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

Lab 5 Report

##### *Your name: Bingkun Guo, Jinyang Guo 11/21/2013*

***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.\*;

/\*\*

\* Name: Bingkun Guo, Jinyang Guo

\* Date: 11/21/2013

\*

\* 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.**

**\***

**\* It receives packets from the the Substrate and forwards them either**

**\* to applicaion layer or to the the Router according to the packet type.**

**\***

**\* It also receives payloads from the applicaiton layer and insert it**

**\* into pakcets and then sends to the substrate layer according to the**

**\* longest matching prefix for the destination address in the packet.**

**\***

**\* It also sends packets from the Routers to a specified link.**

**\*/**

**public void run() {**

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

**while (!quit) {**

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

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

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

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

**inPair.left.ttl--;**

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

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

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

**if (inPair.left.protocol == 1) {**

**try {**

**toSnk.put(inPair.left);**

**} catch (Exception e) {**

**System.out.println("Error: " + e);**

**}**

**} else {**

**try {**

**toRtr.put(inPair);**

**} catch (Exception e) {**

**System.out.println("Error: " + e);**

**}**

**}**

**} else if (inPair.left.ttl > 0) {**

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

**int lnkNum = lookup(inPair.left.destAdr);**

**if (lnkNum != -1 && sub.ready(lnkNum)) {**

**sub.send(inPair.left, lnkNum);**

**}**

**}**

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

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

**// send it to the Substrate**

**Pair<Packet, Integer> outPair = fromRtr.peek();**

**if (sub.ready(outPair.right)) {**

**try {**

**fromRtr.take();**

**} catch (Exception e) {**

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

**}**

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

**}**

**} else 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**

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

**//printTable();**

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

**if (sub.ready(lnkNum)) {**

**try {**

**fromSrc.take();**

**} catch (Exception e) {**

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

**}**

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

**}**

**} else {**

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

**try {**

**Thread.sleep(1);**

**} catch(Exception 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; otherwise add an entry**

**boolean haveMatch = false;**

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

**if (rte.left.equals(nuPrefix)) {**

**haveMatch = true;**

**rte.right = nuLnk;**

**break;**

**}**

**}**

**if (!haveMatch) {**

**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 bestLink = -1;**

**int bestLeng = -1;**

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

**if (pair.left.matches(ip) && (pair.left.leng > bestLeng)) {**

**bestLink = pair.right;**

**bestLeng = pair.left.leng;**

**}**

**}**

**return bestLink;**

**}**

/\*\* 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);

}

}

/\*\* 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 next packet for router, including 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; }

}

***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.\*;

/\*\*

\* Name: Bingkun Guo, Jinyang Guo

\* Date: 11/21/2013

\*

\* Router module for an overlay router.

\*

\* The Router will use a path-vector style protocol, similar to what BGP does

\* except that the protocol will be an intra-domian protocol and will include

\* a path cost equal to the sum of the link cost. A path- vector consists of an

\* advertisement, a path cost an a list of IP addresses deifing the path to the

\* router that issued the advertisement.

\*

\* The router thread sends " hello packets" to its neighbors once every second,

\* the hello packets contain a timestamp that the Router uses to measure the

\* round trip delay for the link, and th link cost is then set to half the round

\* trip delay. The Router use the exponential weighted moving average method (with

\* a parameter a = 0.1) to smooth out variations in the individual measurements.

\* The link costs should be set based on the "smoothed" delay values. If a Router

\* fail to get response to three conscutive hello packets from a given neighbor,

\* it will change the status of that link to "failed". It countinues to send hello

\* packets to failed links, and restores the link as soon as it gets a response.

\*

\* The payload of routing packets are formatted as ASCII text. A hello packet

\* is simply shown as below

\*

\* RPv0

\* type: hello

\* timestamp: 123.456

\*

\* The reply to a hello packet is shown as below

\*

\* RPv0

\* type: hello2u

\* timestamp: 123.456

\*

\* the timestamp in the hello packet is the time in seconds from an arbitrary

\* starting point at the sending router. Timestamp are echo back in replies

\* to hello packets

\*

\* The other type of packet used by the routing protocol is a route advertisement

\* packet. An example is shown below.

\*

\* RPv0

\* type: advert

\* pathvec: 1.5.0.0/16 345.678 .346 1.2.0.1 1.2.3.4 1.5.4.3

\*

\* Each path vector starts with advertised prefix, followed by a timestamp for

\* the vector and the cost of the path(in seconds). The remainder of the path

\* vector is a list of the IP addresses of the routes along the path, ending

\* with the router theat originated this advertisement (1.5.4.3 in the example).

\* The timestamp for a path vector represents the time at whic the advertising

\* router first sent this path vector. It is forwarded without change by other

\* routers and used by routers when deciding how to update their routing table

\* entries.

\*

\* Each Routers sends an advertisement for each of its own prefixes to each of

\* its neighbors periodically(every 10 seconds). The path vectors for these

\* advertisements will have a cost of zero and a path consisting only the IP

\* address of the sending Router.

\*

\* When a Router receives an advertisement from one of its neighbors, it first

\* checks to see if its own IP address appears in the path vector. If so, it

\* just discards this advertisement. Otherwise, it decides whether or not to

\* update its routing table, based on the contents of the path vector. Each

\* entry path vector contains the following fields.

\*

\* prefix, timestamp, cost, path, outputlink

\*

\* The prefix is an IP address prefix for the subnet that this route tell us

\* how to reach. The timestamp is the timestamp of the most recent advertisement

\* packet that caused an update to this routing table entry. The cost is the

\* sum of the link costs in the path to the destination subnet. The path is a

\* list of IP address strings defining the path to the destination subnet.

