed) {**

**// print route table if route is changed and debug is enabled**

**if(debug > 0) printTable();**

**// If one route is changed, extend the message**

**// and send it to other neighbors.**

**extend(lnk, String.format("RPv0\ntype:**

**+fadvert\nlinkfail: %s %s %s %s%s\n",**

**Util.ip2string(firstIp), Util.ip2string(secondIp), chunks[2], myIpString, path));**

**}**

}

/\*\*

\* Extend an advertisement to all of the router's neighbors through its incident links

\* except the link on which this advertisement arrives

\*

\* @param fromLnk is the link number of the link on which the advertisement arrives

\* Value of -1 means no ancestor link and is used in sending hello/advertisement packets

\* originated by the router

\*

\* @param payload is the value of payload field of the packet to be sent

\*/

**private void extend(int fromLnk, String payload) {**

**for(int i = 0; i < nborList.size(); i++) {**

**if(i == fromLnk || lnkVec.get(i).helloState == 0) continue;**

**Packet p = new Packet();**

**p.srcAdr = myIp; p.destAdr = nborList.get(i).ip;**

**p.protocol = 2; p.ttl = 100;**

**p.payload = payload;**

**fwdr.sendPkt(p, i);**

**}**

**}**

/\*\* Print the contents of the routing table. \*/

public void printTable() {

String s = String.format("Routing table (%.3f)\n"

+ "%10s %10s %8s %5s %10s \t path\n", now, "prefix",

"timestamp", "cost","link", "VLD/INVLD");

for (Route rte : rteTbl) {

s += String.format("%10s %10.3f %8.3f",

rte.pfx.toString(), rte.timestamp, rte.cost);

s += String.format(" %5d", rte.outLink);

if (rte.valid == true)

s+= String.format(" %10s", "valid");

else

s+= String.format(" %10s \t", "invalid");

for (int r :rte.path)

s += String.format (" %s",Util.ip2string(r));

if (lnkVec.get(rte.outLink).helloState == 0)

s += String.format("\t \*\* disabled link");

s += "\n";

}

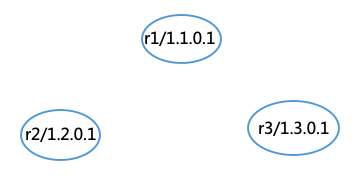
System.out.println(s);

}

}

***Part C****.* **[20 points]** Put your files for this lab in the directory ~/473/lab5. In this part, you will be running some tests using the configuration and script you will find in the *net1* sub-directory. Commit all log files to your repository after finishing this part.

1. (5 points) Draw a diagram showing the logical links joining the three routers in the overlay network defined by the configuration files *r1*, *r2* and *r3* in the *net1* sub-directory. Label the inter-router links with their assigned link costs.

**



1. (10 points) Run *script1* in the *net1* sub-directory by typing

./script1 1 .333 20 static

Paste a copy of the output below.

delta= .333 runlength= 20 static

\*\*\*\*\*\*\*\*\*\*\* log 1 \*\*\*\*\*\*\*\*\*\*\*\*\*

Final Report

Routing table (28.293)

prefix timestamp cost link VLD/INVLD path

1.3.0.0/16 20.005 0.012 1 valid 1.3.0.1

1.2.0.0/16 20.040 0.022 0 valid 1.2.0.1

Forwarding table (28.293)

0.0.0.0/0 0

1.3.0.0/16 1

1.2.0.0/16 0

Router link cost statistics

peerIp count avgCost minCost maxCost

1.2.0.1 25 0.023 0.020 0.042

1.3.0.1 25 0.013 0.010 0.029

SrcSnk statistics

destIp count avgDelay minDelay maxDelay

1.2.0.1 27 0.029 0.021 0.044

1.3.0.1 34 0.023 0.011 0.043

\*\*\*\*\*\*\*\*\*\*\* log 2 \*\*\*\*\*\*\*\*\*\*\*\*\*

Final Report

Routing table (28.239)

prefix timestamp cost link VLD/INVLD path

1.1.0.0/16 20.006 0.022 1 valid 1.1.0.1

1.3.0.0/16 20.009 0.034 1 valid 1.1.0.1 1.3.0.1

Forwarding table (28.239)

0.0.0.0/0 0

1.1.0.0/16 1

1.3.0.0/16 1

Router link cost statistics

peerIp count avgCost minCost maxCost

1.3.0.1 25 0.053 0.051 0.071

1.1.0.1 25 0.023 0.021 0.044

SrcSnk statistics

destIp count avgDelay minDelay maxDelay

1.1.0.1 35 0.029 0.021 0.044

1.3.0.1 26 0.036 0.032 0.044

\*\*\*\*\*\*\*\*\*\*\* log 3 \*\*\*\*\*\*\*\*\*\*\*\*\*

Final Report

Routing table (28.226)

prefix timestamp cost link VLD/INVLD path

1.1.0.0/16 20.024 0.013 0 valid 1.1.0.1

1.2.0.0/16 20.063 0.035 0 valid 1.1.0.1 1.2.0.1

Forwarding table (28.226)

0.0.0.0/0 0

1.1.0.0/16 0

1.2.0.0/16 0

Router link cost statistics

peerIp count avgCost minCost maxCost

1.1.0.1 26 0.014 0.011 0.034

1.2.0.1 26 0.054 0.051 0.068

SrcSnk statistics

destIp count avgDelay minDelay maxDelay

1.1.0.1 30 0.026 0.011 0.045

1.2.0.1 31 0.035 0.031 0.044

For each pair of routers *ri* and *rj*, write down the shortest path from *ri* to *rj* and the total cost of that path. Verify that the final routing tables and forwarding tables printed by *script1* are consistent with these shortest paths.

*r1 to r2:*

*Path: r1 – r2, Cost: 0.02*

*R1 to r3：*

*Path: r1 – r3, Cost: 0.01*

*r2 to r1:*

*Path: r2 – r1, Cost: 0.02*

*r2 to r3:*

*Path: r2 – r1 - r3, Cost: 0.03*

*r3 to r1:*

*Path: r3 – r1, Cost: 0.01*

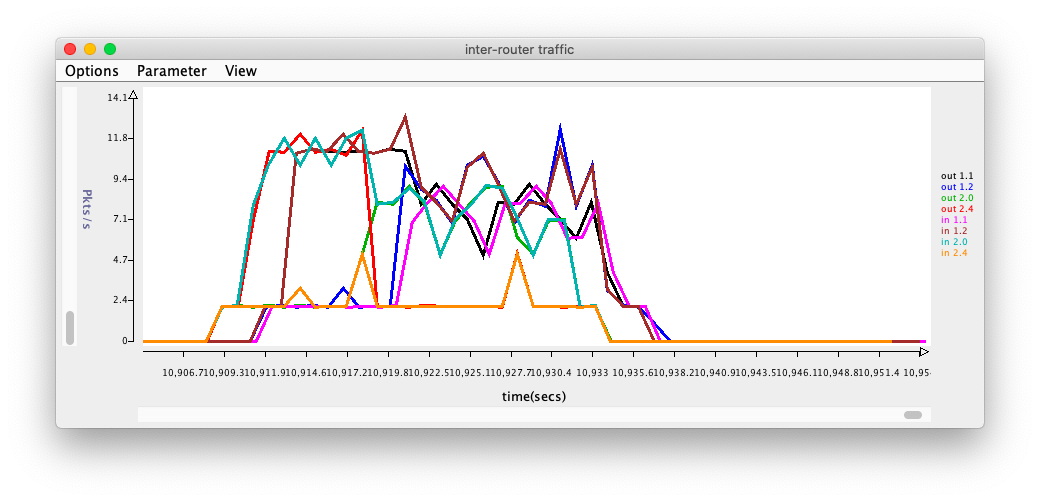
*r3 to r2:*

*Path: r3 – r1 – r2, Cost: 0.03*

*It’s consistent with the outputs of the final routing table and forwarding table*

*.*

Paste a screenshot of the monitoring window from your *script1* run below.

