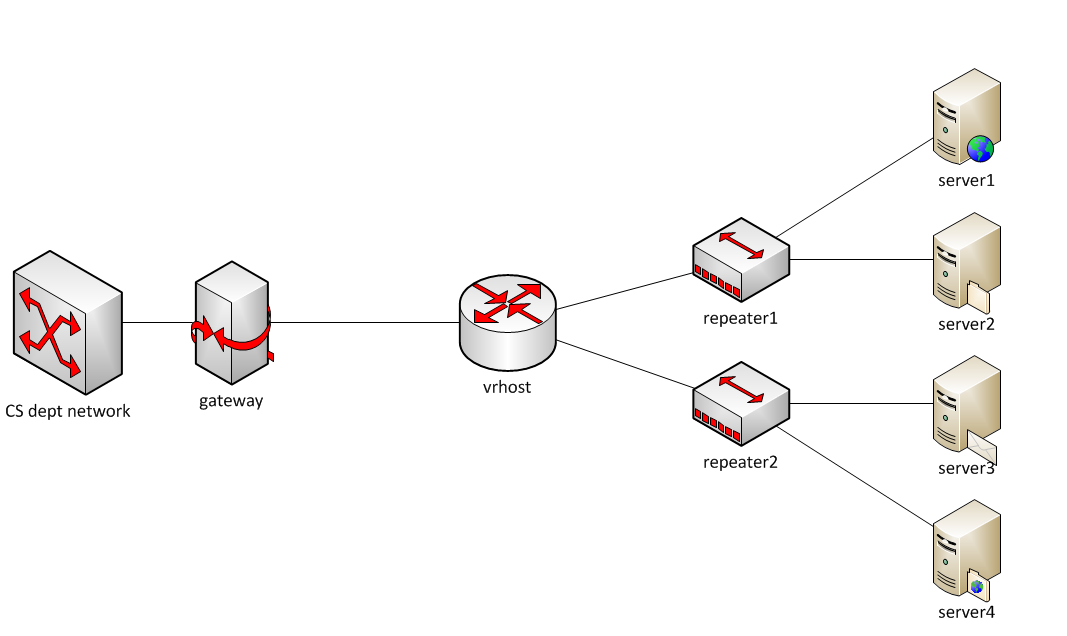
Project: A Simple IP Router

In this assignment you will implement a functional IP router that is able to route real Internet traffic. You will be given an incomplete router to start with. What you need to do is to implement the basic Address Resolution Protocol (ARP) and IP forwarding. A correctly implemented router should be able to forward traffic for any IP applications, including downloading files between your favorite web browser and a web server via your router.

# Overview



This is the network topology, and *vrhost* is the router that you will work on. Each group will be assigned one topology with specific IP addresses and Ethernet addresses to the network interfaces. The *vrhost* connects the CS department network to two internal subnets, each of which has two web servers in it. The goal of this project is to implement essential functionality in the router so that you can use a regular web browser in the CS network to access the web servers to download files.

***Note: This topology is only accessible from within the CS department network. You need to run your code on lectura or one of the machines in CS labs.***

In this assignment we provide you the skeleton code for a Simple Router (sr), which will connect to vrhost and launch itself, but doesn’t implement any packet processing or forwarding, which will be filled in by you.

~~The project is divided into two milestones. The first milestone implements half of the ARP protocol, and the second milestone implements the rest of ARP as well as IP forwarding.~~

# Test Driving the sr Stub Code

The first thing is to get familiar with the sr stub code. Download the stub code tarball from D2L and save it locally. You also need a user package tarball which is attached to your assignment email.

To run the code, untar the code package *tar xvf stub\_sr\_vnl.tar*, which will generate a ***stub\_sr*** directory. Unpack the user package *tar xvf vnltopo\*.tar*, move all the user package files into the ***stub\_sr*** directory. Compile the code by *make*. Once compiled, you can start the router as follows:

./sr -t topID

where topID is the topology ID assigned to you. After sr is started, it will connect to vrhost, and start running as the router’s software.

Another command-line option that may be useful is -r routing\_table\_file, which allows you to specify the routing table to load. By default, it loads the routing table from file *rtable*, which is specific to your topology and included in the user package.

You can also use -h to print the list of acceptable command line options.

The router has three network interfaces, all of which and their Ethernet and IP addresses are stored in a linked list, and the head of the list is member *if\_list* of *struct sr\_instance*.