\* The output link is the link used to reach the first router on the path.

\*

\* When a Router received an advertisement, it affects the routing table.

\*

\* First, if the routing table has no entry for the subnet whose prefix is

\* specified in the advertisement, then a new route is added to the table

\* based on the information in the advertisement. The cost field of the routing

\* table entry is obtained by adding the cost in the received advertisement to

\* the of the link on which the advertisement was received.

\*

\* Second, if the routing table already had an entry for the subnet whose prefix

\* is specified in the advertisement then the program may need to modify the

\* existing route. The program modifies the routing table according to

\* following subcase

\*

\* If the advertisement arrived on a link that is currently disabled, then

\* the new advertisement is ignored

\*

\* Otherwise, if the new advertisement uses the same path as the current routing

\* table entry, then the program updates the timestamp field and the cost of the

\* existing routing entry, based on the information in the advertisement.

\*

\* Otherwise, the new advertisement defines a new route that uses a different

\* path than the old ones, We'll update the current entry based on this new route

\* if any of the following three conditions is true.

\*

\* the cost of the new route is at least 10% smaller than a cost of

\* the current route

\*

\* the new route is at least 20 seconds newer than the old route (as defined

\* by the timestamp of the current route and timestamp of the advertisemen)

\*

\* the current route uses a lnk that is disabled.

\*

\* If a new route is added to the routing table, or if the link field of an

\* existing entry is changed, the nthe Router should also change the corresponding

\* entry in the Forwarder's internal table.

\*

\* whenever the debuggingis enabled and a received advertisement either

\* cause us to add a route to the table or modify the path, contents of the routing

\* table should be printed

\*

\* Whenever a received advertisement cause any change at all to the routing

\* table, then that advertisement should be extended and forwarded to all the

\* neighboring routers (with the exception of the router does two things.

\* First, it modifies the cost field in the advertisement to include the

\* cost of the link on which it received the advertisement. Second, it adds its

\* own IP address to the front of the path portion of the advertisement.

\*

\*/

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

}

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

/\*\* 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) {

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

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

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

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.**

**\***

**\* Every second, we send a hello packet to the neighbors and every 10**

**\* seconds we advertise our prefixes to the neighbors.**

**\* when we get a hello, we echo it back and when we get a reply to**

**\* our own hello, we use it to update our link costs**

**\***

**\* When we get an advertisement from neigbor, we first check if we'are in**

**\* the path and discard it, If not, we update our own routing state and**

**\* forward, as appropriate.**

**\***

**\*/**

**public void run() {**

**double t0 = System.nanoTime()/sec;**

**now = 0;**

**double helloTime = 0;**

**double adTime = 0;**

**while (!quit) {**

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

**if (now > helloTime + 1) {**

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

**sendHellos();**

**helloTime = now;**

**} else if (now > adTime + 10) {**

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

**sendAdverts();**

**adTime = now;**

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

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

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

**handleIncoming();**

**} else {**

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

**try {**

**Thread.sleep(1);**

**} catch(Exception e) {**

**System.exit(1);**

**}**

**}**

**}**

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

**"peerIp count avgCost minCost maxCost\n");**

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

**if (lnk.count == 0) continue;**

**s += String.format("%s %d %.3f %.3f %.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 rte : rteTbl) {**

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

**}**

**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; just add it**

**\*/**

**private void addRoute(Route rte) {**

**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 in any way, 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 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) {**

