Delay-Tolerant Link-State Routing

Ben Andrew, ba405
Magdalene College

February 7, 2022

1 Declaration of Originality

2 Proforma

3 Table of Contents

4 Introduction

5 Preparation

6 Implementation

6.1 Repository Overview

```
dtlsr                          Root
|- configs                     CORE config files
|- include                     Include directory
|  |- algorithm
|  |- network
|  "- process
|- src                         Source directory
|  |- algorithm
|  |- network
|  "- process
|- tests                       Unit tests directory
|  |- algorithm
|  |- network
|  "- process
|- tools                       Custom development tools
"- core-py                     Custom CORE scripts
```

`algorithm` contains pure algorithms and data structures, for example graph, graph searching, and heap implementations.

`network` contains functionality for socket manipulation, and sending data over the network.

`process` contains funtionality for interacting with both CORE and the Unix OS, for example retreiving node information, logging, and manipulating the Unix routing table.

The project was implemented almost entirely in C, with CORE-specific scripts in Python. I used the CMake build system and Check unit testing framework.

6.2 CORE Background

The Common Open Research Emulator (CORE) is a network emulator that models nodes as lightweight Unix virtual machines, complete with filesystems and network interfaces.

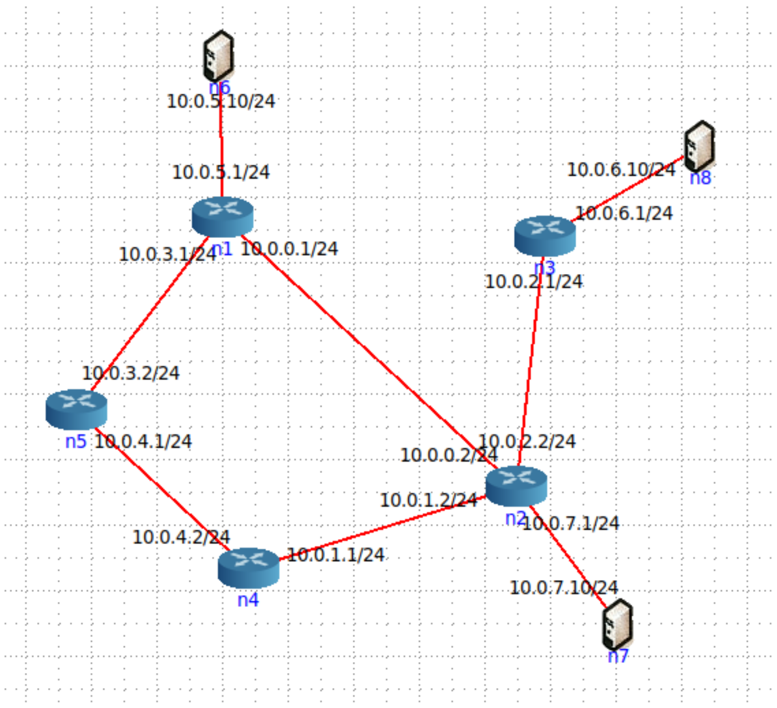

Figure 1: Example emulated network in the CORE GUI, showing routers and hosts.

Programs can be written for these nodes and run as Unix processes via a Python API. These processes are then able to send and receive data through the emulated network via standard Unix socket programming. The emulated routers contain default routing protocol implementations, such as OSPFv2 and v3 from the Quagga protocol suite, which can be disabled and replaced by custom implementations. As well as this, standard network testing tools like `iperf` can be run on emulated endpoint hosts, making testing and evaluation of the network surprisingly simple.

Link properties can be programmatically customised through the Python API, with properties including bandwidth, delay, loss, and jitter. This will aid greatly in evaluation, for example simulating network partitions.

One limitation is that CORE operates in real-time using the OS's clock, and thus the performance of the simulated network is somewhat dependent on the performance of the machine it's being run on. I will therefore be taking great care to create a reliable and statistically sound testing environment, for example setting processor priority, not running other programs, and taking baseline measurements to compare against.

6.3 Link-State Routing Design

TODO: include DefaultRoute2, and how I don't do neighbour discovery as it isn't relevant to the project.

The link-state routing protocol that I implemented is split into two programs, `heartbeat` and `dtlsr`. Routers run both programs, and hosts run only `heartbeat`.

`heartbeat` sends periodic heartbeat messages to all neighbouring nodes, notifying them that the connecting link is still up. The message is received by `dtlsr`.

`dtlsr` implements the bulk of the protocol. It maintains the link-state graph representation of the network, manipulates the node's routing table, and sends link-state announcements (LSAs) in response to detected changes in the network.

These changes are detected in one of three ways:

1. A heartbeat message is received from a neighbour with a DOWN link.
2. A heartbeat is not received in time from a neighbour with an UP link.
3. An LSA is received from a neighbour.

Both programs lend themselves to an event-driven design, and so I used file descriptors multiplexed with the `select` Unix call to detect and respond to events. Timer file descriptors trigger when they expire and receiving socket file descriptors trigger when data arrives.

Heartbeats and LSAs are recieved on sockets with distinct ports, allowing them to be differentiated easily. While CORE supports IPv6 routing, I restricted myself to IPv4 to reduce unnecessary implementation complexity.

Routes are derived from the link-state graph, where we compute the shortest path from ourselves to every other node in the graph. The metric for a path is given as the number of hops in the path, which I chose over a more standard OSPF-style metric of bandwidth as I am more concerned with delay than throughput. Notably, if a link is DOWN, we disallow the pathfinding to go through that link. Once we have a list of paths we use the next-hop address to add each route to the routing table, after aggregation is done on routes with the same next-hop and network prefix.

6.3.1 Delay-Tolerant Modifications

For the delay-tolerant version, `dtlsr` can be recompiled with the `-DDTLSR` flag active, keeping `heartbeat` identical. `dtlsr` maintains the same implementation design decisions as above with a few additions.

Firstly in the pathfinding algorithm we allow routes through DOWN links, so that when we update the routing table packets will be sent along the path up to the DOWN link. The metric is modified to be, instead of hop count, an exponential time-average of the uptime history of the node, so that unreliable nodes will have a much higher cost. These uptime histories are maintained locally for neighbour links only, when we advertise our link state we share only the derived metric. The metric for a path is still the summation of costs of each link along the path.

7 Evaluation

8 Conclusions

9 Bibliography

10 Appendices

11 Index

12 Project Proposal