CIS 553 / TCOM 512 Networked Systems

Project 1: Link State and Distance Vector Routing

1.0 Milestone 1

In this milestone 1, you will work in teams to develop basic *neighbor discovery* capabilities to each node. The goals of this first milestone are to become familiar with the ns-3 development environment and understand the TA's skeleton code. Before you write any code, make sure you read in detail the code documentation, understand the API and structure of the relevant parts of the ns-3 code.

All your code should go inside the `upenn_cis553` directory and you should not modify other files in the ns-3 directory. Your assignment is to modify the file ls-routing-protocol.cc and dv-routing-protocol.cc in order to support neighbor discovery. Note that you are free to add new packet types to ls-message.cc and dv-message.cc. Feel free to structure your own code, for instance, introduce your own helper files, or have one neighbor discovery module shared by both LS and DV.

1. **Neighbor Discovery.** Your node continuously probes the network at a low rate to discover its immediate neighbors in the network topology. Again, you devise a solution within these constraints:
    - You may add additional packet types, e.g. periodic broadcast "hello" message on all outgoing interfaces to neighboring nodes to inform them of the node's presence. For each "hello" message, neighboring nodes respond with a "hello reply" message with their IP addresses. The TTL of the broadcast message has to be set to 1, to avoid flooding the message to entire network.
    - Neighbors disappear as well as appear (as specified in the scenario file or via the interactive command line). You will want to print out the current list of neighbors so you can see who they are, perhaps only printing when there is a change.
    - As a tip, you will need to use timers to implement continuous, low-rate background activity. Be very careful with automated mechanisms, especially when using flooding and broadcast! They should operate on the timescale of at least tens of seconds (tens of thousands of milliseconds in the API calls!).
    - Add the following command to your node: "DUMP NEIGHBORS" to dump a list of all of the neighbors to which your node believes it is currently connected. Each neighbor entry should include <neighbor node number, neighbor IP address, interface>. See the code documentation for the exact commands.

    The above constitutes the intellectual bulk of your milestone 1. It is probably simplest to implement the functions in the order they are given above. As you write your program, note that our TA solution is extremely short (in fact, the solution mirrors that of the ping/pong code). If you are writing hundreds of lines of code then you are probably doing something the hard way and should talk to us about it.

2.0 Milestone 2

Your assignment is to extend your node to support efficient routing by implementing two protocols: *link-state* (45%) and *distance-vector* routing (40%). If your implement works, you will be able to route packets hop-by-hop through the network, having packets propagate through a path, only involving nodes enroute to the destination.

2.1 Link-state routing

Your node must implement link-state routing to construct a routing table, and use this routing table to forward packets towards their destination. You should read about link-state routing in Peterson 4.2.3. Note that we are not implementing OSPF, but a different and simpler link-state protocol. The link-state protocol generally involves four steps:

1. **Neighbor discovery.** To determine your current set of neighbors. You will use the neighbor discovery code that you built in milestone 1.
2. **Link-state flooding.** To tell all nodes about all neighbors. You will have to implement flooding code that uses the neighbor table that you build in project 1 to disseminate link-state packets. Your flooding protocol should not result in infinite packet loops or unnecessary duplicate packets (i.e. a node should not forward a link-state packet it has seen previously, or has seen a more recent one). A key design choice is whether to send out a LinkState packet periodically or immediately after the neighbor list changes. A key design criterion is to distribute link state information using as few packets as possible, while keeping nodes as up to date as possible.
3. **Shortest-path calculation using Dijkstra's algorithm.** To build and keep up-to-date a routing table that allows you to determine the next-hop to forward a packet towards its destination. The least-cost route is the one with the fewest hops.
4. **Forwarding.** To send packets using the next-hops.

Some basic guidelines

1. See the details in the Peterson textbook for a good suggestion on how to implement Dijkstra's algorithm. The result of this algorithm should be a routing table containing the next-hop neighbor to send to for each destination address.
2. You should forward packets using the next-hop neighbors in your calculated routing table. The exception is packets sent to the network broadcast address (e.g., for neighbor discovery or sending link state information); these should be flooded. Note, then, that when your node receives a packet, it may perform one of three actions: (1) if the packet is destined for the node, it will "deliver" the packet locally; (2) if the packet is destined for another node, it will "route" the packet; (3) if the packet is destined for the broadcast address, it will both deliver packet locally, and continue flooding the packet (while avoiding infinite loops).
3. Add a command to your node: "DUMP ROUTES" to dump the contents of the routing table. You may find this more convenient than logging the entire routing table after every change. Note that this command is already in the skeleton code, and you need to find the function that corresponds to this command and modify it accordingly. Each routing table entry should contain <destination node number, destination IP address, next hop node number, next hop IP address, interface, cost>.