**// TODO**

**if (lnkVec.get(nuRte.outLink).helloState == 0) {**

**return false;**

**} else if (rte.path.equals(nuRte.path)) {**

**rte.timestamp = nuRte.timestamp;**

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

**return true;**

**} else if ( (nuRte.cost < 0.9 \* rte.cost)**

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

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

**rte.timestamp = nuRte.timestamp;**

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

**rte.path = nuRte.path;**

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

**return true;**

**}**

**return false;**

**}**

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

**\***

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

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

**\* the link is considered down, so we increase the cost of all**

**\* routes using that link.**

**\*/**

**public void sendHellos() {**

**int lnk = 0;**

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

**// TODO**

**if (!lnkInfo.gotReply) {**

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

**// helloState if it's not already 0**

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

**lnkInfo.helloState--;**

**}**

**}**

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

**lnkInfo.gotReply = false;**

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

**p.protocol = 2;**

**p.ttl = 100;**

**p.srcAdr = myIp;**

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

**p.payload = "RPv0\ntype: hello\n"**

**+ "timestamp: " + Double.toString(now) + "\n";**

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

**lnk++;**

**}**

**}**

**/\*\* Send initial advertisement to each of our neighbors. \*/**

**public void sendAdverts() {**

**// TODO**

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

**int lnk = 0;**

**for (NborInfo nbr : nborList) {**

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

**p.protocol = 2;**

**p.ttl = 100;**

**p.srcAdr = myIp;**

**p.destAdr = nbr.ip;**

**p.payload = "RPv0\ntype: advert\n"**

**+ "pathvec: " + pfx.toString()**

**+ " " + now + " 0 "**

**+ myIpString + "\n";**

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

**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();**

**// TODO**

**if (type.equals("advert")) {**

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

**handleAdvert(lines, lnk);**

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

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

**Packet reply = new Packet();**

**reply.protocol = 2;**

**reply.ttl = 100;**

**reply.srcAdr = myIp;**

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

**reply.payload = "RPv0\ntype: hello2u\n"**

**+ lines[2] + "\n";**

**if (fwdr.ready4pkt()) {**

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

**}**

**} else if (type.equals("hello2u")) {**

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

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

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

**// weighted moving average method**

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

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

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

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

**double halfRtt = (now - timestamp) / 2;**

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

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

**lnkVec.get(lnk).cost = 0.1 \* halfRtt**

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

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

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

**lnkVec.get(lnk).minCost**

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

**lnkVec.get(lnk).maxCost**

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

**}**

**}**

**/\*\* 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**

**// TODO**

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

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

**int ip;**

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

**double timestamp;**

**double cost;**

**int leng;**

**boolean loopFound = false;**

**if (!chunks[0].trim().equals("pathvec")) {**

**System.out.println("wrong name!!!");**

**return;**

**}**

**chunks = chunks[1].trim().split("\\s+");**

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

**cost = Double.parseDouble(chunks[2]);**

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

**path.add(Util.string2ip(chunks[i]));**

**}**

**Prefix newPfx = new Prefix(chunks[0].trim());**

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

**for (int adr : path) {**

**if (adr == myIp) {**

**loopFound = true;**

**break;**

**}**

**}**

**if (loopFound)**

**return;**

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

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

**Route newRoute = new Route();**

**newRoute.pfx = newPfx;**

**newRoute.timestamp = timestamp;**

**newRoute.cost = cost + lnkVec.get(lnk).cost;**

**newRoute.path = path;**

**newRoute.outLink = lnk;**

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

**// and update as appropriate;**

**Route rte = lookupRoute(newRoute.pfx);**

**boolean rteTblChanged = false;**

**if (rte == null) {**

**addRoute(newRoute);**

**if (debug > 0) {**

**printTable();**

**}**

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

**rteTblChanged = true;**

**} else {**

**boolean pathChanged = !newRoute.path.equals(rte.path);**

**boolean linkChanged = (newRoute.outLink != rte.outLink);**

**if (updateRoute(rte, newRoute)) {**

**rteTblChanged = true;**

**// whenever an update**

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

**if (debug > 0 && pathChanged) {**

**printTable();**

**}**

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

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

**if (linkChanged) {**

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

**}**

**}**

**}**

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

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

**if (rteTblChanged) {**

**StringBuilder pathStr = new StringBuilder();**

**for (int adr : newRoute.path) {**

**pathStr.append(" ");**

**pathStr.append(Util.ip2string(adr));**

**}**

**String payload = "RPv0\ntype: advert\n"**

**+ "pathvec: " + newRoute.pfx.toString()**

**+ " " + Double.toString(newRoute.timestamp)**

**+ " " + Double.toString(newRoute.cost)**

**+ " " + myIpString**

**+ pathStr.toString() + "\n";**

**int outputLnk = 0;**

**for (NborInfo nbr : nborList) {**

**if (outputLnk != lnk) {**

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

**p.protocol = 2;**

**p.ttl = 100;**

**p.srcAdr = myIp;**

**p.destAdr = nbr.ip;**

**p.payload = payload;**

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

**}**

**outputLnk++;**

**}**

**}**

**}**

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

public void printTable() {

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

+ "prefix timestamp cost path link\n",now);

for (Route rte : rteTbl) {

s += String.format("%s %.3f %.3f",

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

for (int r :rte.path)

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

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

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

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

s += "\n";

}

System.out.println(s);

}

}

***Part C****.* 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.

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*. Label the inter-router links with their assigned link costs.

*your diagram here*

1. (5 points) Run *script1* in the *net1* sub-directory by typing

./script1 .333 20 static

Paste a copy of the output below.

your output here

guobingkun@onlusr:~/473/lab5/net1$ ./script1 .333 20 static

delta= .333 runlength= 20 static

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

Final Report

Routing table (28.282)

prefix timestamp cost path link

1.3.0.0/16 20.001 0.012 1.3.0.1 1

1.2.0.0/16 20.001 0.023 1.2.0.1 0

Forwarding table (28.286)

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 0.021 0.043

1.3.0.1 25 0.013 0.011 0.021

SrcSnk statistics

destIp count avgDelay minDelay maxDelay

1.3.0.1 27 0.026 0.013 0.047

1.2.0.1 34 0.032 0.023 0.066

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

Final Report

Routing table (28.298)

prefix timestamp cost path link

1.1.0.0/16 20.001 0.022 1.1.0.1 1

1.3.0.0/16 20.001 0.034 1.1.0.1 1.3.0.1 1

Forwarding table (28.298)

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 0.052 0.051 0.054

1.1.0.1 25 0.022 0.021 0.023

SrcSnk statistics

destIp count avgDelay minDelay maxDelay

1.3.0.1 27 0.038 0.033 0.045

1.1.0.1 34 0.031 0.023 0.056

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

Final Report

Routing table (28.329)

prefix timestamp cost path link

1.1.0.0/16 20.001 0.012 1.1.0.1 0

1.2.0.0/16 20.001 0.035 1.1.0.1 1.2.0.1 0

Forwarding table (28.328)

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 25 0.013 0.012 0.017

1.2.0.1 25 0.053 0.051 0.059

SrcSnk statistics

destIp count avgDelay minDelay maxDelay

1.1.0.1 24 0.025 0.013 0.045

1.2.0.1 37 0.039 0.033 0.061

1. (5 points) 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: shortest path = r1 -> r2 total cost = 0.02*

*r1 to r3: shortest path = r1 -> r3 total cost = 0.01*

*r2 to r3: shortest path = r2 -> r1 -> r3 total cost = 0.03*

*The forwarding tables and routing tables are consistent with each other since they have the same link and the path for the same destination prefix.*

