Implementing A Routing Protocol

Deadline:

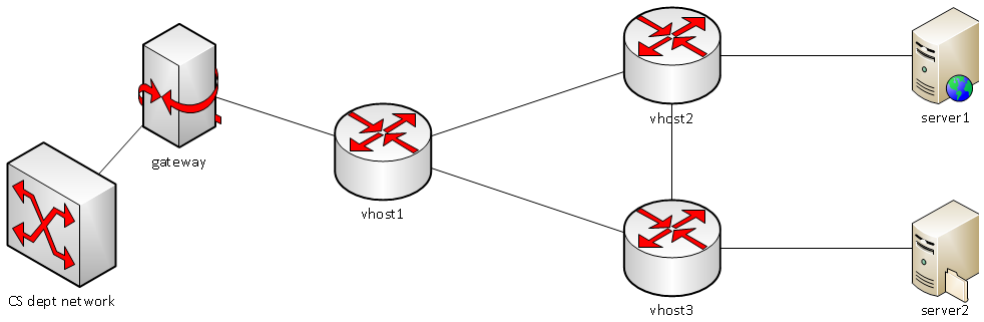
Friday 12/9/2016, 11:59pm

Overview

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

The protocol specification of PWOSPF is available on D2L.

Each student will be assigned the following topology for development:



The topology has three routers, namely vhost1, vhost2, and vhost3. All three routers will run your virtual router software and PWOSPF implementation. Once PWOSFP is running, 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 necessary.
Note: This topology is only accessible from within the CS department network.

Step 0: Prep your router

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 incorporate your own code from phase 1 into the new stub code, or, you can stick with your current codebase and copy over from the new stub code for the PWOSPF subsystem.

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.
- You need the new *sr_vns_comm.c*, as well as the updated *struct sr_if* from *sr_if.h*.
- If you're adding PWOSPF stuff into your project 1 code, then in sr_main.c, change "*return 999"* to "*return 0"* to allow empty rtable file. 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, traceroute, 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, connecting to vhost3 on topology 300 should look like:

./sr -t 300 -v vhost3 -r rtable.vhost3

You should start all three routers using the rtable files from the assignment email. These rtable files provide static routes needed for packet forwarding in your topology.

Important note: 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, each 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. There is also a default route pointing to the outside of the topology.

Dealing with Routes

When implementing pwospf, special care must be taken to ensure that routes are treated correctly by your router. Before starting pwospf development, we suggest that you first ensure your router properly handles default routes, gateways and subnets.

Next Hop: The routing table consists of 4 fields: destination prefix, next-hop, network mask and the outgoing interface. To deliver a packet, the router should forward the packet to the next-hop router using the outgoing interface. If the destination is directly connected to the router, the router should forward the packet to the destination directly. In this case the next-hop in the routing table will be 0.0.0.0. For example, if a router receives a packet going to destination 1.2.3.4 and the routing table has the following entry:

1.2.3.0 0.0.0.0 255.255.255.0 eth1

The router knows that the destination is directly connected, thus it will find the MAC address of 1.2.3.4, and send the packet to it via eth1.

Longest prefix match: The subnet-mask field of the routing table specifies the size of the subnet being routed to. More specifically, given a particular destination IP address (ipdest), it matches with a route if (ipdest & mask) == (dest & mask), where dest is the destination prefix field in the route. It is important to note that a given IP address can match with multiple routes (in fact this is often the case). For example, all destinations match the default route ((ipdest & 0) = 0). Given multiple matches, the router must choose the **longest prefix match**. This is done by selecting the match with the longest prefix or the largest value subnet mask.

Step 2: Implement PWOSPF

PWOSPF is a link-state routing protocol. It builds the routing table and maintains it in face of topology dynamics. 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, traceroute, web) should all work.*

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 marking routes as static/dynamic.

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 3 links of its own respectively.

How to compute paths: Given the network topology, a router should compute shortest path to each subnet. It is up to you how to store the topology and how to compute the paths, but there're a number of things to pay attention to:

- The cost of each link is 1. In this case you don't have to use Dijkstra's algorithm; breath-first search will work just fine. It is your decision what algorithm and data structure to use for the path computation.
- What you will need in the end is the routing table content, which is about how to reach each subnet (link), not each node.
- 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 address assignment.

Some Pitfalls

Be careful not to add a route in your routing table for subnets directly connected to one of your interfaces even if these subnets are being advertised (in many cases they will be by the router on

the other end of the link).

If you use multiple threads in this assignment to support the periodic updates (e.g. HELLO packets). Be careful to avoid race conditions. Your routing table in particular will need to be locked off so that you don't fall of the end when looking up a route for a data packet during updates.

If you don't have any experience with thread programming, a safer alternative is to use signal 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. Failing a link is to set the loss to 100% on both ends, and recovering a link is to 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

Another test that you should do using this script is pinging from server 1 to server 2 with or without link failures:

vnltopoXX.sh server1 ping ip_of_server2

Required Functionalities

1. Start vhost1 with rtable.net, vhost2 and vhost3 with rtable.emtpy, wait for one minute to allow routing convergence. Ping and web to both app servers should work. Pinging from server1 to server2 and vice versa should work. Traceroute to servers should show shortest path.

2. Fail link vhost1-vhost2 or link vhost1-vhost3, wait for one minute to allow routing convergence. Ping and web to both app servers should work. Pinging from server1 to server2 and vice versa should work. Traceroute to servers should show shortest path.
3. Bring the failed link up, wait for one minute to allow routing convergence. Ping and web to both app servers should work. Pinging from server1 to server2 and vice versa should work. Traceroute to servers should show shortest path.
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 corresponding rtable.vhost* files, and ping/traceroute/web will be tested.

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 can work by yourself or in a group of two students. No group can have more than two students.

Your code must be written in C or C++ and use the 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. Name your working directory "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.