Routing Lab

Project C

Out: Monday, February 24
Due: Wednesday, March 12, midnight.

Overview

In this project your group will implement a distance-vector algorithm and a link-state algorithm in the context of a simple routing simulator. Your implementations can mirror the algorithms described in the book, and will consist of only a small number of lines of code. It is important that you understand what is going on before you start, however. While it will be tempting for you and your partner to each separately implement an algorithm, we believe it will be far easier if you collaborate on an algorithm.

This is the very first iteration of this lab. Your feedback on rough spots is very important. We'll probably release a few updates over the course of the lab.

Getting the Code

The routing lab is currently separate from Minet. To fetch it, cd to your home directory and then run:

```
cvs –d /home/pdinda/CVS_MINET co routelab-w03
```

This will create a routelab-w03 directory, which will contain:

```
context.cc     SimulationContext (Written for you)
context.h
demo.topo      A demonstration network topology file
demo.event     A demonstration event file
error.h
event.cc       Event (Written for you)
event.h
eventqueue.cc  EventQueue (Written for you)
eventqueue.h
link.cc        Link (Written for you)
link.h
Makefile
messages.cc    RoutingMessage (You will write this)
messages.h
node.cc        Node (You will extend this)
node.h
README
routesim.cc    main program
```

```
table.cc       RoutingTable (You will write this)
table.h
topology.cc    Topology (Written for you)
topology.h
```

To compile for the first time, execute "touch .dependencies". Next, execute "make depend". Finally, execute "make TYPE=GENERIC". This will build a single executable "routesim", which contains no routing algorithm. Thereafter, you should be able to just run "make TYPE=GENERIC".

To build routesim with your distance vector protocol, execute "make TYPE=DISTANCEVECTOR clean depend all". To build routesim with your link-state protocol, execute "make TYPE=LINKSTATE clean depend all".

To execute routesim, you should do the following:

```
export PATH=$PATH:/home/pdinda/netclass-execs
./routesim topologyfile eventfile [singlestep]
```

We will say more about topology and event files soon. You need to have netclass-execs on the path because routesim uses the dot and dotty programs available there to draw graphs for you. Singlestep indicates that routesim should show you every event before it dispatches it, and then wait for you to hit return after it has been dispatched. Even without singlestep, routesim will pause every time it draws a graph, waiting for you to close the graph window.

Event-driven Simulation

Routesim is an event-driven simulator. What this means is that instead of simulating the passage of time directly, it instead jumps from event to event. For example, suppose a node decides to send a routing message to neighbor. If the current time is 100, and the link latency to the neighbor is 10, then instead of simulating time 100.1, 100.2, …, 109.9, 110, the simulator "posts" an event (the arrival of the message at its neighbor) to occur at time 110. If there are no other events posted for times between 100 and 110, the simulator can jump ahead to 110.

Event-driven simulators are very powerful tools that are widely used in science and engineering. Routesim is implemented in the usual manner for event-driven simulators. There is an priority queue (implemented as a heap), called the event queue, which stores the events in time order. The simulator main loop simply repeatedly pulls the earliest event from the queue and passes it to a handler until there are no more events in the queue. The handler for an event may insert one or more new events into the event queue. For example, the handler for the routing message arrival may update the neighbor node's distance table and then post a new arrival event for *its* neighbor.

Events in Routesim

Events in routesim come from the topology file, the event file, and from handlers that are executed in response to events. The topology file generally only contains events that construct the network topology (the graph), while the event file generally only contains events that modify link characteristics in the graph, or draw the graph, a path, or a shortest paths tree. In the events and topology files, lines that are blank or whose first character is a '#' are ignored.

Here are events that can occur in a topology file:

```
arrival_time ADD_NODE node_num latency bandwidth
arrival_time DELETE_NODE node_num latency bandwidth
arrival_time ADD_LINK src_node_num dest_node_num latency bandwidth
arrival_time DELETE_LINK src_node_num dest_node_num latency bandwidth
```

Note that a network topology in Routesim is a *directed* graph. We've included a demonstration network topology file and will release more over time. Note that topology files are very easy to write.