**

Note how the bandwidth on some links changes part way through the run. Explain why this happens. How are packets routed during the first few seconds of the run? Why does this happen?

*At beginning, there are only protocol packets on the links, so the bandwidths of the links are almost the same.*

*Some links’ bandwidths decrease as the time flows, because the costs of the links are high so that there are only protocol packets and no data packets on these links (e.g. r2 – r3)*

*Some links’ bandwidths are large because there are more packets on these links because of the low costs (e.g. r1 – r2, r1 – r3). For instance, r1’s data packets will use this link r1 – r3 to r3 and r2’s data packets will also use this link to r3.*

*During the first few seconds, the packets are routed by the prefix 0.0.0.0/0, because there are no advertisements packets so the forwarding table only contains 1 entry 0.0.0.0/0.*

How many packets per second should be sent on the link from *onl* router port 1.1 to *onl* router port 2.0 during the first part of the run? Your answer should include all packets sent by the routing algorithm and all packets sent by the *SrcSnk* that would travel over this link. Explain your answer. Does your answer match the observed packet rates?

*The first part of the run is time before the advertisement packets are sent.*

*There are 8 packets per second on this link. Port 1.1 to Port 2.0 is r1 -> r2 (one-way).*

*The 8 packets include 1 hello packets (r1 ->r2), 1 hello2u packets (r1 ->r2), 2\*1/0.333 = 6 packets (r1 ->r2, r3->r1->r2). (Since there is only 0.0.0.0/0 in forwarding table and packets are forwarded as the order of the link numbers)*

*The answer doesn’t match the observed packet rates.*

How many packets per second should be sent on the link from the *onl* router port 1.1 to *onl* router port 2.0 during the second part of the run? Explain your answer. Does your answer match the observed packet rates

*There should be 8 packets per second on the link.*

*8 packets = 1 hello packet (r1->r2) + 1 hello2u (r1->r2) + 2\*1/0.333 (r1->r2; r3->r1->r2).*

*This is compatible with the observed rate.*

1. (5 points) Run *script1* again by typing

./script1 2 .333 20 static debugg

and paste a copy of the resulting *log2\_2* file below. Add comments to the output in bold to explain how the advertisements trigger changes in the routing table. Also, explain why some received advertisements do not trigger changes to the routing table.

**#receive initial advert from 1.1.0.1, add to routing table**

/192.168.7.1:31313 received from /192.168.4.2:31313 at 9.989

protocol=2 ttl=100 srcAdr=1.1.0.1 destAdr=1.2.0.1

RPv0

type: advert

pathvec: 1.1.0.0/16 10.000 0.000 1.1.0.1

**#receive initial advert from 1.3.0.1, add to routing table**

/192.168.7.1:31313 received from /192.168.2.4:31313 at 10.009

protocol=2 ttl=100 srcAdr=1.3.0.1 destAdr=1.2.0.1

RPv0

type: advert

pathvec: 1.3.0.0/16 10.001 0.000 1.3.0.1

Routing table (9.995)

prefix timestamp cost link VLD/INVLD path

1.1.0.0/16 9.995 0.022 1 valid 1.1.0.1

Forwarding table (10.032)

0.0.0.0/0 0

1.1.0.0/16 1

Routing table (10.042)

prefix timestamp cost link VLD/INVLD path

1.1.0.0/16 9.995 0.022 1 valid 1.1.0.1

1.3.0.0/16 10.042 0.056 0 valid 1.3.0.1

Forwarding table (10.050)

0.0.0.0/0 0

1.1.0.0/16 1

1.3.0.0/16 0

**#receive advert from 1.1.0.1, route: 2->1->3, update routing table**

/192.168.7.1:31313 received from /192.168.4.2:31313 at 10.031

protocol=2 ttl=100 srcAdr=1.1.0.1 destAdr=1.2.0.1

RPv0

type: advert

pathvec: 1.3.0.0/16 10.001 0.020 1.1.0.1 1.3.0.1

/192.168.7.1:31313 sending to /192.168.4.2:31313 at 10.064

protocol=2 ttl=100 srcAdr=1.2.0.1 destAdr=1.1.0.1

RPv0

type: advert

pathvec: 1.2.0.0/16 10.039 0.000 1.2.0.1

Routing table (10.056)

prefix timestamp cost link VLD/INVLD path

1.1.0.0/16 9.995 0.022 1 valid 1.1.0.1

1.3.0.0/16 10.056 0.042 1 valid 1.1.0.1 1.3.0.1

Forwarding table (10.068)

