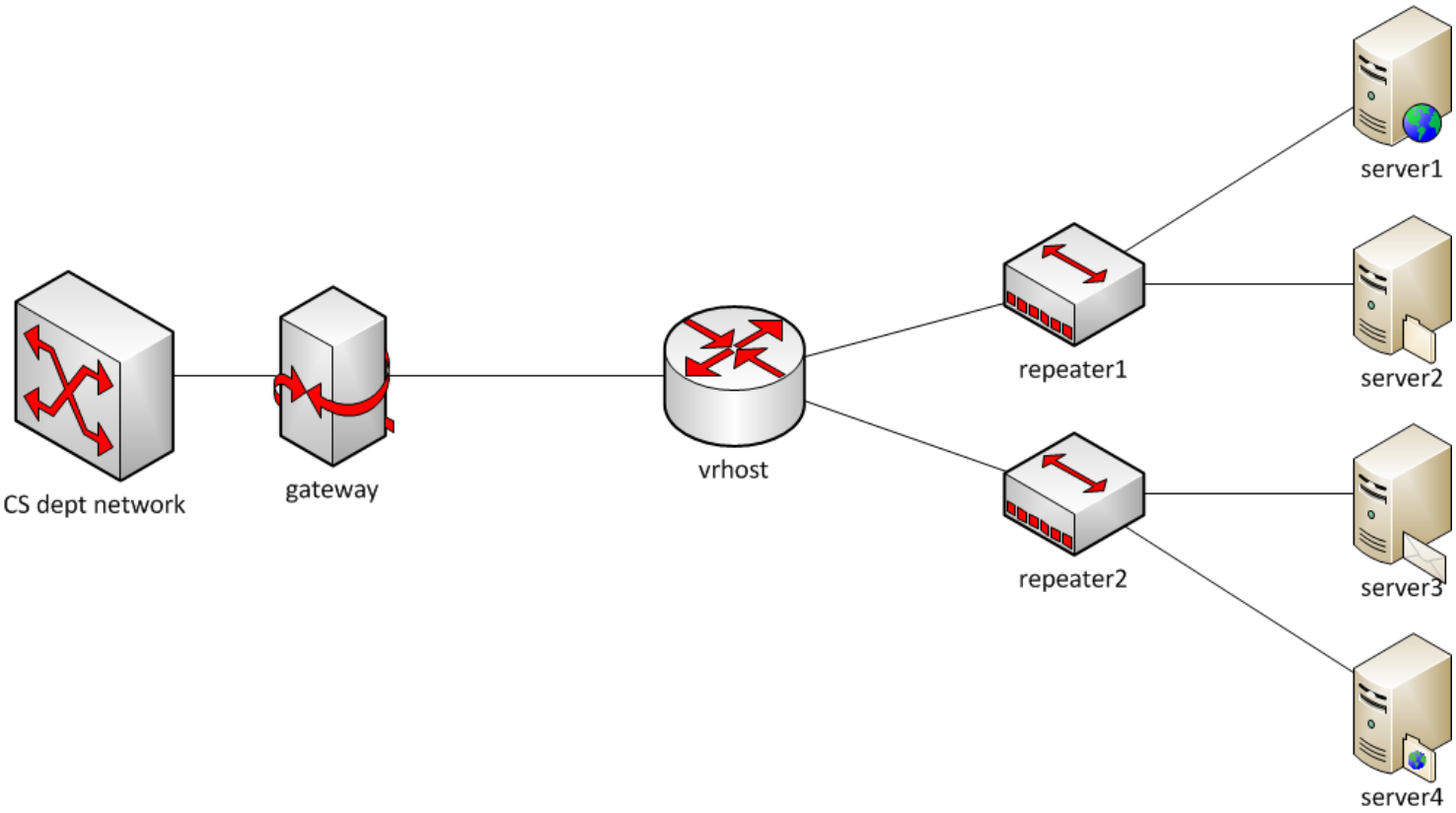
Building your own Internet Router

Introduction

In this assignment you will implement a fully functional Internet router that routes real network traffic. The goal is to give you hands-on experience as to how a router really works. Your router will run as a user process locally, and when finished will route real packets that are flowing across the CS department network to application servers. We'll be giving you a skeleton, incomplete router (the "sr" or simple router) that you have to complete, and then demonstrate that it works by performing traceroutes, pings and downloading some files from a web server via your router.