Once you have completed all of the above, you should be able to ping any other node (by node number) in the network, and have a packet travel to that node and back without being unnecessarily flooded throughout the network. To see that your protocol is working, we have provided an example test scenario. (Read Section 5 of code documentation for details.) For your own testing purposes, you might want to set up a small ring network, use ping to test whether you can reach remote nodes, and then break reachability by stopping a node on the path so that ping no longer receives a response. Your routing protocol should detect this and repair the situation by finding an alternative path. When it does, ping will work again. Congratulations! You have a real, working network!

2.2 Distance-vector routing

The second routing protocol you have to implement is the *distance-vector* routing protocol, described in Section 4.2.2 in the textbook. Your solution should address the count-to-infinity problem by bounding the

distance to a maximum of 16 hops. Note that we are not implementing the entire RIP protocol, but a simple distance vector routing protocol that consists of the following four steps:

1. **Neighbor discovery.** You will use the neighbor discovery code that you built in milestone 1.
2. **Distance-vector exchange**. Each node periodically sends its distance-vector to all its neighbors.
3. **Route calculation.** Based on the vectors received from each neighbor, each node computes a routing table similar to the above link-state protocol.
4. **Forwarding.** To send packets using the next-hops, similar to the link-state protocol.

For this protocol, we will leave the actual implementation more open-ended to allow for your creativity. Use the same command "DUMP ROUTES" to dump the table of distance vector. The routing table has the same format as that of link-state.

Given a similar network of diameter less than 16, your distance-vector protocol implementation should compute next hops with the same shortest path costs as your link-state protocol. Similar to the link-state protocol, all your code should go inside the `upenn-cis553` directory.

4.0 Extra Credits (due together with milestone 2)

Doing extra credit is entirely optional. We offer extra-credit problems as a way of providing challenges for those students with both the time and interest to pursue certain issues in more depth. You should only attempt the extra credits after you have completed the regular portions of the project. **Do note that if your regular credit LS and DV do not work correctly, we reserve the right not to award extra credit points.** Hence, we recommend that you only start working on extra credits after you have finished the regular credits.

Here are some example extra credits that you can implement. For extra credit, you are responsible for providing your own test cases and design documents:

- **Path vector protocol** (5%). Similar to distance vector protocol, except that complete paths are advertised. To avoid the count-to-infinity problem, cycles must be explicitly checked.
- **Hazy sighted link state (HSLS)** (15%, possibly up to 25%). HSLS is another variant of Link state routing for mobile ad-hoc networks. You can search online to read up more on this protocol. To get the full 15% credit, you have to demonstrate a working implementation running an actual mobile ad-hoc network (i.e. replace our current pt-to-pt network with an ns-3 MANET setup). For another 10% credit, you can demonstrate a working implementation of another MANET routing protocol not currently available in ns-3 (talk to the teaching staff to get your protocol approved).
- **Delay tolerant networking** (15%). Implement summary-based epidemic routing protocol. Demonstrate actual implement in a highly disconnected wireless network where nodes are connected to each other infrequently.
- **Metric-based link-state protocol** (10%), based on Section 4.2.4. Instead of computing routes based on fewest hop count, allow routes to be selected based on link metrics (e.g. lower loss rates, lower latency, higher bandwidth are all reasonable choices). Your design can and should make use of the facilities defined above (link state advertisements). Link metrics should be syntactically generated (either by writing your own generator or modifying the Inet topology generator), which you have to import into our simulator.
- **Incremental Djkstra computation** (10%). Given a route update, instead of recomputing Djkstra from scratch for all entries, perform incremental recomputation that only updates routes relevant to the shortest path. To demonstrate working implementation, you need to demonstrate that your simulation can run significantly faster for the same network size (while computing the correct routes of course!).
- **Your proposal (X%).** If you have a cool extension in mind, feel free to contact the teaching staff to discuss feasibility and points awarded.