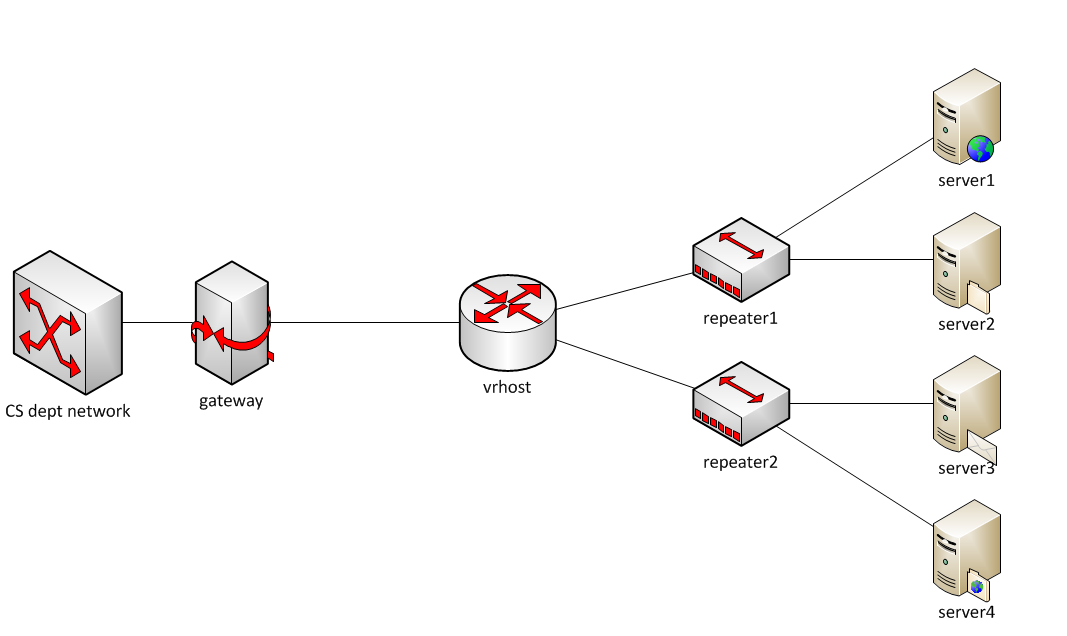
Build Your Own Router

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

# Overview



This is the topology of the network that you will work on. The node vrhost is the router that will run your code. The two repeaters are for configuration purposes and can be ignored. The servers run nginx web servers to serve some web pages and files to be download.

Each student will be assigned one topology, which is separated from others.

* ***Make sure to use only the topology assigned to you***
* ***This topology is only accessible from within the CS department. You need to run your router software and tests on Lectura.***

# Test Driving the Router Stub Code

Download the stub code tarball from D2L and untar it

*tar xvf stub\_sr.tar*

This will generate a ***stub\_sr*** directory that contains all the code.

You will also receive an assignment email with an attachment, top#.tar, where top# is the topology number, such as 301.

Save this tarball in stub\_sr and untar it:

*tar xvf top#.tar*

The resulting files must be in the stub\_sr folder. These files include:

* rtable: the static routing table for your router
* vnltopo#.iplist: the list of IP addresses for all the interfaces in your topology.
* vnltopo#.sh: a shell script to configure some aspects of the topology.

Compile the code: *make*

Run the router: *./sr -t top#*

***Topology No: 311***

You should see something like this, which means sr is now successfully running on vrhost:

Using VNS sr stub code revised 2010-01-21 (rev 0.23)