Overview of the Virtual Network Lab (VNL)

Virtual Network Lab (VNL) is an educational platform where students can gain hands-on experience on network protocols by programming routers and hosts. It is inspired by Stanford VNS. VNL is comprised of two components: (1) The VNL services which run in a set of virtual machines on `postino.cs.arizona.edu`, and (2) VNL soft-hosts such as your router. The service intercepts packets on the network, forwards the packets to the soft-hosts, receives packets from the soft-host and injects them back into the network. The soft-hosts are run locally by the students as regular user processes and connect to the service via ssh tunnels. Clients, once connected to the server, are forwarded all packets that they are supposed to see in the topology. The soft-hosts can manipulate the packets in any way they wish, generate responses based on the packets, or make routing decisions for those packets and send the replies back to the service to place back onto the network.



For example, on the above topology, the VNL service on `vrhost` might receive a TCP SYN packet from the CS department network destined for application server 1. The VNL service sends the packet to the VNL soft-host which would receive the packet on interface eth0, decrement the TTL, recalculate the header checksum, consult the routing table and send the packet back to the service with directions to inject it back onto the network out of interface eth1. What will the destination ethernet address be for the packet sent back by the client? What if the client doesn't know the ethernet address for application server 1?

In this assignment we provide you with the skeleton code for a basic VNL soft-host (called `sr` or Simple Router) that can connect and talk to the VNL service. Therefore, you don't need to be concerned about the interaction between VNL service and soft-host, or how packets flow in the physical topology. You can just focus on the virtual topology and your own router. Your job is to make the router fully functional by implementing the packet processing and forwarding part within the skeleton code. More specifically, you'll need to implement ARP, ICMP, and basic IP forwarding.

Test Driving the `sr` Stub Code

Before beginning development you should first get familiar with the `sr` stub code and some of the functionality it provides. Download the Stub Code Tarball from D2L and save it locally. You also need a VNL user package as a separate download (the download link is emailed to you). As described before, it handles all of the dirty-work required for connecting and communicating with the service. To run the code, untar the code package `tar -zxvf stub_sr_vnl.tar.gz` and the VNL user package `tar xf vnltopo*.tar`, move the files from user package into the stub_sr directory, and then compile it via `make`. Once compiled, you can connect to the VNL services as follows:

```
./sr -t <topo-id>
```

For example, connecting to the service on topology 87 would look like:

```
./sr -t 87
```

Other options that are useful include `-r routing table`. You can use `./sr -h` to print a list of the accepted command line options.

You are only able to access the VNL service from within CS department network.

After you connect successfully, the service will send you a description of the host including all the interfaces and their IP addresses. The stub code uses this to build the interface list in the router (the head of the list is member `if_list` for `struct sr_instance`). The routing table is constructed from the file `rtable` and by default consists of only the default route which is the gateway sitting between the CS department network and your router. The routing table format is as follows:

```
destination gateway mask interface
```

A valid `rtable` file may look as follows:

```
0.0.0.0 172.29.8.41 0.0.0.0 eth0
172.29.8.32 0.0.0.0 255.255.255.248 eth1
172.29.8.48 0.0.0.0 255.255.255.248 eth2
```

Note: 0.0.0.0 as the destination means that this is the default route; 0.0.0.0 as the gateway means that it is the same as the destination.

The VNL Server, on connection should return the IP addresses associated with each one of the interfaces. The output for each interface should look something like:

```
Router interfaces:
eth0    HWaddr82:6f:18:22:d3:ed
    inet addr 172.29.8.40
eth1    HWaddrb2:c2:4b:2b:cd:6d
    inet addr 172.29.8.38
eth2    HWaddr52:66:ea:bb:e9:11
    inet addr 172.29.8.54
```

To test if the router is actually receiving packets try pinging or running traceroute to the IP address of eth0 (which is connected to the gateway in the assignment topology). The `sr` should print out that it received a packet. What type of packet do you think this is?

What should your router do on receipt of an ARP request packet?

Developing Your Very Own Router Using the SR Stub Code

Data Structures You Should Know About

The Router (`sr_router.h`): The full context of the router is housed in the `struct sr_instance` (`sr_router.h`). `sr_instance` contains information about topology the router is routing for as well as the routing table and the list of interfaces.

Interfaces (`sr_if.c/h`): After connecting, the service will send the soft-host the hardware information for that host. The stub code uses this to create a linked list of interfaces in the router instance at member `if_list`. Utility methods for handling the interface list can be found at `sr_if.h/c`.

The Routing Table (`sr_rt.c/h`): The routing table in the stub code is read on from a file (default filename "`rtable`", can be set with command line option `-r`) and stored in a linked list of routing entries in the current routing instance (member `routing_table`).

