

Routing Protocols

CS 5565 Spring 2009, Project 2B

Due dates

Part 1 Wednesday, Apr 29, 11:59pm

Part 2 Wednesday, May 6, 11:59pm

This project is worth 150 points, 75 points for each part. You may form teams of up to two students for this project. You are not required to team with the same student as in past projects, but you may not switch partners after you have started this project.

Introduction

A network interconnecting multiple hosts and networks is a dynamic entity. Most real world networks offer multiple paths from a source to a destination. Each path from a source to a destination has a cost associated with it. The cost may represent financially tangible quantities such as a dollar amount to be paid for using the path, or it may be a performance cost, in that different paths offer differing levels of performance. In contrast to this complexity, applications such as ssh or ftp take a simple point-to-point view of the network, i.e., data sources assume that they are directly connected to the destination. Network layer protocols reconcile these views of the network by discovering the topology of the network and building forwarding tables that encapsulate paths to each destination.

In this project, you will implement the distance vector and link-state routing protocols that will run on top of a simulated network of unknown topology. The network setup consists of an arbitrary number of routers, each of which has an arbitrary number of ports connected to other routers. The routers are connected using 10Mbps network links with a one-way propagation delay of $50\mu\text{s}$. Each port on a router connected to another router has a dollar cost associated with it, which may change over time. Additionally, links may fail, forcing the routing protocol to find an alternate path through the network. Your implementation should maintain forwarding tables at each router, listing the routes to all other routers and the cost associated with each route. If there are multiple paths to the same destination, your routing protocol should choose the path with the minimum cumulative cost.

Rather than on an actual network, your implementation will run on a simulator that simulates the essential functionality of a network. It provides a functional interface to the network. The simulator uses the RPC infrastructure created in project 2A.

Discrete Event Simulation

Internally, the simulation engine uses a discrete-event approach to simulation. "Discrete" means that time does not run continuously, but rather according to

a virtual clock that is under the simulator's control. Time advances in discrete steps or ticks. The simulator runs at a time resolution of $1\mu\text{s}$, so 1 tick = $1\mu\text{s}$.

"Event" means that the simulator is driven by events that occur at some points in the virtual timeline. Examples of events the simulator supports include: the arrival of a packet at a node¹, a cost change of a link, a link failure, or the expiration of a (virtual) timer. For some events, the simulator will make an outbound call to the node involved in the event. Inbound calls may schedule events, which the simulator will execute once its virtual clock has progressed to the time at which the event is scheduled.

Simulation Timeline

A simulation follows the following timeline:

1. Each router is initialized by the simulator using *init_node()*. Your *init_node()* implementation should create the Inbound TCP connection as in project 2A. In addition, you now need to perform some activity specific to the simulation. In particular, a node instance learns its own node or router id, the total number of routers in the network (*nnodes*) and the number of ports per router (*nports*).
2. After creating the Inbound TCP stream, but before returning from *init_node()*, routers should call *get_cost()* to learn the costs of their associated ports. They can then transmit their connectivity and port costs to their neighbors (or to the entire network, depending on the routing protocol.) In addition, you may wish to set appropriate timers to monitor link cost change and link failure, as discussed below.
3. Simulation will progress as the simulation engine delivers messages to other nodes (reception of which may in turn cause new messages to be sent, etc.), or as timers expire. You should perform the activities of your routing protocol and compute or update local forwarding tables as costs change or links fail.
4. At the end of the simulation, the simulator will score each node. Since the simulator has global knowledge about the topology of the network, it can query each node and compare whether the node's locally computed forwarding table matches the global view. Specifically, it will call *get_rtable_cost(dst)* and *get_rtable_port(dst)* for each node in the network to learn a node's view of the cost and route to a specific destination *dst*.

Handling Cost Changes

During the simulation, the following conditions may occur:

1. The cost associated with a port may increase/decrease.
2. A link may fail.

¹ In this document, I'm using the terms router, node, and node instance interchangeably. Some parts of the project infrastructure refer to routers as switches. I apologize for the confusion.

Either event may force the routing algorithm to choose an alternate route. The cost associated with a port can be retrieved by calling the `get_cost()` function with the appropriate local port identifier. The function returns an integer. A negative value for cost indicates that the port is not connected to any other router. To discover port cost changes, you need to periodically poll the port cost associated with a port. If there is a change in port cost, you may need to update your routing tables and inform other nodes about that change, depending on the routing protocol. A recommended time period to poll for cost changes is 5-10 seconds (about 5M-10M simulator clock ticks).

