Project: Routing Protocol

Deadline:

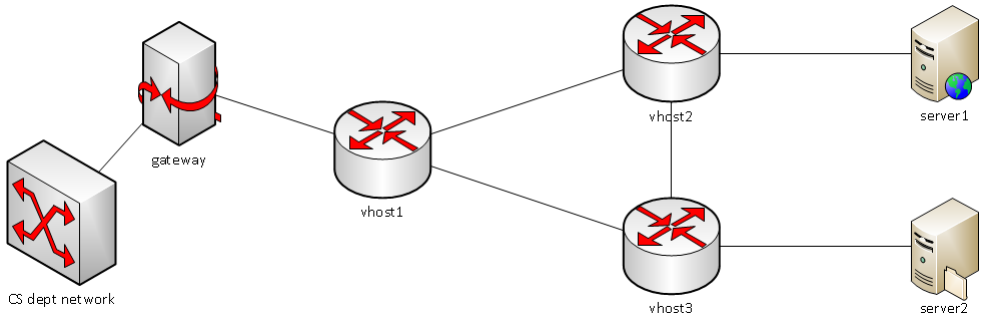
Sunday 12/1/2022, 11:59pm

Overview

This assignment is to implement a link-state routing protocol, PWOSPF, based on your phase 1 simple router, such that the router can build its own routing table from link-state routing messages, and detect link failures and recovery.

The protocol specification of PWOSPF is in a separate document on D2L.

Each student will be assigned the following topology for development:



The topology has three routers: vhost1, vhost2, and vhost3. All three routers will run your virtual router software. With PWOSPF, these routers should exchange routing messages to learn the network topology, compute shortest path to each subnet, detect the failure and recovery of links, and re-compute path when needed.

Note: This topology is only accessible from within the CS department network.

Step 0: Rebase with new stub code

Since this assignment builds on top of the first project, you must have a working phase 1 router to start with.

To ease the implementation of PWOSPF, we've made available modified stub code that creates a PWOSPF subsystem structure within *struct sr_instance* and starts thread that can be used to implement the necessary timer infrastructure. The new stub code also includes a header file *pwospf_protocol.h* which contains some useful definitions.

You can either start with this PWOSPF stub code (download from D2L) and integrate your own code from phase 1 into it, or, you can stick with your current codebase and copy over from the new stub code for the PWOSPF subsystem.

Changes in this project:
1. thread,
2. new structures

Here're the details of the changes in the PWOSPF stub code:

- Use the Makefile from the PWOSPF code. If you have modified the Makefile (e.g., adding source files or libraries) in phase 1, you will need to apply your changes to the Makefile again.
- Put the files *sr_pwospf.h, sr_pwospf.c*, and *pwospf_protocol.h* in the same directory as your other sr source files.
- In sr_router.c, add #include "sr_pwospf.h" to your header includes.
- Also need to add *pwospf_init(sr)* to sr_vns_comm.c.
- Compare other files in the PWOSPF stub code with those in the sr stub code to identify other needed changes.

Lastly, compile the router, test ping and web access using phase 1 topology to make sure everything still works.

Step 1: Run multiple routers and test the topology

Because this assignment requires multiple instances of your router to run simultaneously you will need to use the **-r** and the **-v** command line options. **-r** allows you to specify the routing table file you want to use (e.g. -r rtable.vhost1) and **-v** allows you to specify the vhost you want to connect to (e.g. -v vhost3).

For example, to start all three routers using static routing tables you need to run the following commands in **separate** terminals:

```
./sr -t 300 -v vhost1 -r rtable.vhost1
./sr -t 300 -v vhost2 -r rtable.vhost2
./sr -t 300 -v vhost3 -r rtable.vhost3
```

Make sure that our topology would work in these static tables without implementing any routing protocol

These rtable files provide static routes specific to each vhost, and enable packet forwarding in your topology. They are provided for testing purpose only. You will not need them after PWOSPF has been implemented.

Make sure that you can ping all interfaces in your topology without loss. If not, you may need to debug your packet processing, forwarding and table lookup.

Understanding your routing tables

In the topology, **each link is a subnet**. It has two IP addresses, one for each end. To successfully forward packets to any of the addresses in the topology, a router needs to have an entry in its routing table for every subnet. It is useful to examine the content of the static rtable files of each router, and you can see that it covers every subnet of the topology plus the default route pointing to the gateway. Since there're 5 subnets and one default route in our topology, you would expect 6 entries in rtable. The supplied static rtable files may have fewer entries since some of the entries can be aggregated together but still cover the same address space. With PWOSPF, your router will build its own routing table (instead of reading it from a file), that covers the same set of subnets; Aggregation is not required.

Routing table lookup and longest prefix match are the same as project 1.

if dest is 0.0.0.0, it is gateway

RT has dest and mask

Step 2: Implement PWOSPF

PWOSPF is a simplified version of OSPF. The specification of the protocol is available on D2L.

The expected result is that, when you start vhost1 with rtable.net (which has only a single route pointing to the Internet), and vhost2 and vhost3 with rtable.empty (which has no route), with PWOSPF running, the routers should discover each other, send routing updates, and build their own routing tables that are equivalent to the static content in rtable.vhost. With these dynamically built routing tables, packet forwarding (ping, web) should all work correctly. Even more, when a link fails and recovers, your routers should be able to detect these events, send updates, recompute the routing table entries, and make packet forwarding work again.*

Static vs. Dynamic Routes: During operation, your routing table should contain both static and dynamic routes. Static routes are read in from the routing table (rtable) and should never be removed or modified. Dynamic routes are added/removed by your PWOSPF implementation. You must therefore keep track of which routes are static. You handle this however you like, e.g. maintaining two separate tables, or labelling routes as static/dynamic internally. The only static route in the final product is the single route in rtable.net.