Here are events that can occur in an events file:

```
arrival_time CHANGE_NODE node_num latency bandwidth
arrival_time CHANGE_LINK src_node_num dest_node_num latency bandwidth
arrival_time DRAW_TOPOLOGY
arrival_time DRAW_TREE src_node_num
arrival_time DRAW_PATH src_node_num dest_node_num
arrival_time DUMP_TABLE node_num
```

Note that any DRAW event will cause a window to pop up with a drawing of the topology. The simulation will stall until you close the window.

We've included a demonstration event file and will release more. Also note that these are easy to write.

After the topology file has been loaded and executed, routesim will post a CHANGE_LINK for each link to the node from which the link emerges. These events occur at a large negative arrival time, well before the events in the event file. The point of these events is to inform each of your nodes (through a call to Node::LinkUpdate(), see below) of its outgoing links.

Finally, your code is allowed to post routing message arrival events. Essentially, you only need to write handlers for CHANGE_LINK events and routing message arrival events. More about this later.

Note that although each link event contains both bandwidth and latency numbers, your algorithms will determine shortest paths using only the link latencies.

A Node of One's Own

To implement a routing algorithm in Routesim, you will extend the Node class and write implementations of the Table and RoutingMessage classes. You will wrap your implementation code in #ifdefs so that the code can be compiled using each of your algorithms. For example:

```
#if defined(GENERIC)
class Table {
our generic table code
};
#endif

#if defined(LINKSTATE)
class Table {
your link state code
};
#endif

#if defined(DISTANCEVECTOR)
class Table {
your distance vector code
};
#endif
```

The make file will add –D$(TYPE) to each compilation to choose one of the implementations.

Note that although Node contains a pointer to the simulation context for internal reasons, you are NOT PERMITTED to ask the simulation context about topology. Your routing algorithm runs in a node and can talk only to that node's neighbors.

Node has four functions that you must implement for each algorithm:

`void Node::LinkUpdate(const Link *l)` is called to inform you that an outgoing link connected to your node has just changed its properties. A pointer to a copy of the new Link is sent to you so that you'll know the new latency to that particular neighbor. Don't bother deleting the link. The framework will do this for you.

`void Node::ProcessIncomingRoutingMessage(const RoutingMessage *m)` is called when a routing message arrives at a node. In response, you may send further routing messages using `SendToNeighbors` or `SendToNeighbor`. Don't bother deleting the routing message. The framework will do this.

`void Node::TimeOut()` is called when you have requested that a timeout event be delivered to your node (see below).

`Node *Node::GetNextHop(const Node *dest) const` is called when the simulation wants to know what your node currently thinks is the next hop on the path to the destination node. The pointer returned should be to a ***copy*** of the next node. The framework will eventually delete this.

`Table *Node::GetRoutingTable() const` is called when the simulation wants to get a ***copy*** of your current routing table. The framework will eventually delete the table. We expect your routing table to be able to print itself.

Your implementation will consist of implementations of these five functions, as well as implementations of Table and RoutingMessage.

Your node can call the following functions. If you'd like to get control back at a point in the future, you can do this using

```
void Node::SetTimeOut(const double timefromnow)
```

This will cause Node::TimeOut() to be called at time now+timefromnow.

If, in response to a link update, your algorithm needs to send a routing message to your neighbors, you can do this using

```
void Node::SendToNeighbors(const RoutingMessage *m)
```

If you want to send a routing message to a single neighbor (you probably will not to do this), you can do this using

```
void Node::SendToNeighbor(const Node *n,
                          const RoutingMessage *m).
```

These functions will cause Node::ProcessIncomingRoutingMessage() to be called at time now+latency_of_link_to_node.

A node can discover its neighbors and outgoing links using

```
deque<Node*> *Node::GetNeighbors()
deque<Link*> *Node::GetOutgoingLinks()
```

It is safe to delete the deques that are returned.