Note: Although `get_cost()` will return -1 for ports that are *initially* not connected, you *cannot detect link failures* by calling `get_cost()`. Instead, in order to detect link failure, you must use either an explicit or an implicit signaling protocol. In explicit signaling, each router periodically sends a HELLO message to all its neighbors to indicate that it is alive and well. Loss (failure to arrive) of multiple consecutive HELLO messages indicates failure of a link. In implicit signaling, nodes periodically exchange routing messages and take the absence of routing messages as evidence that a link has failed. A recommended time period after which to assume that a link has failed is 1 minute (about 60M simulator ticks).

Assumptions

For the purposes of this project, you may assume that the link layer does not introduce any errors or drop packets, unless the link has failed. Failed links are not resurrected later.

The maximum link cost is 10. Your implementation should not impose any artificial limits on the number of routers. If there are N routers in a given simulation, then the maximum path cost is $N*10$. The simulator will interpret any cost greater than $N*10$ as infinity.

At the end of the simulation, the routing tables on all routers will be consistent, unless link failure partitioned the network.

Keep in mind that there may be more than one link between a given pair of nodes. Your algorithm must pick the port that provides the lowest cost.

Simulator Functions

The functions offered by the simulator can be divided into four categories: transmit/receive functions, timer functions, utility functions, and scoring functions. These are discussed below, separated by inbound and outbound functions. You implement the outbound functions; you may call inbound functions in the course of doing so.

Transmit/Receive Functions

<i>write_msg</i> : Use this function to transmit a message along a link. You are responsible for the content and	<i>read_msg</i> : The simulator will call this function to notify you that a message has been received on a port and can
--	--

layout of the messages you exchange.	be read.
<pre>// inbound proc write_msg = 8 takes int16 port_id bytearray data returns int32</pre>	<pre>// outbound proc read_msg = 7 takes int16 from_node int16 port_id bytearray data returns int32</pre>
<p><i>port_id</i>: Local port number on which to transmit the message. Ports are numbered between 0 and nports-1. (nports is passed to init_node)</p> <p><i>data</i>: The data to be transmitted</p>	<p><i>from_node</i>: Node identifier of the transmitting node. Nodes are numbered 0 to nnodes-1.</p> <p><i>port_id</i>: Port identifier on which the message was received. (The port id is local with respect to the receiving node.)</p> <p><i>data</i>: The data received</p>
Returns 0	Returns 0

Timer Functions

Note that all timers operate according to the simulator’s virtual time, or forced clock.

<p>You can add timers that will fire once the simulation has progressed to the desired expiration time. No periodic timers are supported.</p> <p><i>del_timer()</i> will release the resources associated with a timer. If the timer has not yet expired, it will be canceled.</p>	<p>You must implement the action to be performed when a timer expired in the outbound <i>timer_expired</i> function. If you wish to obtain a periodic timer, make sure you re-arm your timer here.</p>
<pre>// inbound proc add_timer = 4 takes int32 expires int32 data returns int32 proc del_timer = 6 takes int32 timer_id returns int32</pre>	<pre>// outbound proc timer_expired = 5 takes int32 data returns int32</pre>
<p><i>expires</i>: time at which this one-shot timer expires. Use <i>add_timer(get_current_time() + X, _)</i> to schedule a timer X ticks in the future.</p> <p><i>data</i>: A integer value that will be passed to <i>timer_expired</i> when the</p>	<p><i>data</i>: Integer that was passed to <i>add_timer</i> when the timer was armed.</p> <p>You may use the <i>data</i> parameter to distinguish between multiple timers.</p>

timer expires.	
<i>timer_id</i> : id of timer to be canceled	
<i>add_timer</i> returns a <i>timer_id</i> suitable for use in <i>del_timer</i> . <i>del_timer</i> returns 0.	Returns 0

Utility Functions

<i>get_current_time</i> : Ask for the current simulation time.	<i>get_cost</i> : Retrieve the cost associated with a port.
<pre>// inbound proc get_current_time = 2 takes void returns int32</pre>	<pre>// inbound proc get_cost = 3 takes int16 port returns int16</pre>
<i>void</i>	<i>port</i> : The local identifier of the port.
Returns current time in ticks. Each clock tick represents 1 μ s.	Return the cost associated with the link. -1 indicates that the port is not connected in the initial topology. Note: if a link fails, <i>get_cost()</i> will return its old cost before it failed – you cannot use <i>get_cost()</i> to detect link failure.