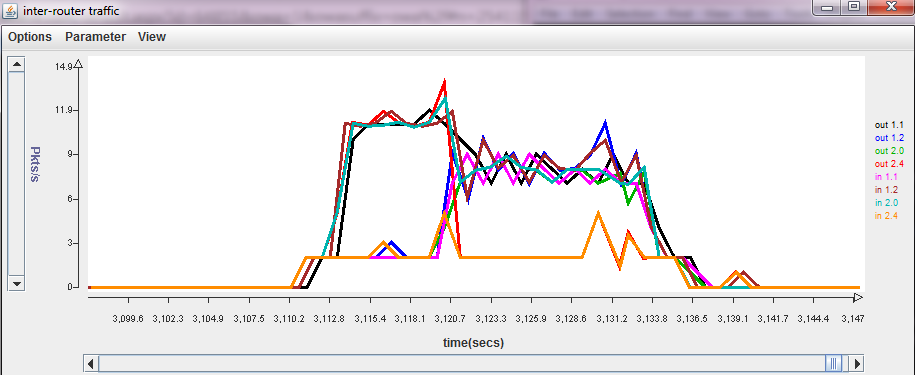
*The tables for r1 are consistent with its shortest path, since r1 directly goes to r2 and r3 without intermediate route. The costs*

*The tables for r2 are consistent with its shortest path, since the routes from r2 to r1 and r2 to r3 both go to r1 first.*

*The tables for r3 are consistent with its shortest path, since the routes from r3 to r1 and r3 to r1 both go to r1 first.*

1. (5 points) Paste a screenshot of the monitoring window from your *script1* run below.

*your screenshot here*

**

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?

*For the first 10 seconds, all the packets are forwarded to the default link 0. That means r1 forward all packets to r2, r2 forwards all its packets to r3 and r3 forwards all its packets to r1.*

*After about 10 seconds, each router starts distributing advertisements, thus the path would be updated and the traffic changes.*

1. (5 points) 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?

*During the first part of the run, r1 sends a hello packet every second to r2 and r1 replies the hello packet sent from r2, so that’s 2 packets per second. Also, since the delta is 0.333, r1 sends nearly 3 packets per second to r2, and also 3 packets to r3 during the first part of the run. Finally, the SrcSnk at r3 also sends 3 packets to r2 at the first part of the run. Thus the total would be 2 + 6 + 3= 11 packets per second.*

1. (5 points) 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

*During the second part of the run, r1 sends 2 hello(including reply) packets to r2, r1 sends 3 SrcSnk packets to r2, r3 also sends 3 SrcSnk packets r2, and these 3 packets will pass over this link, thus the total would be 2 + 3 + 3 = 8 packets.*

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

./script1 .333 20 static debugg

and paste a copy of the resulting *log2* 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.

your output here

/192.168.7.1:31313 sending to /192.168.4.2:31313 at 10.023

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

RPv0

type: advert

pathvec: 1.2.0.0/16 10.000677644000461 0 1.2.0.1

/192.168.7.1:31313 received from /192.168.4.2:31313 at 10.035

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

RPv0

type: advert

pathvec: 1.1.0.0/16 10.000150208999912 0 1.1.0.1

**Above ad contains a new prefix, so it gets added to the routing table.**

/192.168.7.1:31313 sending to /192.168.2.4:31313 at 10.056

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

RPv0

type: advert

pathvec: 1.2.0.0/16 10.000677644000461 0 1.2.0.1

/192.168.7.1:31313 received from /192.168.2.4:31313 at 10.049

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

RPv0

type: advert

pathvec: 1.3.0.0/16 10.000779504000093 0 1.3.0.1

**Above ad contains a new prefix, so it gets added to the routing table.**

Routing table (10.048)

prefix timestamp cost path link

1.1.0.0/16 10.000 0.021 1.1.0.1 1

Forwarding table (10.062)

0.0.0.0/0 0

1.1.0.0/16 1

Routing table (10.064)

prefix timestamp cost path link

1.1.0.0/16 10.000 0.021 1.1.0.1 1

1.3.0.0/16 10.001 0.051 1.3.0.1 0

Forwarding table (10.069)

0.0.0.0/0 0

1.1.0.0/16 1

1.3.0.0/16 0

/192.168.7.1:31313 received from /192.168.4.2:31313 at 10.062

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

RPv0

type: advert

pathvec: 1.3.0.0/16 10.000779504000093 0.011402300282522398 1.1.0.1 1.3.0.1

**This ad contains an existing prefix in the routing table. And the cost**

**is less than 90% of the existing cost, so the routing table will be updated.**

/192.168.7.1:31313 sending to /192.168.4.2:31313 at 10.093

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

RPv0

type: advert

pathvec: 1.3.0.0/16 10.000779504000093 0.051099134884868405 1.2.0.1 1.3.0.1

Routing table (10.090)

prefix timestamp cost path link

1.1.0.0/16 10.000 0.021 1.1.0.1 1

1.3.0.0/16 10.001 0.033 1.1.0.1 1.3.0.1 1

Forwarding table (10.098)

0.0.0.0/0 0

1.1.0.0/16 1

1.3.0.0/16 1

/192.168.7.1:31313 received from /192.168.2.4:31313 at 10.091

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

RPv0

type: advert

pathvec: 1.1.0.0/16 10.000150208999912 0.011040765590872945 1.3.0.1 1.1.0.1

**This ad contains an existing prefix, and it has a higher total cost than the existing one, so the routing table will not be updated.**

/192.168.7.1:31313 sending to /192.168.2.4:31313 at 10.115

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

RPv0

type: advert

pathvec: 1.1.0.0/16 10.000150208999912 0.021412172916198725 1.2.0.1 1.1.0.1

/192.168.7.1:31313 received from /192.168.2.4:31313 at 10.120

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

RPv0

type: advert

pathvec: 1.2.0.0/16 10.000677644000461 0.03436437975568608 1.3.0.1 1.1.0.1 1.2.0.1