The First Methods to Get Acquainted With

The two most important methods for developers to get familiar with are as follows:

in `sr_router.c`

```
void sr_handlepacket(struct sr_instance* sr,
        uint8_t * packet/* lent */,
        unsigned int len,
        char* interface/* lent */)
```

This method is called by the router each time a packet is received. The "packet" argument points to the packet buffer which contains the full packet including the ethernet header. The name of the receiving interface is passed into the method as well.

in `sr_vns_comm.c`

```
int sr_send_packet(struct sr_instance* sr /* borrowed */,
                        uint8_t* buf /* borrowed */ ,
                        unsigned int len,
                        const char* iface /* borrowed */)
```

This method will send an arbitrary packet of length, len, to the network out of the interface specified by "iface".

Dealing with Protocol Headers

Within the `sr` framework you will be dealing directly with raw Ethernet packets. There are a number of resources which describe the protocol headers in detail, including www.networksorcery.com, Stevens Unix Network Programming book, and the Internet RFC's for ARP (RFC826), IP (RFC791), and ICMP (RFC792). The stub code itself provides some data structures in sr_protocols.h which you may use to manipulate headers. There is no requirement that you use the provided data structures, you may prefer to write your own or use standard system includes.

Inspecting Packets with tcpdump

As you work with the `sr` router, you will want to take a look at the packets that the router is sending and receiving. The easiest way to do this is by logging packets to a file and then displaying them using a program called tcpdump.

First, tell your router to log packets to a file in a format that tcpdump can read by passing it the -l option and a filename:

```
./sr -t <topo-id> -l <logfile>
```

As the router runs, it will log the packets that it receives and sends (including headers) to the indicated file. After the router has run for a bit, use tcpdump to display the packets in a readable form:

```
tcpdump -r <logfile> -e -vvv -x
```

The `-r` switch tells tcpdump where to look for the logfile. `-e` tells tcpdump to print the headers of the packets, not just their payload. `-vvv` makes the output very verbose, and `-x` puts the packets in a hex format that is usually easier to read than ASCII. You may want to specify the `-xx` option instead of `-x` to print the link-level (Ethernet) header in hex as well.

The Application Server

Once you've correctly implemented the router, you can visit the web page located at `http://<server1-ip>:16280/`, `http://<server2-ip>:16280/`, `http://<server3-ip>:16280/`, and `http://<server4-ip>:16280/`. The application server also hosts a photo album, some music, an FTP service, and a simple UDP service. You will see how to access them when you get to the web page.

Remember: you are only able to access the VNL service and the application server from within CS department network.

Troubleshooting

You can view the status of VNL services: (substitute 87 with your topoid)

```
./vnltopo87.sh gateway status
./vnltopo87.sh vrhost status
./vnltopo87.sh server1 status
./vnltopo87.sh server2 status
./vnltopo87.sh server3 status
./vnltopo87.sh server4 status
```

If your topology does not work correctly, you can attempt to reset it: (substitute 87 with your topoid)

```
./vnltopo87.sh gateway run
./vnltopo87.sh server1 run
./vnltopo87.sh server2 run
./vnltopo87.sh server3 run
./vnltopo87.sh server4 run
```

Required Functionality

We will declare that your router is functioning correctly if and only if:

1. The router correctly handles ARP requests and replies.
2. The router responds correctly to ICMP echo requests.
3. The router correctly handles traceroutes through it (where it is not the end host) and to it (where it is the end host).
4. The router can successfully route packets between the gateway and the application servers.
5. The router maintains an ARP cache whose entries are invalidated after a timeout period (timeouts should be on the order of 15 seconds).
6. The router queues all packets waiting for outstanding ARP replies. If a host does not respond to 5 ARP requests, the queued packet is dropped and an ICMP host unreachable message is sent back to the source of the queued packet.
7. The router does not needlessly drop packets (for example when waiting for an ARP reply)
8. The router handles tcp/udp packets sent to one of its interfaces. In this case the router should respond with an ICMP port unreachable.
9. The router enforces guarantees on timeouts. Currently the stub code is event based. That is, code is executed each time a packet is received. This makest it hard to correctly enforce timeouts. For example, if the router is waiting for an ARP reply that doesn't come, it will have to wait for another packet to arrive before it can handle the timeout. Of course, if a packet never arrives, the timeout will never be serviced. You should implement a method of timing out ARP requests and ARP cache entries that can be guaranteed to function within a (relatively) fixed period, even if no more packets arrive at the router.