CSE 160 – Computer Networks

University of California, Merced

Project 2: Link State Routing

Introduction

Your assignment is to extend your TOSSIM node to support efficient routing. In project 1, you were able to send packets from a source to a destination using flooding. In this project, you will route packets hop-by-hop through the network, having packets propagate through a path, only involving nodes en route to the destination.

Objectives and Goals

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 3.4.3 (page 242). Note that we are not implementing OSPF, but a different and simpler link-state protocol. A link-state protocol generally involves four steps:

- **Neighbor discovery**. To determine your current set of neighbors. You will use the neighbor discovery code that you built in project 1.
- **Link-state flooding**. To tell all nodes about all neighbors. You will use the flooding code that you built in project 1 to disseminate link-state packets.
- **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.
- **Forwarding.** To send packets using your routing table for the next hops.

Where to Start

You should create a LinkState struct that contains a list of the current neighbors for a given node. Note that the link state information is the "payload" (packet contents) of a packet with a normal packet header.

You should use your solution from assignment 1 to flood the link state information throughout the network so that all nodes know the neighbors of other nodes. A key design choice is when to send out a LinkState packet. Periodically? 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.

See the details in Peterson 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. Note that you should not use a link in your routes unless both ends of the link agree that it is available (that they are neighbors).

For your link-state packets, you should use the PROTOCOL_LINKSTATE (2) protocol. Please note that at the moment the protocol is accidentally called PROTOCOL_LIINKEDLIST in the skeleton code, change this to PROTOCOL_LINKSTATE if not already corrected.

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, subject to the TTL and "floods do not propagate forever" constraints from the last project.

To demonstrate your solution, you should be able to call a function in your python run script to print out all of the link state advertisements you used to compute the routing table, and to print the contents of the routing table. You may find this more convenient than logging the entire routing table after every change.

The command should be:

```
def cmdRouteDMP(destination):
```

Requirements

Once you have completed all of the above, you should be able to send a packet over multiple hops to a node and back without being unnecessarily flooded throughout the network. To see that your protocol is working, you might 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. This is the turn-in exercise. Congratulations! You have a real, working network.

Deliverables

What is expected on the due date is the following:

- Source code of the TinyOS implementation with working routing
- A single-page document describing the design process and your design decisions.
- A document with short answers to the discussion questions.

A physical copy of the document and related questions is required on the due date. All documents and a copy of the source code should be submitted to the class website. Please create a

compressed tarball such as Student-Name-proj2.tar.gz. You must do this before class on the day that it is due.

Additionally, you will need to demonstrate that the code you submitted works in the next lab, and be able to describe how it works. Also, discuss the design process in the code.

Discussion Questions

1. Why do we use a link for the shortest path computation only if it exists in our database in both directions? What would happen if we used a directed link AB when the link BA does not exist?
2. Does your routing algorithm produce symmetric routes (that follow the same path from X to Y in reverse when going from Y to X)? Why or why not?
3. What would happen if a node advertised itself as having neighbors, but never forwarded packets? How might you modify your implementation to deal with this case?
4. What happens if link state packets are lost or corrupted?
5. What would happen if a node alternated between advertising and withdrawing a neighbor, every few milliseconds? How might you modify your implementation to deal with this case?

Grading Guidelines

Each part of the project is graded on a 5-point (0-4) scale, multiplied by the weight of the project. The weighted grades from all parts of the project are added together to produce the final grade.

The five-point scale is based on how well you show your understanding of the problem, and in the case of code how well your implementation works:

0 – nothing turned in

1 – show minimal understanding of the problem / most things don't work

2 – show some understanding of the problem / some things that work

3 – show a pretty good understanding of the problem / most things work

4 – show an excellent understanding of the problem/everything works

The weights for Project 2 are:

90% - Link state routing implementation

5% - Write-up design decisions

5% - Discussion questions

Your submission will be graded on correctness, completeness, and your ability to explain its implementation.