**This ad will not change the routing table, since r2 has 0 cost to itself.**

/192.168.7.1:31313 sending to /192.168.2.4:31313 at 10.152

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

RPv0

type: advert

pathvec: 1.3.0.0/16 10.000779504000093 0.03281447319872112 1.2.0.1 1.1.0.1 1.3.0.1

/192.168.7.1:31313 sending to /192.168.4.2:31313 at 20.024

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

RPv0

type: advert

pathvec: 1.2.0.0/16 20.00145780799994 0 1.2.0.1

/192.168.7.1:31313 received from /192.168.4.2:31313 at 20.019

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

RPv0

type: advert

pathvec: 1.1.0.0/16 20.0009122499996 0 1.1.0.1

/192.168.7.1:31313 received from /192.168.4.2:31313 at 20.032

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

RPv0

type: advert

pathvec: 1.3.0.0/16 20.001728181000544 0.012761337492010534 1.1.0.1 1.3.0.1

**Repeating ad, update timestamp and cost**

/192.168.7.1:31313 sending to /192.168.2.4:31313 at 20.054

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

RPv0

type: advert

pathvec: 1.2.0.0/16 20.00145780799994 0 1.2.0.1

/192.168.7.1:31313 received from /192.168.2.4:31313 at 20.055

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

RPv0

type: advert

pathvec: 1.3.0.0/16 20.001728181000544 0 1.3.0.1

**All the subsequent ads are repeats of above ads, they will not change the forwarding table and they might only update timestamps and costs.**

/192.168.7.1:31313 received from /192.168.2.4:31313 at 20.067

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

RPv0

type: advert

pathvec: 1.1.0.0/16 20.0009122499996 0.012581603562662552 1.3.0.1 1.1.0.1

/192.168.7.1:31313 sending to /192.168.2.4:31313 at 20.084

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

RPv0

type: advert

pathvec: 1.1.0.0/16 20.0009122499996 0.022587045369771755 1.2.0.1 1.1.0.1

/192.168.7.1:31313 received from /192.168.2.4:31313 at 20.071

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

RPv0

type: advert

pathvec: 1.2.0.0/16 20.00145780799994 0.0349849236470495 1.3.0.1 1.1.0.1 1.2.0.1

/192.168.7.1:31313 sending to /192.168.2.4:31313 at 20.103

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

RPv0

type: advert

pathvec: 1.3.0.0/16 20.001728181000544 0.03534838286178229 1.2.0.1 1.1.0.1 1.3.0.1

Final Report

Routing table (28.237)

prefix timestamp cost path link

1.1.0.0/16 20.001 0.023 1.1.0.1 1

1.3.0.0/16 20.002 0.035 1.1.0.1 1.3.0.1 1

Forwarding table (28.237)

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 0.052 0.051 0.057

1.1.0.1 25 0.023 0.021 0.047

SrcSnk statistics

destIp count avgDelay minDelay maxDelay

1.3.0.1 30 0.037 0.032 0.048

1.1.0.1 31 0.028 0.021 0.043

1. (15 points) In this part, you will disable and re-enable one of the links while *script1* 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.

Now, run *script1* again (with the filter turned off) by typing

./script1 .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 *log2* file from your run below and add comments in bold, explaining all changes to the routing table at *r1*.

Routing table (10.021)

prefix timestamp cost path link

1.3.0.0/16 10.001 0.051 1.3.0.1 0

Forwarding table (10.053)

0.0.0.0/0 0

1.3.0.0/16 0

Routing table (10.056)

prefix timestamp cost path link

1.3.0.0/16 10.001 0.033 1.1.0.1 1.3.0.1 1

Forwarding table (10.059)

0.0.0.0/0 0

1.3.0.0/16 1

Routing table (10.062)

prefix timestamp cost path link

1.3.0.0/16 10.001 0.033 1.1.0.1 1.3.0.1 1

1.1.0.0/16 10.032 0.021 1.1.0.1 1

Forwarding table (10.066)

0.0.0.0/0 0

1.3.0.0/16 1

1.1.0.0/16 1

**The above changes to the routing table are caused by the first set of advertisements sent at about 10 seconds.**

Routing table (40.021)

prefix timestamp cost path link

1.3.0.0/16 40.003 0.052 1.3.0.1 0

1.1.0.0/16 30.032 0.022 1.1.0.1 1 \*\* disabled link

Forwarding table (40.026)

0.0.0.0/0 0

1.3.0.0/16 0

1.1.0.0/16 1

**Because the link at port 1.1 is disabled, so the path for r2 to route r1 will change, and we are now using link 0 to reach r3.**

Routing table (40.080)

prefix timestamp cost path link

1.3.0.0/16 40.003 0.052 1.3.0.1 0

1.1.0.0/16 40.033 0.064 1.3.0.1 1.1.0.1 0

Forwarding table (40.085)

0.0.0.0/0 0

1.3.0.0/16 0

1.1.0.0/16 0

**Now the path from r2 to r1 changes, since link 1 is disabled, r2’s packets have to first go to r3 from link 0, then reach r1 from r3.**

Routing table (70.012)

prefix timestamp cost path link

1.3.0.0/16 70.005 0.034 1.1.0.1 1.3.0.1 1

1.1.0.0/16 60.034 0.064 1.3.0.1 1.1.0.1 0

Forwarding table (70.016)

0.0.0.0/0 0

1.3.0.0/16 1

1.1.0.0/16 0

**After we re-enabled link 1, r2 goes to r3 through link 1, because it’s shorter**

Routing table (70.042)

prefix timestamp cost path link

1.3.0.0/16 70.005 0.034 1.1.0.1 1.3.0.1 1

1.1.0.0/16 70.035 0.022 1.1.0.1 1

Forwarding table (70.046)