What Routes to Advertise: Your router is responsible for advertising all the subnets connected to its interfaces. In our topology, each vhost should advertise 3 subnets corresponding to its 3 links.

How to store the topology and how to compute paths: Given the network topology, a router should compute shortest path to each subnet. It is totally up to you how to store the topology and how to compute the paths. A few things that you may want to take advantage of:

- In this project the cost of each link is 1. Although you can implement Dijkstra's algorithm, it is not necessary. A breath-first search will work just fine. Furthermore, you can even assume the topology of three routers and do whatever you want to get the shortest paths. Just don't hardcode any addresses from your topology.
- What you will need in the end is the routing table content, which tells how to reach each subnet (i.e., IP prefix), not each node (i.e., a single IP).
- Generalization or performance optimization are not required. As long as the routers can deliver packets in this given topology with reasonable performance it'll be fine. Grading will be done using the exact same topology but different IP/MAC addresses.

Potential Pitfalls
A router should add its own directly connected subnets into the routing table at startup. These subnets may be announced by a neighbor router as well. In this case, don't add the same subnets twice into the routing table.

You probably need two threads: one for all packet-driven events, i.e., actions in response to the arrival of packets, and another one for periodical HELLO packets. If you use multiple threads, be careful to avoid race conditions. Your routing table will need to be locked so that you don't fall of the end when looking up a route during updates.

If you don't have any experience with thread programming, an alternative is to use signals to

implement timers to handle periodic events.

Debugging can be challenging because you're dealing with 3 routers. To debug the network-wide behavior, you probably will rely on printf and tcpdump. When reading the packet traces, tcpdump may not be able to recognize all the headers of PWOSPF and report malformed OSPF packets. This is normal. Tcpdump/wireshark/ethereal can still serve as a crude hex viewer for what is going on on the wire.

Step 3: Tests under topology changes

Now with PWOSPF, the routers can build the routing tables on their own. In this step, you need to test these routers under link failure and recovery.

In your topology package, there is a script, vnltopoXX.sh, where XX is the topology number. This script can set the loss rate of a link. To fail a link, you set the loss to 100% on **both** ends of the link, and to recover it you set the loss to 0%.

For example, to fail the link between vhost1 and vhost2, run both of the following commands:

vnltopoXX.sh vhost1 setlossy eth1 100
vnltopoXX.sh vhost2 setlossy eth0 100

To bring the link back up:

vnltopoXX.sh vhost1 setlossy eth1 0
vnltopoXX.sh vhost2 setlossy eth0 0

To check the current loss rate:

vnltopoXX.sh vhost1 status

1. run with RT having 1 entry to gateway in R1
2. run with RT is empty in R2 and R3

1. hello: when router comes up, it will send hello msg to other routers
2. link state announcement: flood LSA to everyone to learn about topology

once you get the topology, get the shortest path using any algorithm, eg: do breath-first search

Testing: 1. start 3 router for few minutes and do web downloading
2. fail one link (in description), detect the failure and check by hello msg, router should recompute and update RT again.
3. put the failed link back up, router should detect and update RT again

Another test that you can do using this script is pinging from server 1 to server 2 and vice versa.

vnltopoXX.sh server1 ping ip_of_server2

Required Functionalities of the End Product

1. Start vhost1 with rtable.net, vhost2 and vhost3 with rtable.emtpy, wait for one minute to allow routing convergence. The following should work: ping all interfaces of all routers, ping and wget both servers, and ping from server1 to server2 and vice versa.
2. Fail one link, vhost1-vhost2 or vhost1-vhost3 or vhost2-vhost3, wait for one minute to allow routing convergence. All the ping and wget should work.
3. Bring the failed link up, wait for one minute to allow routing convergence. All the ping and wget should work.
4. If none of the above works, the completion of step 1 will be tested and evaluated for partial credit. In this scenario, the routers will be started with static rtable.vhost* files, and tested with ping and wget.

Grading

The project will be graded on the same topology but with different IP assignment. Therefore do NOT hardcode anything about your topology in the source code.

You work in the same group as in project 1, unless you notify the instructor about group change and get approved.

Your code must be written in C or C++ and use the supplied stub code.

Your router will be graded on department Linux machine Lectura only.

Grading is based on functionality, i.e., what works and what doesn't, **not the source code**, i.e., what has been written. For example, when a required functionality doesn't work, its credit will be deducted, regardless of whether it's caused by a trivial oversight in the code vs. a serious design flaw.

Submission

Only one submission per group.

1. Rename your working directory stub_sr to "topXX", where XX is the topology ID in your assignment.

2. Make sure this directory has all the source files and the Makefile. Include a README.txt file listing the names and emails of group members, and anything you want us to know about your router. Especially when it only works partially, it helps to list what works and what not.

3. Create a tarball

```
cd topXX
make clean
cd ..
tar -zcf topXX.tgz topXX
```

4. Upload topXX.tgz onto D2L/Gradescope.