Both the link-state and distance vector algorithms can be implemented without using the last three functions. Notice that link-state works by flooding, meaning that you'll want to flood a link update (and a routing message) to all of your neighbors. Similarly, when a path gets updated in a distance vector algorithm, all neighbors need to be informed.

General approach to implementing a routing algorithm

1. Make sure you understand what routesim (TYPE=GENERIC) is doing.

2. Develop a Table class. This should be provide you with what you need to implement the GetNextHop() and GetRoutingTable() calls. It should also be updatable, as its contents will change with link updates and routing messages.
3. Extend the Node datastructure to include your table
4. Implement Node::GetNextHop() and Node::GetRoutingTable()
5. Develop your routing message. Think about where your routing message will go. There are subtleties involved in flooding, in particular.
6. Implement Node::LinkUpdate
7. Implement Node::ProcessRoutingMessage()
8. Implement Node::TimeOut(), if needed.

Suggestions and Hints

- Since there are two routing algorithms, it will be tempting for you and your partner to each implement an algorithm independently. WE STRONGLY SUGGEST THAT YOU DO **NOT** DO THIS. It'll take some time to understand the framework, and that will be made a lot easier if the two of you work together. Furthermore, after you implement the first algorithm, you'll find the second to be a lot easier.
- You will probably find it easier to implement the distance vector algorithm first since it doesn't require any node-local representation of the topology.
- Your implementations should not assume limits on the number of nodes and edges. On the other hand, it does not have to be blindingly fast. You may find it very useful to leverage STL components such as vector. For example vector<vector<double> > would give you an extensible table for the distance vector algorithm.
- For the link-state algorithm, simple Djikstra is fine. Don't worry about making the graph implementation fancy. The graph implementation in topology.cc is probably going to limit your performance anyway.
- For link-state, you will need to implement flooding to convey a link update received by your node to all other nodes in the network. The simulator will only deliver link updates to the node from which the link is outgoing.
- You're going to have two rather different implementations of Node, Table, and RoutingMessage, link-state and distance-vector. You should separate these implementations using separate files or #ifs as shown above.
- Keep it simple. You won't be graded on performance. It is possible to write each algorithm in 100 lines of code or so.

Mechanics

- Your code must work on the tlab machines.
- If we run make TYPE=DISTANCEVECTOR clean depend all, we must get a routesim that implements distance vector. The same command with TYPE=LINKSTATE should give us a routesim that implements link-state.
- We will give you detailed handin instructions later.

Extra Credit

Read about network topologies and link change models and write a short summary (5 pages). A few important papers:

- V. Paxson*, End-to-end routing behavior in the Internet*, IEEE/ACM Transactions on networking, 5:5, 1997.
- M. Faloutsos, el al, *On power-law relationships of the Internet topology*, SIGCOMM '99.
- A. Downey, *Using pathchar to estimate Internet link characteristics*, SIGCOMM '99.

Run your algorithms on a variety of different topologies and report on their behavior. Some example topologies:

- Random
- Random Power-law (this is the SOTA model of Internet topologies)
- Linear
- Loop
- Tree (common auto-configured topology for Ethernet LANs)
- Fat Tree (common topology for expensive Ethernet LANs and some clusters)
- 2D Mesh (Common topology on parallel computers)
- 2D Torus (Common topology on parallel computers)
- 3D Mesh (Common topology on expensive parallel computers)
- 3D Torus (Common topology on expensive parallel computers)
- Hypercube (Mathematically beautiful topology)

Write topology generators that output routesim-format topologies for the above topologies.

Things That May Help You

- Chapter 4 of your book.
- An understanding of RIP and OSPF. Although the algorithms you're implementing here are not for hierarchical routing, they are asynchronous, with each node acting independently, and this presents many of the same issues as in RIP and OSPF.
- The handout "Make in a Nutshell"
- The C++ Standard Template Library. Herb Schildt's "STL Programming From the Ground Up" is a good introduction.