The routing table is read from the file rtable. Each line of the file has four fields:

Destination, gateway(i.e., nexthop), mask, and interface.

A valid rtable file may look as follows:

0.0.0.0 172.24.74.17 0.0.0.0 eth0

172.24.74.64 0.0.0.0 255.255.255.248 eth1

172.24.74.80 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 the nexthop address is the same as the destination address of the incoming packet.

Upon start the router will print out its interface information as something like this:

Router interfaces:

eth0 HWaddrc6:31:9f:bb:4b:6e

inet addr 172.29.0.9

eth1 HWaddrcb:6c:4f:12:a5:2d

inet addr 172.29.0.10

eth2 HWaddr85:e4:4d:99:e1:2c

inet addr 172.29.0.12

To test if the router and the topology are set up correctly, try access one of the web servers by running the following command from a CS machine:

*wget* [*http://ServerIP:16280*](http://ServerIP:16280)

where ServerIP is the IP address of one of the servers in your topology, and all the web servers are run on port 16280. If the sr prints out that it has received a packet, your stub code and topology are properly set up. However, the sr will not do anything with the packet, so wget will not get any reply and will eventually time out.

# Developing your router using the Stub Code

**Important Data Structures**

The Router (sr\_router.h): The full context of the router is housed in the struct sr\_instance. It contains information about the routing table and the list of interfaces.

Interfaces (sr\_if.c, sr\_if.h): The stub code creates a linked-list of interfaces, if\_list, in the router instance. Utility methods for handling the interface list can be found in sr\_if.h, sr\_if.c. Note that IP addresses are stored in network order, so you shouldn’t apply htonl() when copying an address from the interface list to a packet header.

Routing Table (sr\_rt.c, sr\_rt.h): The routing table in the stub code is read from a file (default filename "rtable", can be set with command line option -r) and stored in a linked-list, *struct sr\_rt \* routing\_table*, as a member of the router instance.

**The First Methods to Get Acquainted With**

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

in sr\_router.c

void sr\_handlepacket(struct sr\_instance\* sr,

uint8\_t \* packet /\* lent \*/,

unsigned int len,

char\* interface /\* lent \*/)

This method is invoked each time a packet is received. The \*packet points to the packet buffer which contains the full packet **including** the Ethernet header (but without Ethernet preamble and CRC). The length of the packet and the name of the receiving interface are also 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 allows you to send out an Ethernet packet contained in the \*buf buffer and of certain length ("len"), via the outgoing interface "iface". Remember that the packet buffer needs to start with an Ethernet header.

***Thus the stub code already implemented receiving and sending packets. What you need to do is to fill in sr\_handlepacket( ) with packet processing logic that implements ARP and IP forwarding.***

**Downloading Files from Web Servers**

Once you've correctly implemented the router, you can visit the web page located at <http://ServerIP:16280/> by using GUI browser, text-based browser like lynx, or command-line tools such as curl and wget, from a CS department machine. “ServerIP” is the IP address of one of your servers.

**Dealing with Protocol Headers**

Within the sr framework you will be dealing directly with raw Ethernet packets, which includes Ethernet header, IP header and the payload. There are a number of online resources which describe the protocol headers in detail. For example, find IP, ARP, and Ethernet on [www.networksorcery.com](http://www.networksorcery.com/enp/Protocol.htm). The stub code itself provides data structures in sr\_protocols.h for IP, ARP, and Ethernet headers, which you can use without defining your own.

With a pointer to a packet (uint8\_t \*), you can cast it to an Ethernet header pointer (struct sr\_ethernet\_hdr \*) and access the header fields. Then move the pointer pass the Ethernet header and cast it again to a pointer to ARP header or IP header, and so on. This is how you access different protocol headers in a packet.

**Inspecting network traffic with tcpdump/wireshark**

Probably the most important network debugging tool is packet sniffer, e.g., tcpdump and wireshark (which has graphical interface). A packet sniffer captures packets from the wire and examines their contents. As you work with the sr router, you will want to take a look at the packets that the router is sending and receiving on the wire. This is done by logging network traffic to a file and then displaying them using tcpdump or wireshark.

First, tell your router to log packets to a file in the so-called pcap format:

./sr -t topID -l logfile

As the router runs, it will record all the packets that it receives and sends out in the file named “logfile.” After stopping the router, you can use tcpdump to display the contents of the logfile:

tcpdump -r logfile -e -vvv –xx

The -r switch tells tcpdump to read “logfile”, -e tells tcpdump to print the headers of the packets, not just the payload, -vvv makes the output very verbose, and -xx displays the content in hex, including the link-level (Ethernet) header.