A note on get_current_time(). For reasons that are beyond the scope of this document, *get_current_time()* does not always provide a monotonically increasing sequence of time stamps. As a consequence, you cannot use the simulation time as a means for distinguishing old link-state advertisements from new ones – this situation, however, mirrors the situation in a network in which routers also do not have access to precisely synchronized clocks.

Scoring Functions

All scoring functions are outbound.

<i>terminate_node</i> : The simulator will call this function as a courtesy before invoking <i>get_rtable_cost</i> and <i>get_rtable_port</i> . ²	<i>get_rtable_cost</i> and <i>get_rtable_port</i> extract information from a node's routing table.
<pre>// outbound proc terminate_node = 9 takes void returns int32</pre>	<pre>// outbound proc get_rtable_port = 10 takes int16 dest_node returns int16 proc get_rtable_cost = 11 takes int16 dest_node returns int32</pre>
<i>void</i>	<i>dest_node</i> : The destination node for

² It is acceptable to postpone running Dijkstra's algorithm until here for the link-state protocol.

	which routing table information is extracted.
Returns 0 Do not exit in <i>terminate_node!</i> Do not free or discard data structures you will need to access in subsequent calls to <i>get_rtable_port/get_rtable_cost!</i>	<i>get_rtable_cost</i> should return the total path cost from the current node to the destination node, as noted in the node's forwarding table. If the current node thinks the destination node is unreachable, a value greater than $N \cdot C$ (INFINITY) should be returned. <i>get_rtable_port</i> should return the local port along which the current node would send packets to the destination node. If the current node thinks the destination node is unreachable, a value of -1 should be returned.

How to Run the Simulator

The simulator is called *net_sim* in project2b. The simulator assumes that all nodes run on the same machine and have already been started. The nodes must be bound to a series of consecutive port numbers. For example, you may start 3 nodes on host cid like so³:

```
[you@cid> project2b_dv 10000 &
[you@cid> project2b_dv 10001 &
[you@cid> project2b_dv 10002 &
```

Then you start the simulator on, say, ghestal as follows.

```
[you@ghestal> netsim -C cid:10000:10002 topology1.dat
```

The simulator will connect to the nodes running at *cid:10000*, *cid:10001*, and *cid:10002* and start the simulation. If your routing protocol performed correctly, you should see, among other output:

```
Total score is XX out of XX or 100.0%
```

```
-----
                        Congratulations!
-----
```

$XX = n \cdot (n-1)$ for n nodes.

The simulator supports the following command line switches:

```
Usage: ./net_sim -C h:lp:hp [-L p] [-s vc] [-qndh] <topology file>
e.g. net_sim topology1.dat
-C host:lowport:highport
    peers are listening on host on ports [lowport, highport]
-q      (quiet) suppress progress display
-n      no cost changes or link faults
-d      dump routing table at the end even if successful
-s vc   use Verification Code from previous run
-L p    set transmission loss probability in percent, default=0 (no loss)
-h      print this help
```

³ I'm using 10000 as port number, but you should use the same number p as in project 1A.

At the end of the simulation, the simulator will output a verification code. You can pass this verification code to a subsequent run via `-s` if you wish to repeat the very same sequence of cost changes and link failures (the topology must be the same, of course).

Use the `-n` switch to turn off link failures and cost changes. We will not use the `-L` switch in this project. If your nodes report correct routing table information, the router will not print the routing table it calculated. Use `-d` to show the table anyway. Unless `-q` is used, the router will show how simulation time progresses and how many messages are transmitted by/received at the individual nodes. Make sure that you start at least as many nodes as are required by the topology file.

At the end of the simulation, the router will output the final topology in a topology file called `topology_state`.