0.0.0.0/0 0

1.1.0.0/16 1

1.3.0.0/16 1

/192.168.7.1:31313 sending to /192.168.4.2:31313 at 10.080

protocol=2 ttl=100 srcAdr=1.2.0.1 destAdr=1.1.0.1

RPv0

type: advert

pathvec: 1.3.0.0/16 10.001 0.056 1.2.0.1 1.3.0.1

/192.168.7.1:31313 sending to /192.168.2.4:31313 at 10.092

protocol=2 ttl=100 srcAdr=1.2.0.1 destAdr=1.3.0.1

RPv0

type: advert

pathvec: 1.1.0.0/16 10.000 0.022 1.2.0.1 1.1.0.1

/192.168.7.1:31313 sending to /192.168.2.4:31313 at 10.094

protocol=2 ttl=100 srcAdr=1.2.0.1 destAdr=1.3.0.1

RPv0

type: advert

pathvec: 1.2.0.0/16 10.039 0.000 1.2.0.1

**#receive advert from 1.3.0.1, route: 2->3->1, but cost is larger, no update**

/192.168.7.1:31313 received from /192.168.2.4:31313 at 10.082

protocol=2 ttl=100 srcAdr=1.3.0.1 destAdr=1.2.0.1

RPv0

type: advert

pathvec: 1.1.0.0/16 10.000 0.011 1.3.0.1 1.1.0.1

/192.168.7.1:31313 sending to /192.168.2.4:31313 at 10.124

protocol=2 ttl=100 srcAdr=1.2.0.1 destAdr=1.3.0.1

RPv0

type: advert

pathvec: 1.3.0.0/16 10.001 0.042 1.2.0.1 1.1.0.1 1.3.0.1

**#receive advert from 1.3.0.1, route has r2, ignore it**

/192.168.7.1:31313 received from /192.168.2.4:31313 at 10.097

protocol=2 ttl=100 srcAdr=1.3.0.1 destAdr=1.2.0.1

RPv0

type: advert

pathvec: 1.2.0.0/16 10.039 0.047 1.3.0.1 1.1.0.1 1.2.0.1

**#receive initial advert from 1.1.0.1 again, no update**

/192.168.7.1:31313 received from /192.168.4.2:31313 at 19.991

protocol=2 ttl=100 srcAdr=1.1.0.1 destAdr=1.2.0.1

RPv0

type: advert

pathvec: 1.1.0.0/16 20.001 0.000 1.1.0.1

**#receive advert from 1.1.0.1, route: 2->1->3, path is the same, no update**

/192.168.7.1:31313 received from /192.168.4.2:31313 at 20.004

protocol=2 ttl=100 srcAdr=1.1.0.1 destAdr=1.2.0.1

RPv0

type: advert

pathvec: 1.3.0.0/16 20.001 0.016 1.1.0.1 1.3.0.1

**#receive initial advert from 1.3.0.1again, no update**

/192.168.7.1:31313 received from /192.168.2.4:31313 at 20.018

protocol=2 ttl=100 srcAdr=1.3.0.1 destAdr=1.2.0.1

RPv0

type: advert

pathvec: 1.3.0.0/16 20.001 0.000 1.3.0.1

**#receive advert from 1.3.0.1, route: 2->3->1, no update**

/192.168.7.1:31313 received from /192.168.2.4:31313 at 20.036

protocol=2 ttl=100 srcAdr=1.3.0.1 destAdr=1.2.0.1

RPv0

type: advert

pathvec: 1.1.0.0/16 20.001 0.012 1.3.0.1 1.1.0.1

/192.168.7.1:31313 sending to /192.168.2.4:31313 at 20.052

protocol=2 ttl=100 srcAdr=1.2.0.1 destAdr=1.3.0.1

RPv0

type: advert

pathvec: 1.1.0.0/16 20.001 0.022 1.2.0.1 1.1.0.1

/192.168.7.1:31313 sending to /192.168.4.2:31313 at 20.063

protocol=2 ttl=100 srcAdr=1.2.0.1 destAdr=1.1.0.1

RPv0

type: advert

pathvec: 1.2.0.0/16 20.040 0.000 1.2.0.1

/192.168.7.1:31313 sending to /192.168.2.4:31313 at 20.075

protocol=2 ttl=100 srcAdr=1.2.0.1 destAdr=1.3.0.1

RPv0

type: advert

pathvec: 1.3.0.0/16 20.001 0.038 1.2.0.1 1.1.0.1 1.3.0.1

/192.168.7.1:31313 sending to /192.168.2.4:31313 at 20.093

protocol=2 ttl=100 srcAdr=1.2.0.1 destAdr=1.3.0.1

RPv0

type: advert

pathvec: 1.2.0.0/16 20.040 0.000 1.2.0.1

**#receive advert from 1.3.0.1, but the path has r2, ignore it**

/192.168.7.1:31313 received from /192.168.2.4:31313 at 20.089

protocol=2 ttl=100 srcAdr=1.3.0.1 destAdr=1.2.0.1

RPv0

type: advert

pathvec: 1.2.0.0/16 20.040 0.039 1.3.0.1 1.1.0.1 1.2.0.1

Final Report

Routing table (28.208)

prefix timestamp cost link VLD/INVLD path

1.1.0.0/16 19.998 0.022 1 valid 1.1.0.1

1.3.0.0/16 20.017 0.038 1 valid 1.1.0.1 1.3.0.1

Forwarding table (28.209)

0.0.0.0/0 0

1.1.0.0/16 1

1.3.0.0/16 1

Router link cost statistics

peerIp count avgCost minCost maxCost

1.3.0.1 25 0.057 0.051 0.161

1.1.0.1 24 0.022 0.021 0.025

SrcSnk statistics

destIp count avgDelay minDelay maxDelay

1.1.0.1 33 0.028 0.021 0.043

1.3.0.1 28 0.037 0.032 0.044

***Part D****.* **[30 points]** In this part, you will be running some tests using the configuration and script you will find in the *net2* sub-directory. Commit all log files to your repository after finishing this part.

1. *图片包含 电子产品

   描述已自动生成*(5 points) Draw a diagram showing the logical links joining the four routers in the overlay network defined by the configuration files *r1*, *r2*, *r3*, and *r4*. Label the inter-router links with their assigned link costs.
2. (10 points) In this part, you will disable and re-enable one of the links while *script2* is running. This is done using a filter that is installed on *onl* router port 1.1. Click on this port in the RLI and select “Filter Table” from the menu. This will show you a “delete filter” which causes all packets received on this link to be discarded. At the right end of the filter table entry you will see a check box. Click on the check box and select “Commit” from the file menu in order to turn on the filter (effectively disabling the link). To turn off the filter (and re-enable the link), uncheck the box and select “Commit” again. Based on the configuration files, predict what this filter will do to the routing tables.

*It will disable the link between r1 and r2. Thus it will affect r1’s and r2’s routing table immediately and affect r3’s and r4’s routing table after those routes using this link expire in 20 seconds.*

Now, run *script2* again (with the filter turned off) by typing

./script2 1 .333 100 static debug

after the script has run for about 30 seconds, turn on the filter. Then wait another 30 seconds and turn off the filter. Paste a copy of the content of *log1\_3* file and add comments in bold, explaining the changes to the routing table at *r3*.

Routing table (10.067)

prefix timestamp cost link VLD/INVLD path

1.4.0.0/16 10.067 0.047 1 valid 1.2.0.1 1.1.0.1 1.4.0.1

Forwarding table (10.085)

0.0.0.0/0 0

1.4.0.0/16 1

**// Received r4’s advertisement on link 1 and added route**

Routing table (10.488)

prefix timestamp cost link VLD/INVLD path

1.4.0.0/16 10.067 0.047 1 valid 1.2.0.1 1.1.0.1 1.4.0.1

1.1.0.0/16 10.488 0.035 1 valid 1.2.0.1 1.1.0.1

Forwarding table (10.500)

0.0.0.0/0 0

1.4.0.0/16 1

1.1.0.0/16 1

**// Received r1’s advertisement on link 1 and added route**

Routing table (10.748)

prefix timestamp cost link VLD/INVLD path

1.4.0.0/16 10.067 0.047 1 valid 1.2.0.1 1.1.0.1 1.4.0.1

1.1.0.0/16 10.488 0.035 1 valid 1.2.0.1 1.1.0.1

1.2.0.0/16 10.748 0.013 1 valid 1.2.0.1

Forwarding table (10.759)

0.0.0.0/0 0

1.4.0.0/16 1

1.1.0.0/16 1

1.2.0.0/16 1

**// Received r2’s advertisement on link 1 and added route**

Routing table (60.091)

prefix timestamp cost link VLD/INVLD path

1.4.0.0/16 60.091 0.102 0 valid 1.4.0.1

1.1.0.0/16 30.476 0.035 1 valid 1.2.0.1 1.1.0.1

1.2.0.0/16 50.749 0.012 1 valid 1.2.0.1

Forwarding table (60.093)

0.0.0.0/0 0

1.4.0.0/16 0

1.1.0.0/16 1

1.2.0.0/16 1

**// Link between r1 and r2 was disabled at approximately time 35. Thus,**

**after time 35 r3 could only receive r4’s advertisement on link 0(i.e. with the cost of 0.1). However, since the cost was higher, the corresponding route would not be updated until the new route was 20 seconds newer than the existing route. Currently it was at time 60 so the new route was new enough to replace the old one.**