0.0.0.0/0 0

1.3.0.0/16 1

1.1.0.0/16 1

**Since link 1 is alive again, r2 can go to r1 directly from link 1.**

Final Report

Routing table (108.174)

prefix timestamp cost path link

1.3.0.0/16 100.007 0.034 1.1.0.1 1.3.0.1 1

1.1.0.0/16 100.037 0.022 1.1.0.1 1

Forwarding table (108.174)

0.0.0.0/0 0

1.3.0.0/16 1

1.1.0.0/16 1

Router link cost statistics

peerIp count avgCost minCost maxCost

1.3.0.1 105 0.052 0.049 0.060

1.1.0.1 74 0.022 0.019 0.032

SrcSnk statistics

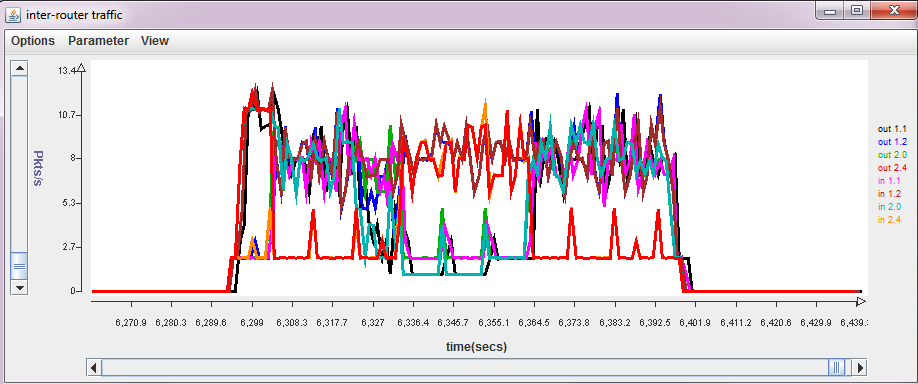
destIp count avgDelay minDelay maxDelay

1.3.0.1 129 0.040 0.031 0.053

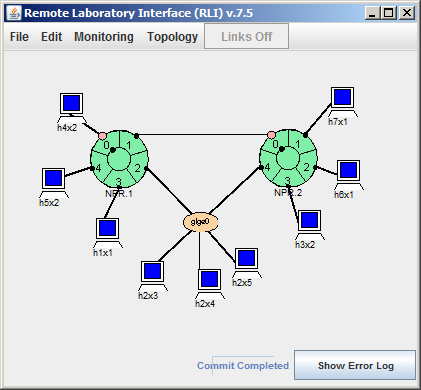
1.1.0.1 140 0.038 0.021 0.064

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

*your screenshot here*

**

Explain the packet rates leaving ONL router port 1.1 and 2.0 during the period when the link is disabled. Justify the specific numerical values observed.

*We could see from the routing graph, when the link between 1.1 and 2.0(the link between r1 and r2) is disabled, all SrcSnk packets packets from r1 would have to reach r2 via r3, and all SrcSnk packets from r2 to r3 would no longer pass over the disabled link, since we know SrcSnk sends 3 packets / second, so the disabled link will decrease 6 SrcSnk packets/second in each direction. Consequently, we could see the link between r2 and r3 (out from 2.4) increased about 6 packets per second, because the SrcSnk packets went between r1 and r2, and between r2 and r3 started passing over this link. The out 1.2 which reflects the link from r1 to r3 didn’t change eventually, because although it added a 3 packets from ScrSnk r1 to r3 , it also minus 3 packets from ScrSnk r2 to r3 via r1 which is now pass through the r2 to r3 directly.*

***Part D***. In this part, you will be using the configuration in the *net2* subdirectory.

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 as 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 .05

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..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.

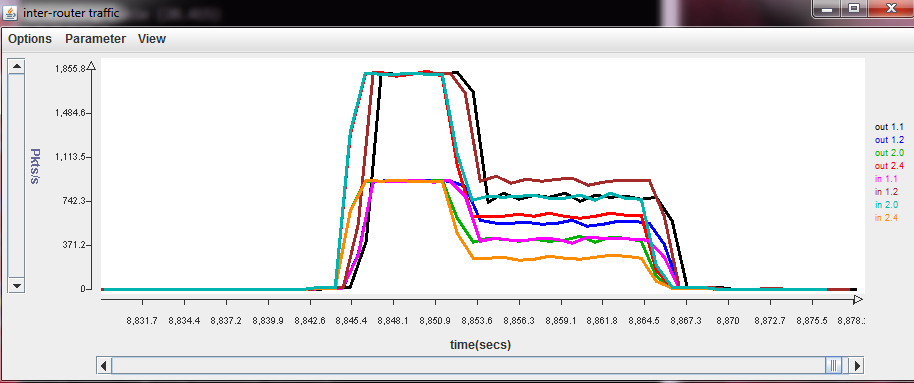
*your diagram here*

1. (10 points) Run the provided *script2* by typing

script2 .01 20 static

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

*your screenshot here*

**

Paste the portion of the output from *log2* showing the final routing table at *r*2.