Representing Topologies

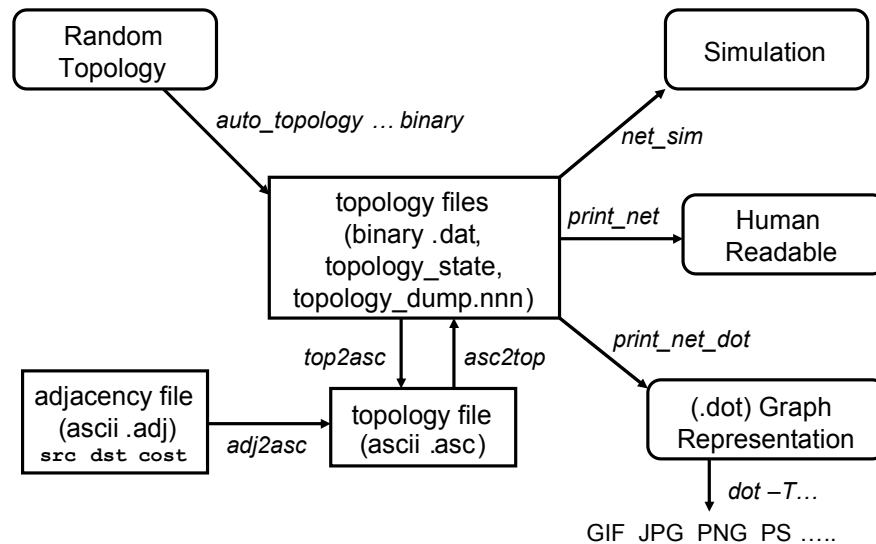


Figure 1: Topology formats used in this project and their relationships.

A *topology file* is a binary file that contains a topology that can be understood by the simulator. Topology files contain the number of nodes and ports, and for each node/port combination the destination node, destination port and link cost if the port is connected.

Use the program `top2asc.py` to create a text representation of the content of a binary topology file. The program `asc2top.py` will read output produced by `top2asc.py` and recreate the binary file.

Use the program `print_net` to view the content of a binary topology file in human-readable form. Use the program `print_net_dot` (redirect stdout into a file: `print_net_dot top.dat > top.dot`) to create a `.dot` representation of the topology.

Dot is a file format used by the graphviz package. I installed a copy of the *dot* executable in `~cs5565/bin`. Dot allows you to create visualizations of directed or undirected graphs in GIF, JPG, Postscript and other formats. Versions for Windows and Mac OS X are available for download from <http://www.graphviz.org/>.

Use the program *auto_topology* to create new topologies of different kinds. Run the program without arguments to figure out the arguments it takes.

You can also enter a topology as adjacency pairs without assigning port numbers. The program *adj2asc.py* will assign port numbers to such a topology and output a file that can be converted via *asc2top.py* into a binary topology file. The format of an `.adj` file has one link per line. Each line lists three parameters: `from_id to_id cost` assuming that there is a link from node `from_id` to `to_id` with cost `cost`. Below is an example of how to use *adj2asc.py*, *asc2top.py*, *print_net*, and *print_net_dot*. Pay particular attention to which programs require stdout redirection and which do not.

```

[cs5565@leo project2b]$ cat mesh.adj
0 1 1
0 2 2
0 3 5
1 2 4
2 3 1
[cs5565@leo project2b]$ ../dlsim/adj2asc.py mesh.adj > mesh.asc
[cs5565@leo project2b]$ ../dlsim/asc2top.py mesh.asc > mesh.dat
[cs5565@leo project2b]$ ../dlsim/print_net mesh.dat
Number of switches: 4
Ports/Switch: 3
Network Connectivity:
-----
Node 0
-----
Port 0 is connected to port 0 on node 1 with cost 1.
Port 1 is connected to port 0 on node 2 with cost 2.
Port 2 is connected to port 0 on node 3 with cost 5.
-----
Node 1
-----
Port 0 is connected to port 0 on node 0 with cost 1.
Port 1 is connected to port 1 on node 2 with cost 4.
Port 2 is not connected.
-----
Node 2
-----
Port 0 is connected to port 1 on node 0 with cost 2.
Port 1 is connected to port 1 on node 1 with cost 4.
Port 2 is connected to port 1 on node 3 with cost 1.
-----
Node 3
-----
Port 0 is connected to port 2 on node 0 with cost 5.
Port 1 is connected to port 2 on node 2 with cost 1.
Port 2 is not connected.
-----

[cs5565@leo project2b]$ ../dlsim/print_net_dot mesh.dat > mesh.dot
[cs5565@leo project2b]$ ~cs5565/bin/dot -Tgif -o mesh.gif mesh.dot
[cs5565@leo project2b]$ ~cs5565/bin/dot -Tps -o mesh.ps mesh.dot
[cs5565@leo project2b]$ ps2pdf mesh.ps mesh.pdf

```

The content of mesh.gif/mesh.pdf and the resulting routing cost table is shown next:

From/To	0	1	2	3
0	---	1	2	3
1	1	---	3	4
2	2	3	---	1
3	3	4	1	---

The correct routing cost table, assuming no cost changes/link failures, is shown above. On the left, costs are shown in red, port numbers are shown in blue. Node numbers are shown in green. *get_cost(2)* would return -1 when called by nodes 1 or 3, because their port 2 is not connected.