Routing table (60.561)

prefix timestamp cost link VLD/INVLD path

1.4.0.0/16 60.091 0.102 0 valid 1.4.0.1

1.1.0.0/16 60.561 0.114 0 valid 1.4.0.1 1.1.0.1

1.2.0.0/16 50.749 0.012 1 valid 1.2.0.1

Forwarding table (60.563)

0.0.0.0/0 0

1.4.0.0/16 0

1.1.0.0/16 0

1.2.0.0/16 1

**// Same logic as above**

Routing table (70.037)

prefix timestamp cost link VLD/INVLD path

1.4.0.0/16 70.037 0.047 1 valid 1.2.0.1 1.1.0.1 1.4.0.1

1.1.0.0/16 60.561 0.114 0 valid 1.4.0.1 1.1.0.1

1.2.0.0/16 60.754 0.012 1 valid 1.2.0.1

Forwarding table (70.037)

0.0.0.0/0 0

1.4.0.0/16 1

1.1.0.0/16 0

1.2.0.0/16 1

**// Link between r1 and r2 was recovered at approximately time 65. Thus at time 70 the r4’s advertisement with lower cost received again and replaced the route during the period the link was disabled.**

Routing table (70.489)

prefix timestamp cost link VLD/INVLD path

1.4.0.0/16 70.037 0.047 1 valid 1.2.0.1 1.1.0.1 1.4.0.1

1.1.0.0/16 70.489 0.034 1 valid 1.2.0.1 1.1.0.1

1.2.0.0/16 60.754 0.012 1 valid 1.2.0.1

Forwarding table (70.498)

0.0.0.0/0 0

1.4.0.0/16 1

1.1.0.0/16 1

1.2.0.0/16 1

**// Same logic as above**

Final Report

Routing table (108.486)

prefix timestamp cost link VLD/INVLD path

1.4.0.0/16 100.032 0.046 1 valid 1.2.0.1 1.1.0.1 1.4.0.1

1.1.0.0/16 100.480 0.034 1 valid 1.2.0.1 1.1.0.1

1.2.0.0/16 100.750 0.012 1 valid 1.2.0.1

**// After time 70, the optimal path for every route was stable. Thus a route was updated whenever a new route with the same path arrived and the timestamp and cost were replaced.**

Forwarding table (108.487)

0.0.0.0/0 0

1.4.0.0/16 1

1.1.0.0/16 1

1.2.0.0/16 1

Router link cost statistics

peerIp count avgCost minCost maxCost

1.4.0.1 105 0.102 0.101 0.118

1.2.0.1 106 0.013 0.010 0.039

SrcSnk statistics

destIp count avgDelay minDelay maxDelay

1.1.0.1 71 0.049 0.031 0.116

1.2.0.1 104 0.019 0.011 0.074

1.4.0.1 69 0.054 0.042 0.102

Paste a screenshot of the monitoring window from your run below. To make everything fit in the window, click on the arrow on the x-axis repeatedly until everything is visible.

*图片包含 屏幕截图

描述已自动生成*

1. (15 points) Run *script2* again, but this time you need to enable link failure advertisement, run script2 with command

./script2 2 .333 100 static debug enFA

after the script has run for about 30 seconds, turn on the filter. Then wait another 30 seconds and turn off the filter. Paste a copy of the content of *log2\_3* file and add comments in bold, explaining all changes to the routing table at *r3*.

Routing table (9.397)

prefix timestamp cost link VLD/INVLD path

1.2.0.0/16 9.397 0.012 1 valid 1.2.0.1

Forwarding table (9.416)

0.0.0.0/0 0

1.2.0.0/16 1

**// Received r2’s advertisement on link 1 and added route**

Routing table (9.424)

prefix timestamp cost link VLD/INVLD path

1.2.0.0/16 9.397 0.012 1 valid 1.2.0.1

1.1.0.0/16 9.424 0.034 1 valid 1.2.0.1 1.1.0.1

Forwarding table (9.432)

0.0.0.0/0 0

1.2.0.0/16 1

1.1.0.0/16 1

**// Received r1’s advertisement on link 1 and added route**

Routing table (10.063)

prefix timestamp cost link VLD/INVLD path

1.2.0.0/16 9.397 0.012 1 valid 1.2.0.1

1.1.0.0/16 9.424 0.034 1 valid 1.2.0.1 1.1.0.1

1.4.0.0/16 10.063 0.048 1 valid 1.2.0.1 1.1.0.1 1.4.0.1

Forwarding table (10.076)

0.0.0.0/0 0

1.2.0.0/16 1

1.1.0.0/16 1

1.4.0.0/16 1

**// Received r4’s advertisement on link 1 and added route**

Forwarding table (34.447)

0.0.0.0/0 0

1.2.0.0/16 1

1.1.0.0/16 -1

1.4.0.0/16 1

Forwarding table (34.447)

0.0.0.0/0 0

1.2.0.0/16 1

1.1.0.0/16 -1

1.4.0.0/16 -1

Routing table (34.446)

prefix timestamp cost link VLD/INVLD path

1.2.0.0/16 29.396 0.012 1 valid 1.2.0.1

1.1.0.0/16 34.446 0.034 1 invalid 1.2.0.1 1.1.0.1

1.4.0.0/16 34.446 0.046 1 invalid 1.2.0.1 1.1.0.1 1.4.0.1

**// Received r2’s link failure advertisement and changed the route which contained the “r2 r1” sequence in the path to invalid**

Routing table (39.480)

prefix timestamp cost link VLD/INVLD path

1.2.0.0/16 39.396 0.013 1 valid 1.2.0.1

1.1.0.0/16 39.480 0.114 0 valid 1.4.0.1 1.1.0.1

1.4.0.0/16 34.446 0.046 1 invalid 1.2.0.1 1.1.0.1 1.4.0.1

Forwarding table (39.479)

0.0.0.0/0 0

1.2.0.0/16 1

1.1.0.0/16 0

1.4.0.0/16 -1

**// After link failure event, r3 could only receive r1’s advertisement on link 0. At time 40 new advertisement arrived on link 0 and replaced the old invalid one.**

Routing table (40.089)

prefix timestamp cost link VLD/INVLD path

1.2.0.0/16 39.396 0.013 1 valid 1.2.0.1

1.1.0.0/16 39.480 0.114 0 valid 1.4.0.1 1.1.0.1

1.4.0.0/16 40.089 0.102 0 valid 1.4.0.1

Forwarding table (40.092)

0.0.0.0/0 0

1.2.0.0/16 1

1.1.0.0/16 0

1.4.0.0/16 0

**// Same logic as above**

Routing table (69.414)

prefix timestamp cost link VLD/INVLD path

1.2.0.0/16 69.396 0.013 1 valid 1.2.0.1

1.1.0.0/16 69.414 0.035 1 valid 1.2.0.1 1.1.0.1

1.4.0.0/16 60.090 0.102 0 valid 1.4.0.1

Forwarding table (69.427)

0.0.0.0/0 0

1.2.0.0/16 1

1.1.0.0/16 1

1.4.0.0/16 0

**// Link between r1 and r2 was recovered at approximately time 65. Thus at time 70 the r1’s advertisement with lower cost received again and replaced the route during the period the link was disabled.**