Loading routing table

---------------------------------------------

Destination Gateway Mask Iface

0.0.0.0 172.29.9.169 0.0.0.0 eth0

172.29.9.160 0.0.0.0 255.255.255.248 eth1

172.29.9.176 0.0.0.0 255.255.255.248 eth2

---------------------------------------------

Client bzhang connecting to Server 171.67.71.18:3250

Requesting topology 313

Virtual Network Lab, connection open

/home/vnl/topo/313.sh: line 250: kill: (3251) - No such process

successfully authenticated as bzhang

Router interfaces:

eth0

hardware address 52:5e:35:95:7d:4a

ip address 172.29.9.168

mask 255.255.255.254

eth1

hardware address 62:24:78:cc:7b:99

ip address 172.29.9.166

mask 255.255.255.248

eth2

hardware address d2:41:af:c3:48:67

ip address 172.29.9.182

mask 255.255.255.248

<-- Ready to process packets -->

On a separate terminal, ping any one of the servers in your topology. (see the #.iplist file)

*ping server\_ip\_address*

ip\_gateway\_eth0=172.29.9.105

ip\_vrhost\_eth0=172.29.9.104

ip\_vrhost\_eth1=172.29.9.102

ip\_vrhost\_eth2=172.29.9.118

ip\_server1\_eth0=172.29.9.97

ip\_server2\_eth0=172.29.9.98

ip\_server3\_eth0=172.29.9.115

ip\_server4\_eth0=172.29.9.116

On the router terminal, you should see output like:

\*\*\* -> Received packet of length 42

\*\*\* -> Received packet of length 42

\*\*\* -> Received packet of length 42

This is sr reporting that it has received a packet. Since it doesn’t have any packet processing capability yet, all these packets are dropped. Therefore on the ping terminal, you should see messages like

From 192.168.56.24 icmp\_seq=1 Destination Host Unreachable

From 192.168.56.24 icmp\_seq=2 Destination Host Unreachable

From 192.168.56.24 icmp\_seq=3 Destination Host Unreachable

Use ctrl-c (^c) to terminate sr and ping.

Your job in this project is to implement functionality necessary for ping and web downloads to work.

# Important functions and data structures

**First Two Functions**

The two most important methods 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 \*sr pointer allows access to this router’s interfaces and routing/forwarding table. The \*packet points to the buffer containing the incoming packet. It includes the Ethernet header (but without Ethernet preamble and CRC) and everything else above Ethernet. 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 (”buf”) of certain length ("len"), via a particular outgoing interface "iface". Remember that the packet buffer needs to start with an Ethernet header.

In the stub code, sr\_handlepacket( ) does nothing but printing a message on stdout. Your code should be added into this function to process and forward packets.

**Important Data Structures**

The context of the router is housed in struct sr\_instance defined in file sr\_router.h. It has two pieces of important information: the router’s routing table (\*routing\_table) and a list of its interfaces (\*if\_list).

Interface:

In our topology, the router has three interfaces. Each interface struct sr\_if (in sr\_if.c, sr\_if.h) contains its hardware address (i.e., Ethernet address), IP address, mask, and its interface name (i.e., eth0, eth1, eth2). At the start time, sr populates the interface list for your code to use. You may find sr\_get\_interface( ) and sr\_print\_if( ) useful. Note that IP addresses are stored in *network order*, so you don’t need htonl( ) when copying an address from the interface list to a packet header.

Routing Table:

The routing table (sr\_rt.c, sr\_rt.h) in this project is read from a file “rtable” at start time. A table has multiple rows/entries and looks like this:

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

The four fields, from left to right, are

dest, gw(i.e., nexthop), mask, and interface. See struct sr\_rt in sr\_rt.h.

Table Lookup:

The “dest” and “mask” together define the *destination network*, or a range of IP addresses. If a packet’s destination address (dest-addr) is within the destination network, i.e.,

*dest & mask == dest-addr & mask*

Then this table entry *matches* this packet’s destination.

If there’re more than one matching entry for a packet, only the longest prefix match should be used in packet forwarding. In other words, the matching entry whose mask is the largest number should be used.

You can see that the first entry in the table above will match any destination address, but its mask is the smallest possible. This entry is called the *default route*, meaning that it will be used only if there is no other matching entry.

IP Packet Forwarding

The matching entry tells where a packet should be forwarded to: the nexthop (gw) node at the outgoing interface. For example, the first entry in the table says packets should be forwarded to 172.24.74.17 via interface eth0, i.e., towards the CS department network.