If you run the simulator with the `-nd` switches, you should see this if your implementation functions correctly:

```

[cs5565@leo project2b]$ ./net_sim -C zapote:7890:7893 -nd mesh.dat
...
Total score is 12 out of 12 or 100.0%
The correct routing cost table should have been:
From node 0:  ---  1  2  3
From node 1:  1  ---  3  4
From node 2:  2  3  ---  1
From node 3:  3  4  1  ---
...
    
```

Debugging Tips

I spent a considerable amount of time adding debugging and visualization functionality to the simulator, which was originally written by Dr. Varadarajan. In particular, I added the ability for the simulator to grade the router's forwarding tables, giving you instant feedback on how well your algorithm performed. I also developed the python scripts for binary/ascii conversion and the `print_net_dot` program. I implemented both parts of this project myself; I'd like to share some of the experiences I gained during this implementation.

Since the simulator is conceptually a single-threaded program, it might get stuck if any router does not return from an outbound call (for instance, because your algorithm ended up in an infinite loop). To reproduce this case, you can interrupt the simulator via `^C` to force it to dump the current topology (it will also show the current routing cost table). The dumped topology file is

named *topology_dump.nnnn*. It can be visualized with *print_net/print_net_dot* like any topology file. *nnnn* is the verification code you can pass to `-s` to repeat the failed run – you will then get exactly the same sequence of link cost changes and link failures. If you do not specify a `-s` flag, the simulator will create a different sequence of cost changes and link failures every time it runs (even on the same topology).

During debugging, you will notice that the time⁴ to finish a simulation run depends on the behavior of your routers. You might encounter a situation where simulation time appears to not progress or progresses very slowly, but the number of transmitted/received packets quickly increases, as shown by the simulator's progress display. Typically, this behavior means that you have a bug in your routing protocol implementation: for instance, your distance vector implementation may not recognize when the algorithm converges, or your link-state flooding algorithm may be buggy and may not recognize when it should stop forwarding a received link-state message.

Test Cases & Grading

We will make a number of test cases public on the class website: at a minimum, your submissions should pass those public test cases. For partial credit, your submissions should pass the public tests with the `-n` flag (no cost changes/link failures). We will set an appropriate timeout to wait for your implementation to finish.

In addition, we might use a number of secret test cases for additional grading. Do not assume that the public test cases test all possible topologies/cases. You should create test cases yourself to thoroughly test your protocols.

Submission

You must implement this project in an environment that is available on the lab machines reachable via `rlogin.cs.vt.edu`. It is your responsibility to port to and test your project on the lab machines if you do your development somewhere else.

⁴ The wall clock time, not the simulated time. (The simulation will always end after 2,147,483,647 ticks, or about 35 simulation minutes.) Runtimes you should expect for a correct implementation over a LAN with correct buffering of RPC requests for above `mesh.dat` topology are about 3 seconds for distance vector routing implemented in 202 lines of C with `-n` (no cost changes/link failures), about 6 seconds without `-n`, 6 seconds for link-state routing implemented in 282 lines of Java with `-n`, about 8 seconds for link-state routing implemented in Java without `-n`. In all cases, the number of packets transmitted by each individual node was below 1000. If your runs take substantially longer or if you transmit substantially more packets, make sure you buffer correctly and that you use the recommended time intervals for your explicit or implicit signaling protocol. Don't forget that `add_timer()` counts in ticks, but that 1 second of simulation time = 1,000,000 ticks!

Submit your source code along with instructions on how to build your project. Use the zip archive format. Do not include object file or executables or other intermediary files in your submission. If you use a graphical IDE, be sure to include instructions on how to *build* and *run* your programs from the command line. Include a `Makefile` or project or solution file, as appropriate.

To help you budget your time, you will submit this project in two parts: each part corresponds to one routing protocol. You can choose whether you submit the distance vector or the link-state algorithm first or second.

Your submissions should include a brief report that describes your implementation strategy for the respective algorithm. Include a description of the types and layout of the messages your protocol exchanges. You should also submit additional topologies you created. I may add topologies you submit to the set of secret test cases.

It is not necessary to describe the code structure of your actual implementation in your report (beyond what is necessary for the TA to run your project!)

Your submissions should include the **names and email addresses of all team members in all source files** and in the project report.

If you have any questions about the project, please do not hesitate to ask us – either by email or drop by during office hours or schedule appointments. Do get started on this project early, a significant portion of your grade will depend on it!

A Final Quote To Ponder

Show me your flowcharts and conceal your tables, and I will continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious.

– Frederick Brooks, in The Mythical Man-Month

Good Luck!