Routing table (70.044)

prefix timestamp cost link VLD/INVLD path

1.2.0.0/16 69.396 0.013 1 valid 1.2.0.1

1.1.0.0/16 69.414 0.035 1 valid 1.2.0.1 1.1.0.1

1.4.0.0/16 70.044 0.048 1 valid 1.2.0.1 1.1.0.1 1.4.0.1

Forwarding table (70.050)

0.0.0.0/0 0

1.2.0.0/16 1

1.1.0.0/16 1

1.4.0.0/16 1

**// Same logic as above**

Final Report

Routing table (108.188)

prefix timestamp cost link VLD/INVLD path

1.2.0.0/16 99.398 0.012 1 valid 1.2.0.1

1.1.0.0/16 99.405 0.034 1 valid 1.2.0.1 1.1.0.1

1.4.0.0/16 100.038 0.046 1 valid 1.2.0.1 1.1.0.1 1.4.0.1

Forwarding table (108.189)

0.0.0.0/0 0

1.2.0.0/16 1

1.1.0.0/16 1

1.4.0.0/16 1

**// After time 70, the optimal path for every route was stable. Thus a route was updated whenever a new route with the same path arrived and the timestamp and cost were replaced.**

Router link cost statistics

peerIp count avgCost minCost maxCost

1.4.0.1 105 0.102 0.100 0.112

1.2.0.1 105 0.012 0.010 0.023

SrcSnk statistics

destIp count avgDelay minDelay maxDelay

1.1.0.1 80 0.061 0.031 0.114

1.2.0.1 102 0.018 0.011 0.076

1.4.0.1 101 0.066 0.042 0.103

Paste a screenshot of the monitoring window from your run below. To make everything fit in the window, click on the arrow on the x-axis repeatedly until everything is visible.

图片包含 屏幕截图

描述已自动生成

From both the screenshots and content of log files above, what’s the benefit of using link failure advertisement?

*Disabling link failure advertisement will cause information inconsistency problems. In scenario 1 we can know that those invalid routes in the routing table of r3 were not updated until more than 20 seconds after the link failure event. Thus, those routing information was actually incorrect before they were replaced by the correct ones and that would cause meaningless packet transmission and consume the network’s bandwidth.*

*However, if we enable link failure advertisement, the information will converge as soon as the fail advertisement is extended through the whole network. We can know from scenario 2 that the routing table was updated correctly at time 40(the moment of first advertisement event after link failure event), which is much earlier than scenario 1. A consequence of this can be seen in the statistics of SrcSnk module. Number of data packets successfully transmitted between r3-r1 or r3-r4 in scenario 2 is significantly greater than that of scenario 1.*

***Part E***. **[40 points] In this part, you will be using the configuration in the *net3* subdirectory. Commit all log files** to your repository after finishing this part.

1. (10 points) Using the information in the provided configuration files draw a network graph that represents this network. Each node in the graph should be labeled with the router number (e.g. *r*1, *r*2, ...) and the last two components of the IP address of its ONL host (so, for example, *r*1 runs on the host whose address is 192.168.6.1, so label its node in the graph with “*r*1/6.1”). Each link should be labeled with its cost and for each router, the endpoints of the links incident to it should be labeled 0, 1, 2,... where these local link numbers are determined by the order in which the neighbors are listed in the configuration file. For example, here is the relevant section from the configuration file for *r*1.

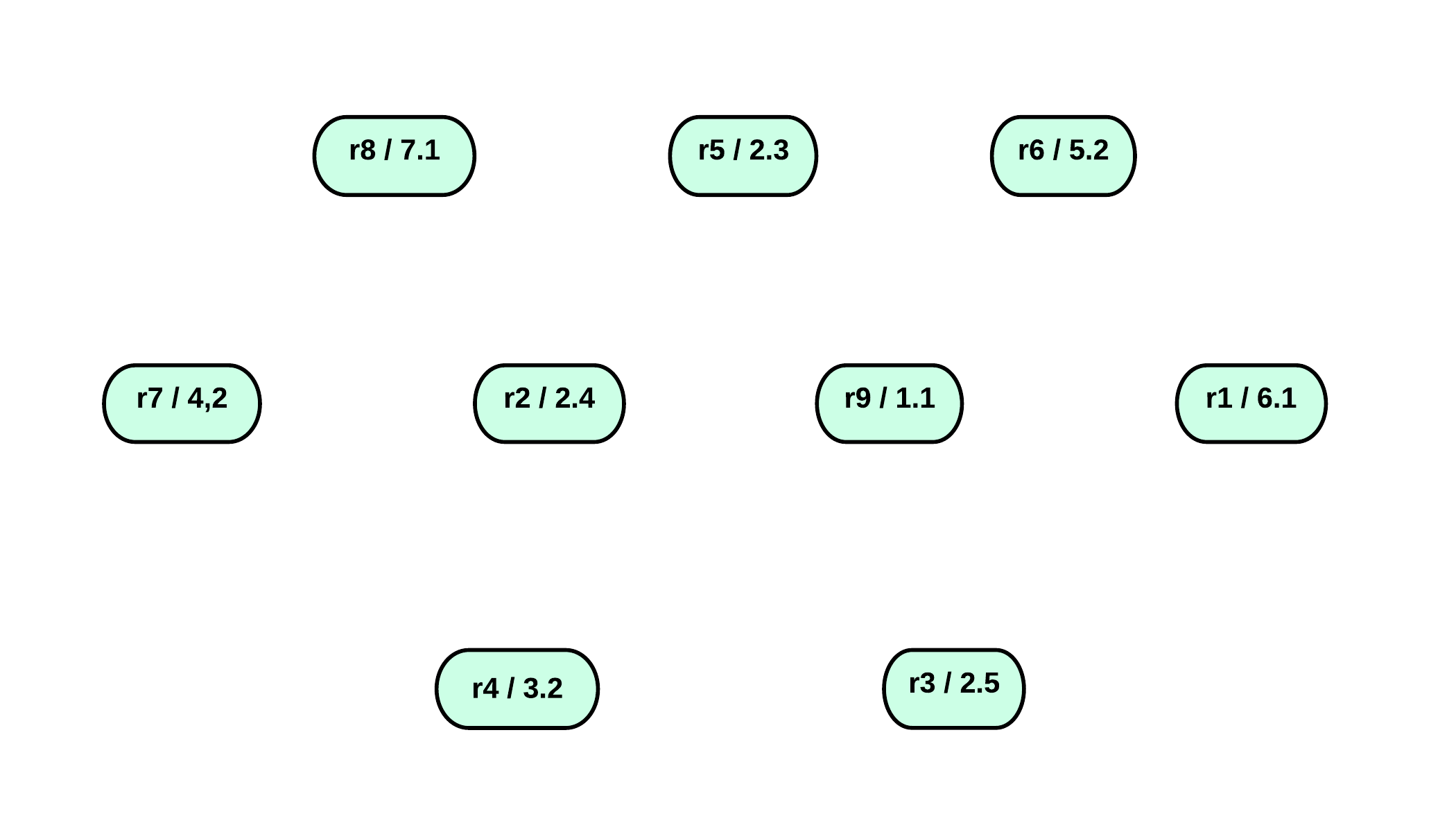
neighbor: 1.3.0.1 192.168.2.5 .03

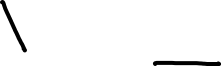
neighbor: 1.6.0.1 192.168.5.2 .01

neighbor: 1.9.0.1 192.168.1.1 .04

The link connecting to *r*3 (which has IP address 1.3.0.1 in the overlay) would have an index of 0 at *r*1. The link connecting to *r*6 (which has IP address 1.6.0.1 in the overlay) would have an index of 1 at *r*1. The link connecting to *r*9 (which has IP address 1.9.0.1 in the overlay) would have an index of 2 at *r*1.