The next two entries in the table have 0.0.0.0 in the gw field. That means the nexthop node is the final destination of the packet, and the destination address in the packet header should be used as the nexthop.

When a packet is being forwarded, its source and destination IP addresses should remain unchanged. Its Ethernet header, however, need to use the nexthop’s Ethernet address in the destination field, and the outgoing interface’s Ethernet address in the source field. This allows the packet to be received by the nexthop node.

How do you get the nexthop’s Ethernet address given the routing table only records its IP? By running the address resolution protocol (ARP), you can learn a node’s Ethernet address given its IP address.

**Dealing with Packet Headers**

In this project the router needs to process various protocol headers. The packet formats are summarized here: [Ethernet](http://www2.cs.arizona.edu/classes/cs525/papers/Ethernet.pdf), [ARP](http://www2.cs.arizona.edu/classes/cs525/papers/ARP.pdf), [IP](http://www2.cs.arizona.edu/classes/cs525/papers/IP.pdf), and [ICMP](http://www2.cs.arizona.edu/classes/cs525/papers/ICMP.pdf). You only need to process a small number of essential fields, not all the protocol details. The [Peterson & Davie book](https://book.systemsapproach.org/index.html) has more explanations of the protocols, and there are plenty of material online if you need further information.

The stub code provides data structures in sr\_protocols.h for IP, ARP, and Ethernet headers. The Linux system also defines data structures for various protocols in /usr/include/net and /usr/include/netinet.

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

When you read and/or print header fields, be careful about the [byte order problem](http://www2.cs.arizona.edu/classes/cs525/papers/byteorder.pdf) (big-endian vs. little-endian).

Avoid using C str\* functions since they are intended for strings that end with ‘\0’. Use memcpy() to copy multi-byte fields or assign values byte by byte.

# Implementing required functionality

**ARP**

The very first packet that will arrive at the router is most likely an ARP request. The EtherType field in the Ethernet header denotes it’s an ARP packet, and the field in the ARP header tells it’s a request or reply message.

An ARP request carries an IP address in the header and asks for the corresponding Ethernet address. Your router needs to check whether the IP address is one of the router’s, and if it is, the router should send an ARP reply containing the Ethernet address.

Often times your router also needs to know the Ethernet addresses of other nodes. It should compose its own ARP request, send it out, receive the ARP reply, and process it.

***While the router is waiting for ARP reply, there may be more data packets arriving. The router should buffer these packets while waiting for ARP, and send them all after getting the ARP reply.***

You can create your ARP packet from scratch, or reuse the received ARP packet and change the value of the fields. Either way works, but you need to make sure that the values of all the fields are correct; otherwise other nodes will not be able to understand the packet.

To avoid sending ARP request for every packet, the router should implement an ***ARP cache***:

* once it learns the Ethernet address for a given IP address, it saves the result and reuses it next time when sending packets to the same nexthop.
* The cached result should be discarded after 10s. You don’t need to be accurate by using a signal or timer. You can record the time (gettimeofday()) when the result is saved, and every time before using the cached result, check whether 10s have passed. If yes, remove the cached result and send an ARP request.

**IP**

If the incoming packet is an IP packet (based on EtherType), the router needs to carry out the following steps:

1. If the destination IP is one of the router’s,
   1. If the packet is an ICMP echo request, the router should respond with an ICMP echo reply. (See ICMP section below).
   2. Otherwise discard the packet, i.e., return from the function without further processing.
2. Decrement TTL by 1.
   1. If the result is 0, discard the packet.
   2. Otherwise, calculate header checksum and save the result to the checksum field.
3. Use the IP destination address to look up the routing table, find the matching entry to be used for packet forwarding.
4. Based on the matching entry, send the packet to the nexthop via the interface
   1. Obtain nexthop’s Ethernet address from ARP cache, or send an ARP request if it’s not in the cache.