Routing table (28.321)

prefix timestamp cost path link

1.4.0.0/16 20.000 0.032 1.4.0.1 0

1.8.0.0/16 20.000 0.044 1.5.0.1 1.8.0.1 3

1.7.0.0/16 20.001 0.062 1.7.0.1 1

1.5.0.0/16 20.009 0.013 1.5.0.1 3

1.6.0.0/16 20.002 0.050 1.9.0.1 1.6.0.1 4

1.9.0.0/16 20.028 0.023 1.9.0.1 4

1.1.0.0/16 20.001 0.066 1.5.0.1 1.6.0.1 1.1.0.1 3

1.3.0.0/16 20.001 0.046 1.9.0.1 1.3.0.1 4

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

*It is almost consistent with the shortest path tree in my network graph. Although some paths have slightly more costs than the network graph. Due to the overhead from application level, the costs might not be necessarily consistent with the ones in the theoretical graph, and also we set a condition that a new path’s cost must be less than 90% cost of the old path in order to be updated in the routing table, so if the cost difference is really small, the theoretical shortest path might not be used.*

1. (10 points) Run *script2* by typing

script2 .01 20 static debugg

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).

/192.168.2.3:31313 received from /192.168.7.1:31313 at 9.777

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

RPv0

type: advert

pathvec: 1.7.0.0/16 10.00075306399998 0.02122251865849751 1.8.0.1 1.7.0.1

/192.168.2.3:31313 received from /192.168.2.4:31313 at 9.812

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

RPv0

type: advert

pathvec: 1.7.0.0/16 10.00075306399998 0.06125852086191749 1.2.0.1 1.7.0.1

/192.168.2.3:31313 received from /192.168.1.1:31313 at 9.868

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

RPv0

type: advert

pathvec: 1.7.0.0/16 10.00075306399998 0.08278175261508376 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 ads and the others do not? Do your best to explain your observations based on the delays that ads will experience as they pass through the network.

*r8, r2, r9 send r5 advertisements for 1.7.0.0/16.*

*r5 has four neighbors which is r8,r2,r9 and r6. Note, in the program, the routers send the update routing information to their neighbor except the one from which they got the particular routing message.*

*r8, r2, r9 send r5 advertisements for 1.7.0.0/16, because when this neighbor got their first advertisement not from r5, so they need to update the routing table and broadcasted this change to their neighbors including r5 (not to the link they got this advertisement). On the other hand r6 got its first 1.7.0.0/16 advertisement from r5 and added a route to routing table, it then broadcasted this change to its neighbors except r5 (the link it got the routing message). Because the route via r5 is the shortest cost route for r6 to r7. When r6 got other advertisements for 1.7.0.0/16 from other nodes , its cost would be bigger than this one. So there is no change in r6’s routing table, so r5 would not get any advertisement of 1.7.0.0/16 from r6.*

*We can confirm our assumption by looking at log6*

*The first advertisement for 1.7.0.0/16 r6 received is from r5:*

*/192.168.5.2:31313 received from /192.168.2.3:31313 at 9.863*

*protocol=2 ttl=100 srcAdr=1.5.0.1 destAdr=1.6.0.1*

*RPv0*

*type: advert*

*pathvec: 1.7.0.0/16 10.00075306399998 0.05270317286586919 1.5.0.1 1.8.0.1 1.7.0.1*

*And the routing table entry for r6 to r7 is as follows (note that it will pass r5, it also is the shortest one, never changed during the whole time after it was added according to log6).*

*Routing table (9.888)*

*prefix timestamp cost path link*

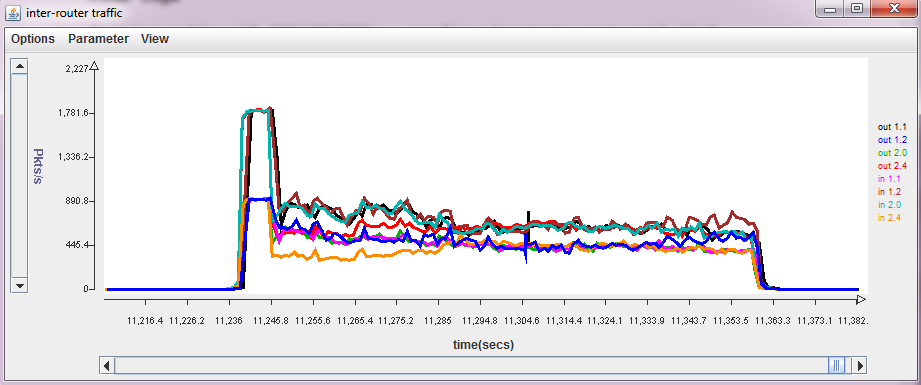
*1.7.0.0/16 10.001 0.094 1.5.0.1 1.8.0.1 1.7.0.1 1*

1. (10 points) Run *script2* by typing

script2 .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 arrow head at the right end of the horizontal axis.

*your screenshot here*

**

Type the command

grep "1.1.0.0.16....." log\*

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 paths defined by the route change, by making them bold.