Find a shortest path tree in your network graph, rooted at router 2. Show the edges in the shortest path tree using heavy weight lines.

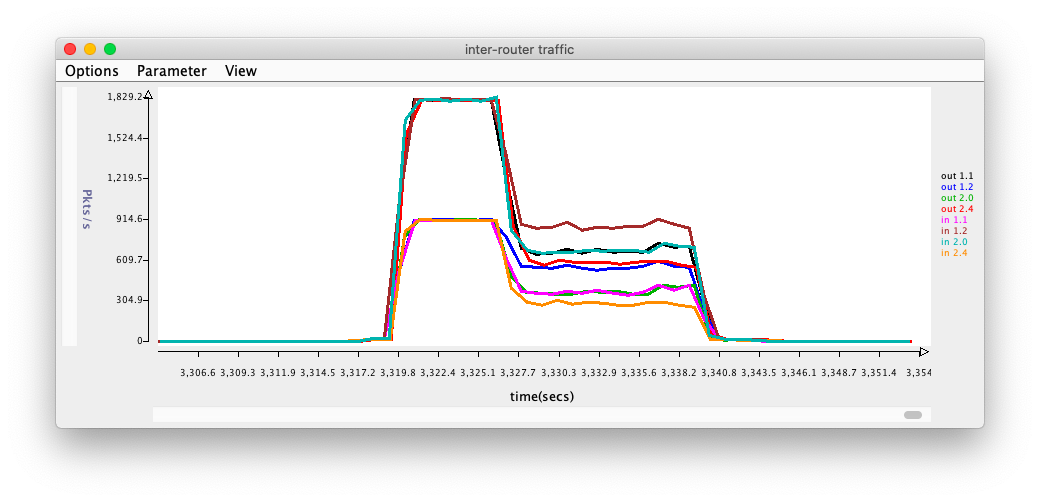
**



1. (10 points) Run the provided *script3* by typing

./script3 1 .01 20 static

Paste a screenshot of the monitoring window from your run here.

**

Paste the portion of the output from *log1\_2* showing the final routing table at *r*2.

Routing table (28.351)

prefix timestamp cost link VLD/INVLD path

1.4.0.0/16 20.031 0.033 0 valid 1.4.0.1

1.8.0.0/16 20.059 0.044 3 valid 1.5.0.1 1.8.0.1

1.9.0.0/16 20.062 0.023 4 valid 1.9.0.1

1.1.0.0/16 20.060 0.065 4 valid 1.9.0.1 1.1.0.1

1.6.0.0/16 20.082 0.050 4 valid 1.9.0.1 1.6.0.1

1.3.0.0/16 20.085 0.045 4 valid 1.9.0.1 1.3.0.1

1.5.0.0/16 20.111 0.012 3 valid 1.5.0.1

1.7.0.0/16 20.206 0.062 1 valid 1.7.0.1

Do the routes in your routing table match the shortest path tree in your network graph? If not, explain why not.

*No, the routes don’t match the shortest path tree in our network graph.*

*Because we only update routes when the cost < 0.9\*orginal cost, some routes may not updates when the costs of the new routes are smaller but not small enough.*

*For example, r2->r7, in our graph, the route should be r2->r4->r7 with the cost of 0.055. However, in the routing table the route is r2->r7 with cost of 0.06. Although 0.055<0.6, but 0.055 > 0.9\*0.6, so the route will not update.*

1. (10 points) Run *script3* by typing

./script3 2 .01 20 static debugg

Check the content of *log2\_5* log file, show all advertisements for prefix 1.7.0.0/\* that are *received* by r5 during the first round of advertisements (the ones that occur at around 10 seconds), and paste a copy of them here.

/192.168.2.3:31313 received from /192.168.7.1:31313 at 10.184

protocol=2 ttl=100 srcAdr=1.8.0.1 destAdr=1.5.0.1

RPv0

type: advert

pathvec: 1.7.0.0/16 10.004 0.023 1.8.0.1 1.7.0.1

/192.168.2.3:31313 received from /192.168.2.4:31313 at 10.214

protocol=2 ttl=100 srcAdr=1.2.0.1 destAdr=1.5.0.1

RPv0

type: advert

pathvec: 1.7.0.0/16 10.004 0.061 1.2.0.1 1.7.0.1

/192.168.2.3:31313 received from /192.168.1.1:31313 at 10.261

protocol=2 ttl=100 srcAdr=1.9.0.1 destAdr=1.5.0.1

RPv0

type: advert

pathvec: 1.7.0.0/16 10.004 0.087 1.9.0.1 1.2.0.1 1.7.0.1

Which of *r*5’s neighbors send it advertisements for this prefix? Why do these neighbors send advertisements and the others do not? Do your best to explain your observations based on the delays that advertisements will experience as they pass through the network.

*r8 (1.8.0.1), r2(1.2.0.1) and r9(1.9.0.1) send the advertisements*

*Only r6 (1.6.0.1) which is the neighbor of r5 doesn’t send the advertisements, because if r6 send the advertisements, it has the most hops and largest costs.*

*As we can see from the advertisements*

*r7 -> r8 -> r5 cost is 0.02 + 0.03 = 0.05*

*r7 -> r2 -> r5 cost is 0.06 + 0.01 = 0.07*

*r7 -> r2 -> r9 -> r5 cost is 0.06+0.02+0.02 =0.100*

*Meanwhile, the arriving time of the advertisements is compatible with their cost of their paths, which means advertisements through paths with lower costs will arrive sooner.*

*Router 6 is the neighbor of the r5, but if it want to send advertisement to r5, the path must be r7 -> r2 -> r9 -> r6 -> r5. It has most hops and the cost is 0.06+0.02+0.025+0.04 = 0.145. Thus, r6 doesn’t send advertisements.*

1. (10 points) Run *script3* which exercises variable link delays by typing

./script3 3 .01 120 debug

Paste a screenshot of the monitoring window below. To get the entire run on the display, you will need to zoom out, by clicking repeatedly on the arrowhead at the right end of the horizontal axis.