The Peterson & Davie book has [IP header checksum algorithm and sample code](https://book.systemsapproach.org/direct/error.html#internet-checksum-algorithm) that you can use. One way to test if your checksum implementation is correct is to run it on an arriving packet. If you get the same checksum that is already contained in the packet, then your function is working. Remember to zero out the checksum field before feeding the packet to the checksum calculation. If the checksum is wrong, tcpdump/wireshark will point it out when displaying the packet.

**ICMP**

In order to be able to ping your router, it needs to support ICMP echo request/reply. You can tell it’s an ICMP packet by the “protocol” field in IP header, and whether it’s an echo request by “Type” in ICMP header.

When the router receives an ICMP echo request destined to the router itself, it should change the Type to echo reply, update the checksum (which covers the entire ICMP message starting from the Type field), and send it back to the original sender. Keep the rest of the ICMP part unchanged. Note that you also need to take care of the IP and Ethernet header.

**Inspecting network traffic with tcpdump/wireshark**

Probably the most important network debugging tool is a packet sniffer, e.g., tcpdump and [wireshark](https://www.wireshark.org/) (which has GUI and is easier to use). A packet sniffer captures packets from the wire and displays their contents. Working on the sr router, you may often want to see what is exactly transmitted on the wire. This is done by logging network traffic to a file and then displaying them using tcpdump or [wireshark](https://www.wireshark.org/).

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

./sr -t top# -l logfile

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

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 on lectura, you need to make a copy of /usr/sbin/tcpdump in your local directory and run this local copy.*** Or you can download the log file to local machine and use wireshark to read it.

***Learn to read the hexadecimal output from tcpdump. It shows you the packet content including the Ethernet header and IP header, which will be very helpful in debugging.*** For example, you can see how a correctly formatted ARP request (coming from other nodes) looks like, and compare it with the ARP packet generated by your code to see whether there may be an issue or not.

# Testing

A correctly implemented router is expected to pass the following tests:

1. Ping all the interfaces of the router results in 0% loss and delay < 10ms.
2. Ping all server interfaces results in 0% loss and delay < 10ms.
3. Examine the traffic log to verify correct ARP behavior, including the use of ARP cache and the timeout of ARP cache.
4. Successful retrieval of a web page:

*wget http://server\_IP:16280*

1. Successful retrieval of a large web object:

*wget http://server\_IP:16280/64MB.bin*

Optionally you can use a web browser to access the web pages hosted on the servers. Lectura has a text-based browser, “lynx”, that you can use. To use a browser on your own computer, you need to set up a SOCKS proxy so that traffic will go through lectura:

* On your local machine, run an ssh tunnel for SOCKS proxy:
  + *ssh -D 1080 lectura.cs.arizona.edu*
* Configure your browser to use the proxy. For example, on Firefox, open network settings, choose manual proxy, SOCKS, 127.0.0.1, port 1080.
* Access http://server\_IP:16280

**Troubleshooting of the topology**

Normally you shouldn’t need to use any of the following commands. But just in case, they’re listed here for your information.

You can view the status of your topology nodes: (replace 87 with your topology number)

./vnltopo87.sh gateway status

./vnltopo87.sh vrhost status

./vnltopo87.sh server1 status

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

./vnltopo87.sh gateway run

./vnltopo87.sh server1 run

# Grading

**The project will be graded on lectura using a different topology.**

**DO NOT hardcode anything about your topology (e.g., IP address, Ethernet address) in your code.**

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

You can work by yourself or in a group of two students.

**Grading is based on functionality**, i.e., what works and what doesn’t, **not the source code**.

# Submission

**Only one submission per group.**

1. Rename your working directory to be “topXX”, where XX is the topology number. E.g., top301.

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 if your router only works partially, you would want to explain what works and what not in README.txt to help the instructor/TA determine partial credit.

3. Create a tarball

cd topXX

make clean

cd ..

tar -zcf topXX.tgz topXX

4. Upload topXX.tgz to Gradescope/D2L.

# Deadline

**Friday Oct 6, 2023, at 11:59pm.**