***NOTE: in order to use tcpdump in lectura, you need to make a copy of /usr/sbin/tcpdump in your local directory and execute this local copy.***

***Learn to read the hexadecimal output from tcpdump. It shows you the packet content including the Ethernet header and IP header, which helps you debug. For example, you can see how a correctly formatted ARP request (coming from the gateway) looks like, and check which part of your ARP request/reply packet might have problem.***

**Troubleshooting of the topology**

You can view the status of your topology nodes: (substitute 87 with your topology ID)

./vnltopo87.sh gateway status

./vnltopo87.sh vrhost status

./vnltopo87.sh server1 status

./vnltopo87.sh server2 status

If your topology does not work correctly, you can attempt to reset it: (substitute 87 with your topology id), or notify the TA.

./vnltopo87.sh gateway run

./vnltopo87.sh server1 run

./vnltopo87.sh server2 run

# First Step: Answering ARP Requests

When the router is running and you initiate web access to one of the servers, the very first packet that the router receives will be an ARP request, sent by the gateway node to the router asking the Ethernet address of the router’s.

In this step, you need to

* Get familiar with the stub code
* Process the ARP request
* Send back a correct ARP reply
* Receive the next packet from the gateway, which should be a TCP SYN packet going to one of the web servers.

If your ARP reply is correct, you will receive a new, different packet from the gateway. Otherwise the same ARP request will be repeatedly sent to your router. The correct behavior can also be verified by logging the packets and viewing them with tcpdump/wireshark.

# Second Step: Implementing a working router

In this step, you’ll continue to implement the rest of ARP and the basic IP forwarding. More specifically the required functionalities are listed as follows:

1. The router correctly handles ARP requests and replies. When it receives an ARP request, it sends back a correctly formatted ARP reply. When it forwards a packet to the nexthop but doesn’t know the nexthop’s Ethernet address, it sends an ARP request and parse the returned ARP reply to get the Ethernet address.
2. The router maintains an ARP table: once it learns the Ethernet address for a given IP address, it saves this information, and will reuse it next time when sending packets to the same IP.
3. The router can successfully forward packets between the gateway and the application servers. When an IP packet arrives, the router should do the following:
   1. Look up the routing table to find out the IP address of the nexthop.
   2. Check ARP table for the Ethernet address of the nexthop. If the Ethernet address is already known, send the packet to nexthop.
   3. If the ARP table doesn’t have the Ethernet address, the router should send an ARP request to get the Ethernet address.
   4. While the router is waiting for ARP reply, it should buffer incoming packets that go to the same nexthop. After the ARP reply is received, save the information in ARP table, and send out all the packets that are waiting for the ARP reply.
4. Using the router, you can ping every server in the topology and download files from any of them.

The IP checksum algorithm as well as sample source code is in the Peterson & Davie textbook, page 95. A good way to test if your checksum function works is to apply it on an arriving packet. If your calculation gives the same checksum as the one carried in the incoming packet, then your code is correct. Remember to zero out the checksum field when you feed the packet to your checksum calculation. If the checksum is wrong, tcpdump will point it out when displaying the packet.

# Testing, Grading, and Submission.

**The project will be graded on a different topology. Don’t hardcode anything about your topology (e.g., IP and Ethernet addresses) in the source code.**

Your code must be written in C and use the Stub Code.

We will test your code as follows:

1. Ping the web servers from a CS machine. We expect 0% loss.
2. Download files from a web server. E.g.,

wget <http://ServerIP:16280> (this retrieves the web front page from the server.)

wget <http://ServerIP:16280/64MB.bin> (this retrieves a 64MB file.)

Download speed is not required. But if it is too slow, usually it means something is wrong with the code.

1. Log and examine the traffic. In particular, we will check that ARP behavior is correct, e.g., ARP request/reply are sent/received correctly, ARP is not sent for every packet because the previous results are saved in the ARP table.

**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.

~~The total score of the project is 100 points with the distribution of Milestone 1 (30 pts), and Milestone 2 (70 pts).~~

# 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 partially works.

3. Create a tarball

cd topXX

make clean

cd ..

tar -zcf topXX.tgz topXX

4. Upload topXX.tgz onto D2L dropbox.

# Deadlines

**Wednesday May 6, at 11:59pm.**