*图片包含 屏幕截图

描述已自动生成*

Type the command

grep "1.1.0.0.16....." log3\_\*

and paste the results below. Remove all lines that are *identical* to the one above them. Highlight all the places after time 10, where the path to 1.1.0.0/16 changes in a particular router, by making them bold.

log3\_2:1.1.0.0/16 10.198 0.270 4 valid 1.9.0.1 1.3.0.1 1.1.0.1

log3\_2:1.1.0.0/16 30.511 0.388 4 valid 1.9.0.1 1.6.0.1 1.1.0.1

log3\_2:1.1.0.0/16 40.742 0.469 3 valid 1.5.0.1 1.6.0.1 1.1.0.1

log3\_2:1.1.0.0/16 60.201 0.416 4 valid 1.9.0.1 1.1.0.1

log3\_2:1.1.0.0/16 100.281 0.532 4 valid 1.9.0.1 1.1.0.1

log3\_2:1.1.0.0/16 120.348 0.575 4 valid 1.9.0.1 1.1.0.1

log3\_2:1.1.0.0/16 120.348 0.575 4 invalid 1.9.0.1 1.1.0.1 \*\* disabled link

log3\_3:1.1.0.0/16 10.073 0.068 1 valid 1.1.0.1

log3\_3:1.1.0.0/16 20.070 0.141 1 valid 1.1.0.1

log3\_3:1.1.0.0/16 60.153 0.262 1 valid 1.1.0.1

log3\_3:1.1.0.0/16 90.216 0.194 1 valid 1.1.0.1

log3\_3:1.1.0.0/16 100.235 0.157 1 valid 1.1.0.1

log3\_3:1.1.0.0/16 120.277 0.274 1 valid 1.1.0.1

log3\_4:1.1.0.0/16 10.246 0.338 1 valid 1.2.0.1 1.9.0.1 1.3.0.1 1.1.0.1

log3\_4:1.1.0.0/16 30.518 0.539 2 valid 1.3.0.1 1.1.0.1

log3\_4:1.1.0.0/16 90.546 0.544 2 valid 1.3.0.1 1.1.0.1

log3\_4:1.1.0.0/16 120.498 0.481 2 valid 1.3.0.1 1.1.0.1

log3\_5:1.1.0.0/16 9.842 0.356 2 valid 1.9.0.1 1.3.0.1 1.1.0.1

log3\_5:1.1.0.0/16 9.859 0.119 0 valid 1.6.0.1 1.1.0.1

log3\_5:1.1.0.0/16 29.511 0.135 0 valid 1.6.0.1 1.1.0.1

log3\_5:1.1.0.0/16 39.812 0.160 0 valid 1.6.0.1 1.1.0.1

log3\_5:1.1.0.0/16 50.133 0.284 0 valid 1.6.0.1 1.1.0.1

log3\_5:1.1.0.0/16 99.724 0.231 0 valid 1.6.0.1 1.1.0.1

log3\_5:1.1.0.0/16 119.896 0.303 0 valid 1.6.0.1 1.1.0.1

log3\_6:1.1.0.0/16 10.243 0.046 0 valid 1.1.0.1

log3\_6:1.1.0.0/16 70.350 0.255 0 valid 1.1.0.1

log3\_6:1.1.0.0/16 90.067 0.151 0 valid 1.1.0.1

log3\_6:1.1.0.0/16 120.193 0.157 0 valid 1.1.0.1

log3\_7:1.1.0.0/16 10.343 0.474 2 valid 1.4.0.1 1.2.0.1 1.9.0.1 1.3.0.1 1.1.0.1

log3\_7:1.1.0.0/16 10.454 0.389 1 valid 1.2.0.1 1.9.0.1 1.3.0.1 1.1.0.1

log3\_7:1.1.0.0/16 11.039 0.288 0 valid 1.8.0.1 1.5.0.1 1.6.0.1 1.1.0.1

log3\_7:1.1.0.0/16 31.004 0.482 0 valid 1.8.0.1 1.5.0.1 1.6.0.1 1.1.0.1

log3\_7:1.1.0.0/16 70.807 0.666 1 valid 1.2.0.1 1.9.0.1 1.1.0.1

log3\_7:1.1.0.0/16 91.148 0.867 1 valid 1.2.0.1 1.9.0.1 1.1.0.1

log3\_7:1.1.0.0/16 100.637 0.571 2 valid 1.4.0.1 1.3.0.1 1.1.0.1

log3\_7:1.1.0.0/16 110.748 0.772 2 valid 1.4.0.1 1.3.0.1 1.1.0.1

log3\_7:1.1.0.0/16 120.651 0.668 2 valid 1.4.0.1 1.3.0.1 1.1.0.1

log3\_8:1.1.0.0/16 10.397 0.538 1 valid 1.7.0.1 1.4.0.1 1.2.0.1 1.9.0.1 1.3.0.1 1.1.0.1