log2:1.1.0.0/16 10.001 0.269 1.9.0.1 1.3.0.1 1.1.0.1 4

log2:1.1.0.0/16 10.001 0.230 1.5.0.1 1.6.0.1 1.1.0.1 3

log2:1.1.0.0/16 20.001 0.440 1.5.0.1 1.6.0.1 1.1.0.1 3

**log2:1.1.0.0/16 40.002 0.425 1.9.0.1 1.3.0.1 1.1.0.1 4**

log2:1.1.0.0/16 50.002 0.447 1.9.0.1 1.3.0.1 1.1.0.1 4

**log2:1.1.0.0/16 60.003 0.356 1.9.0.1 1.1.0.1 4**

log2:1.1.0.0/16 80.003 0.359 1.9.0.1 1.1.0.1 4

log2:1.1.0.0/16 120.006 0.383 1.9.0.1 1.1.0.1 4

log3:1.1.0.0/16 10.001 0.067 1.1.0.1 1

log3:1.1.0.0/16 20.001 0.130 1.1.0.1 1

log3:1.1.0.0/16 30.002 0.116 1.1.0.1 1

log3:1.1.0.0/16 60.003 0.125 1.1.0.1 1

log3:1.1.0.0/16 80.003 0.219 1.1.0.1 1

log3:1.1.0.0/16 90.004 0.250 1.1.0.1 1

log3:1.1.0.0/16 100.005 0.238 1.1.0.1 1

log3:1.1.0.0/16 110.006 0.188 1.1.0.1 1

log3:1.1.0.0/16 120.006 0.117 1.1.0.1 1

log4:1.1.0.0/16 10.001 0.343 1.2.0.1 1.9.0.1 1.3.0.1 1.1.0.1 1

log4:1.1.0.0/16 10.001 0.304 1.2.0.1 1.5.0.1 1.6.0.1 1.1.0.1 1

**log4:1.1.0.0/16 30.002 0.346 1.3.0.1 1.1.0.1 2**

log4:1.1.0.0/16 40.002 0.324 1.3.0.1 1.1.0.1 2

log4:1.1.0.0/16 70.003 0.426 1.3.0.1 1.1.0.1 2

log4:1.1.0.0/16 90.004 0.467 1.3.0.1 1.1.0.1 2

log4:1.1.0.0/16 100.005 0.560 1.3.0.1 1.1.0.1 2

log4:1.1.0.0/16 120.006 0.353 1.3.0.1 1.1.0.1 2

log4:1.1.0.0/16 120.006 0.353 1.3.0.1 1.1.0.1 2

log5:1.1.0.0/16 10.001 0.336 1.9.0.1 1.3.0.1 1.1.0.1 2

log5:1.1.0.0/16 10.001 0.132 1.6.0.1 1.1.0.1 0

log5:1.1.0.0/16 30.002 0.442 1.6.0.1 1.1.0.1 0

log5:1.1.0.0/16 50.002 0.373 1.6.0.1 1.1.0.1 0

log5:1.1.0.0/16 60.003 0.230 1.6.0.1 1.1.0.1 0

log5:1.1.0.0/16 70.003 0.138 1.6.0.1 1.1.0.1 0

log5:1.1.0.0/16 90.004 0.298 1.6.0.1 1.1.0.1 0

log5:1.1.0.0/16 120.006 0.298 1.6.0.1 1.1.0.1 0

log6:1.1.0.0/16 10.001 0.046 1.1.0.1 0

log6:1.1.0.0/16 30.002 0.204 1.1.0.1 0

log6:1.1.0.0/16 50.002 0.196 1.1.0.1 0

log6:1.1.0.0/16 60.003 0.123 1.1.0.1 0

log6:1.1.0.0/16 70.003 0.080 1.1.0.1 0

log6:1.1.0.0/16 90.004 0.165 1.1.0.1 0

log6:1.1.0.0/16 120.006 0.131 1.1.0.1 0

log7:1.1.0.0/16 10.001 0.478 1.4.0.1 1.2.0.1 1.9.0.1 1.3.0.1 1.1.0.1 2

log7:1.1.0.0/16 10.001 0.395 1.2.0.1 1.9.0.1 1.3.0.1 1.1.0.1 1

log7:1.1.0.0/16 10.001 0.312 1.8.0.1 1.5.0.1 1.6.0.1 1.1.0.1 0

log7:1.1.0.0/16 20.001 0.591 1.8.0.1 1.5.0.1 1.6.0.1 1.1.0.1 0

**log7:1.1.0.0/16 40.002 0.466 1.4.0.1 1.3.0.1 1.1.0.1 2**

**log7:1.1.0.0/16 70.003 0.473 1.8.0.1 1.5.0.1 1.6.0.1 1.1.0.1 0**

**log7:1.1.0.0/16 100.005 0.641 1.2.0.1 1.9.0.1 1.1.0.1 1**

log7:1.1.0.0/16 120.006 0.654 1.2.0.1 1.9.0.1 1.1.0.1 1

log8:1.1.0.0/16 10.001 0.545 1.7.0.1 1.4.0.1 1.2.0.1 1.9.0.1 1.3.0.1 1.1.0.1 1

log8:1.1.0.0/16 10.001 0.463 1.7.0.1 1.2.0.1 1.9.0.1 1.3.0.1 1.1.0.1 1

log8:1.1.0.0/16 10.001 0.249 1.5.0.1 1.6.0.1 1.1.0.1 0

log8:1.1.0.0/16 30.002 0.630 1.5.0.1 1.6.0.1 1.1.0.1 0

log8:1.1.0.0/16 70.003 0.344 1.5.0.1 1.6.0.1 1.1.0.1 0

log8:1.1.0.0/16 80.003 0.402 1.5.0.1 1.6.0.1 1.1.0.1 0

log8:1.1.0.0/16 90.004 0.525 1.5.0.1 1.6.0.1 1.1.0.1 0

log8:1.1.0.0/16 110.006 0.665 1.5.0.1 1.6.0.1 1.1.0.1 0

log8:1.1.0.0/16 120.006 0.584 1.5.0.1 1.6.0.1 1.1.0.1 0

log9:1.1.0.0/16 10.001 0.158 1.3.0.1 1.1.0.1 3

log9:1.1.0.0/16 40.002 0.250 1.3.0.1 1.1.0.1 3

**log9:1.1.0.0/16 60.003 0.180 1.1.0.1 1**

log9:1.1.0.0/16 80.003 0.216 1.1.0.1 1

log9:1.1.0.0/16 100.005 0.274 1.1.0.1 1

log9:1.1.0.0/16 120.006 0.195 1.1.0.1 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.

*your diagram here*

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 \_\_\_\_\_\_ to time \_\_\_\_\_\_.*

*your diagram here*