log3\_8:1.1.0.0/16 10.517 0.453 1 valid 1.7.0.1 1.2.0.1 1.9.0.1 1.3.0.1 1.1.0.1

log3\_8:1.1.0.0/16 10.821 0.224 0 valid 1.5.0.1 1.6.0.1 1.1.0.1

log3\_8:1.1.0.0/16 20.581 0.340 0 valid 1.5.0.1 1.6.0.1 1.1.0.1

log3\_8:1.1.0.0/16 30.583 0.298 0 valid 1.5.0.1 1.6.0.1 1.1.0.1

log3\_8:1.1.0.0/16 40.816 0.359 0 valid 1.5.0.1 1.6.0.1 1.1.0.1

log3\_8:1.1.0.0/16 51.078 0.503 0 valid 1.5.0.1 1.6.0.1 1.1.0.1

log3\_8:1.1.0.0/16 90.376 0.485 0 valid 1.5.0.1 1.6.0.1 1.1.0.1

log3\_8:1.1.0.0/16 110.688 0.519 0 valid 1.5.0.1 1.6.0.1 1.1.0.1

log3\_8:1.1.0.0/16 120.840 0.574 0 valid 1.5.0.1 1.6.0.1 1.1.0.1

log3\_9:1.1.0.0/16 10.136 0.160 3 valid 1.3.0.1 1.1.0.1

log3\_9:1.1.0.0/16 20.133 0.287 3 valid 1.3.0.1 1.1.0.1

log3\_9:1.1.0.0/16 30.411 0.224 2 valid 1.6.0.1 1.1.0.1

log3\_9:1.1.0.0/16 40.368 0.393 2 valid 1.6.0.1 1.1.0.1

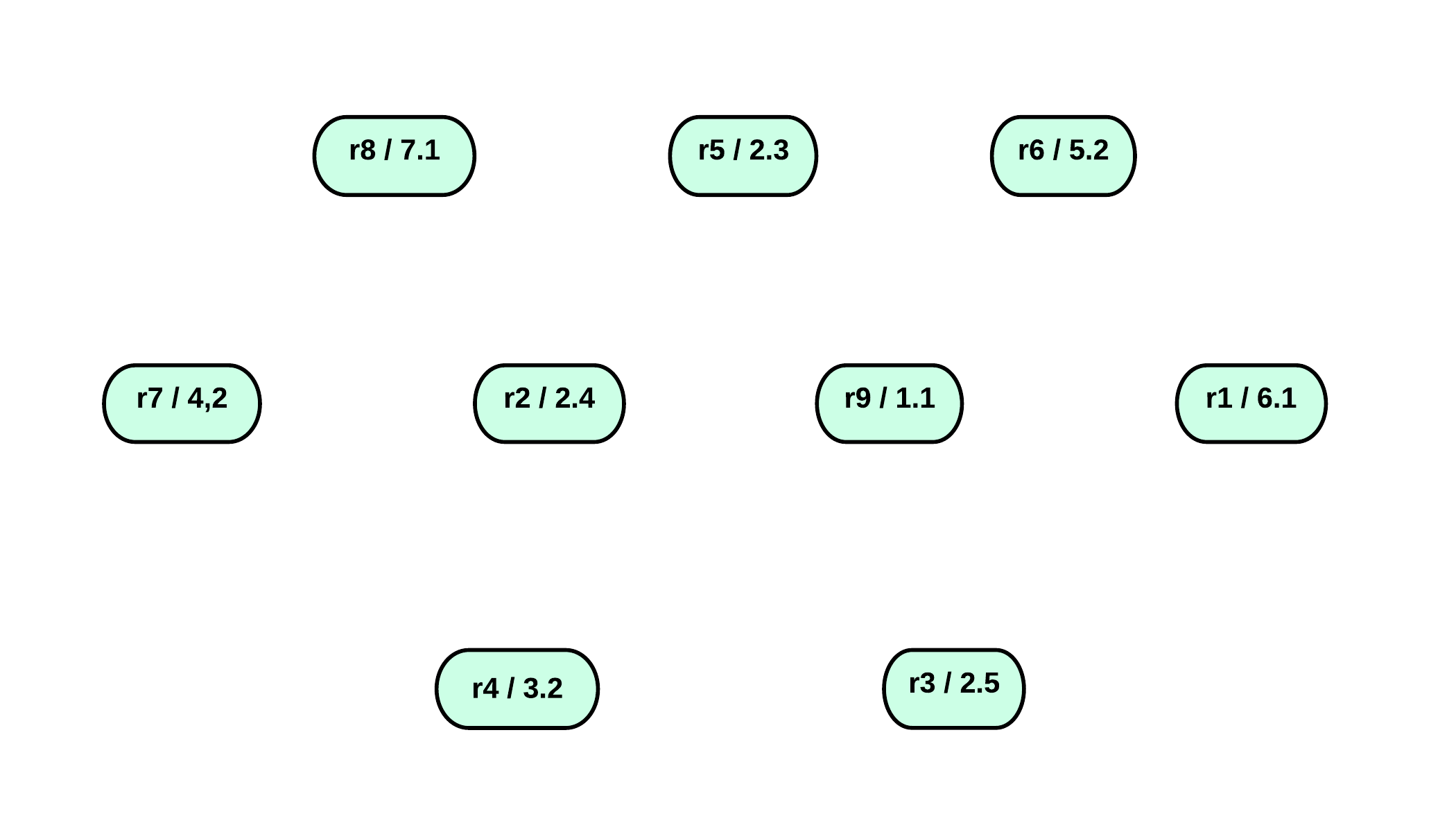
log3\_9:1.1.0.0/16 60.041 0.250 1 valid 1.1.0.1

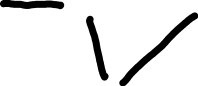
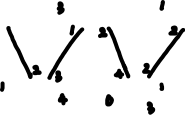
log3\_9:1.1.0.0/16 80.070 0.161 1 valid 1.1.0.1

log3\_9:1.1.0.0/16 110.424 0.361 1 valid 1.1.0.1

log3\_9:1.1.0.0/16 120.072 0.281 1 valid 1.1.0.1

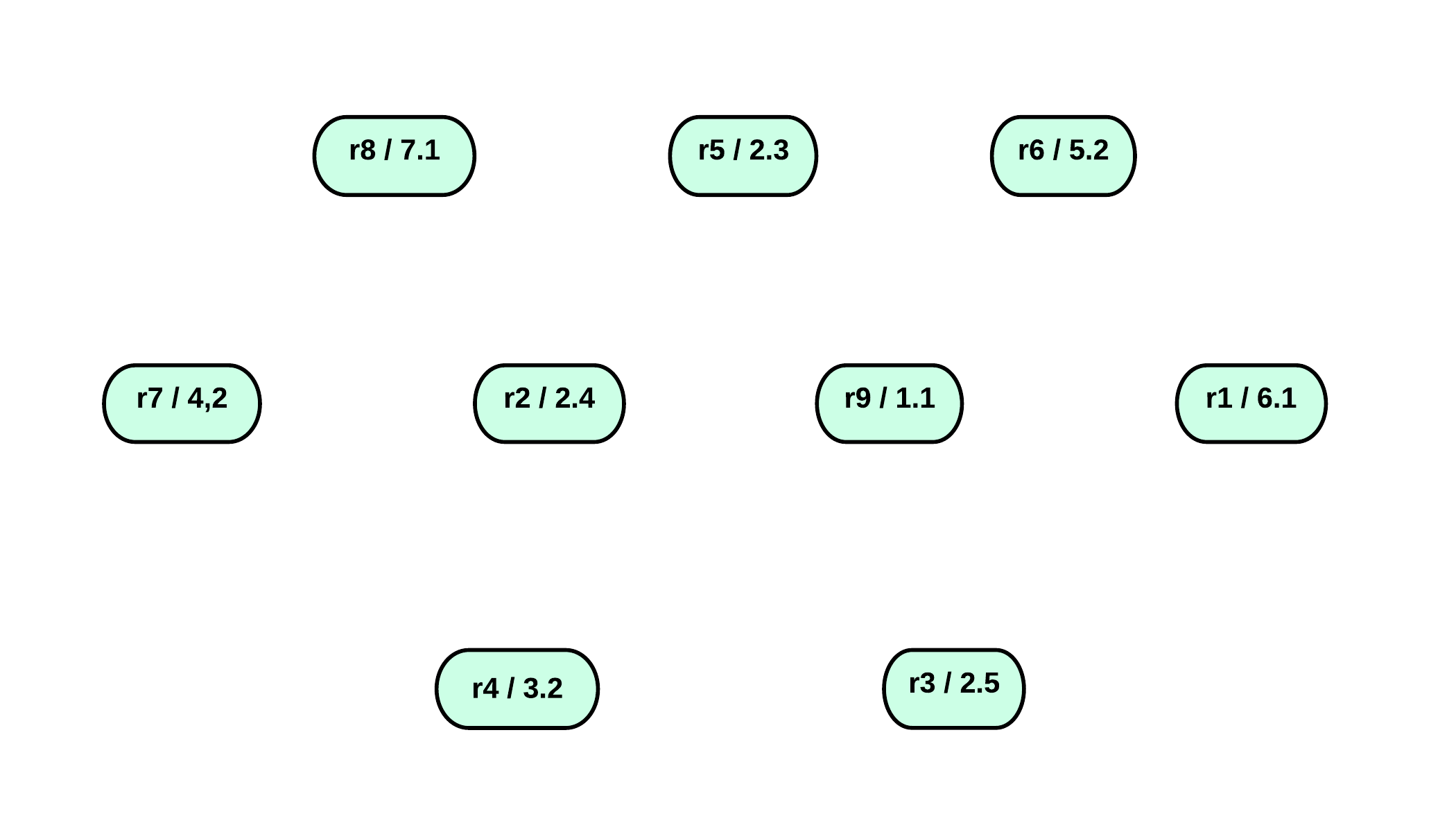
Paste a copy of your network graph below and highlight the shortest path tree defined by the routes going to *r*1 at time 15, by making the links heavy weight.

**



Find a time when the shortest path tree to *r*1 differs from the one at time 15. Paste another copy of your network graph below and highlight the links in the shortest path tree at that time. During what time period is this shortest path tree used?

*The shortest path tree shown below is used from time \_\_\_30\_\_\_ to time \_\_\_40\_\_\_.*

**

