COMPSCI 356: Computer Network Architecture

Spring 2015

COMPSCI 356 Computer Network Architectures: IP Routing+Forwarding with RIP

Introduction

In this assignment you will be constructing a **Virtual IP Network** using UDP as the link layer. Your network will support dynamic routing. Each node will be configured with its (virtual) links at startup and support the activation and deactivation of those links at run time. You will build a simple routing protocol over these links to dynamically update the nodes' routing tables so that they can communicate over the virtual topology. The relevant class lectures and textbook will be especially helpful with this part of the project. This is a 2-person group project. You should find a partner to work with right away, and email TAs to inform us of your pairing. If you are having problems with this (there could be an odd number of people in the class), post something on Piazza.

Requirements

Before you start coding, you need to understand what you're doing. It will take a little while to wrap your head around, but once you do, it will seem straightforward, we promise.

There are a two main parts to this assignment:

- The first is **routing**, the process of exchanging information to populate the routing tables you need for forwarding.
- The Second is IP in UDP encapsulation, and the design of **forwarding** --- receiving packets, delivering them locally if appropriate, or looking up a next hop destination and forwarding them.

Your network will be structured as a set of cooperating processes. You might run several processes on a single machine or use separate machines; it doesn't matter because your link layer is UDP. Files you will need to bootstrap your network are available in IP-Home. You will write a network topology file (we've supplied two examples) describing the virtual topology of your intended network.

Implementation Details

Your nodes will come up, and begin running RIP on the specified links. Each node will also support a simple command line interface, described below, to bring links up and down, and send packets. When IP

packets arrive at their destination, if they aren't RIP packets, you should simply print them out in a useful way. You will use UDP as your link layer for this project. Each node will create an interface for every line in its links file --- those interfaces will be implemented by a UDP socket. All of the virtual IP packets it sends should be directly encapsulated as payloads of UDP packets that will be sent over these sockets. You must observe an Maximum Transfer Unit (MTU) of 1400 bytes; this means you must never send a UDP (link layer) packet larger than 1400 bytes. However, be liberal in what you accept. Read link layer packets into a 64KiB buffer, since that's the largest allowable IP packet (including the headers). To enforce the concept of the network stack and to keep your code clean, we require you to provide an abstract interface to your link layer rather than directly make calls on socket file descriptors from your forwarding code. For example, define a network interface structure containing information about a link's UDP socket and the physical IP addresses/ports associated with it, and pass these to functions which wrap around your socket calls.

Routing - RIP

The second part of this assignment is implementing routing using the RIP protocol described in class, but with some modifications to the packet structure.

You must adhere to the following packet format for exchanging RIP information.

```
uint16_t command;
uint16_t num_entries;
struct {
uint32_t cost;
uint32_t address;
} entries[num_entries];
```

`command` will be 1 for a request of routing information, and 2 for a response. `num\_entries` will not exceed 64 (and must be 0 for a request command). `cost` will not exceed 16; in fact, we will define infinity to be 16. `address` will be an IPv4 address.
As with all network protocols, all fields must be sent on the wire in network byte order.
Once a node comes online, it must send a request on each of its interfaces. Each node must send periodic updates to all of its interfaces every 5 seconds. A routing entry should expire if it has not been refreshed in 12 seconds\footnote{When testing your project, feel free to make these times longer if it assists with using a debugger.}. If a link goes down, then the network should be able to recover by finding different routes to nodes that went through that link.
You must implement split horizon with poisoned reverse, as well as triggered updates.

Forwarding - IP

You will design a network layer that sends and receives IP packets using your link layer. The IP packet header is available in `/usr/include/netinet/ip.h` as`struct ip`. Those of you not using C/C++ may use `/usr/include/netinet/ip.h` or other sources as a reference for crafting your headers RFC 791, the IPv4 specification, would be a good place to start!}. Although you are not required to send packets with IP options, you must be able to accept packets with options (ignoring the options). You network layer will read packets from your link layer, then decide what to do with the packet: local delivery or forwarding. You will need an interface between your network layer and upper layers for local delivery. In this project, some of your packets need to be handed off to RIP, others will simply be printed. These decisions are based on the IP protocol field. Use a value of 200 for RIP data, and a value of 0 for the test data from your send command, described below.
Even without a working RIP implementation, you should be able to run and test simple forwarding, and

local packet delivery. Try creating a static network (hard code it, read from a route table, etc.) and make sure that your code works. Send data from one node to another one that requires some amount of forwarding. Integration will go much smoother this way.
Remember to:

- (Re-)Calculate IP checksum: using the files here
- Decrement the TTL

Your Code

Input

Your program, let's call it `node`, must take in one file as commandline input. Below is the format of the file:

- The first line of the file specifies the IP and port for this node: "[IP-address]:[port]" e.g. localhost:17000
- Every line after the first line specifies an interface on this node: [IP-address-of-remote-node]:[port-of-remote-node] [VIP of my interface] [VIP of the remote node's inteface]" e.g. "localhost:17001 10.116.89.157 10.10.168.73"

To create a network with 3 nodes: Node A, Node B, Node C. I would start up 3 instances of your program, each with a different input file. Below, you find examples of such input files.

- **Node A's Input:**
  localhost:17000
  localhost:17001 10.116.89.157 10.10.168.73

- **Node B's Input:**
  localhost:17001
  localhost:17000 10.10.168.73 10.116.89.157
  localhost:17002 10.42.3.125 14.230.5.36

- **Node C's Input:**
  localhost:17002
  localhost:17001 14.230.5.36 10.42.3.125

Using these input files, each copy of your program will determine its link information. These files mean that Node A has one interface defined by a pair of tuples, the IP ``localhost'' and port 17000 and the IP ``localhost'' and port 17001. The interface's virtual IP is 10.116.89.157. It is connected to another interface (defined by the reversed tuple) with virtual IP 10.10.168.73.

Output

Your program, let's call it `node`, must support the following commands.

- **ifconfig** Prints information about each interface, one per line. The print out for each interface should have this format: "[inteface_id]\t[interface_vip]\t[status]" e.g. "4 12.23.34.23 up". Where status is "up" or "down", signifying if the interface is active or inactive.

- **routes** Print information about the route to each known destination, one per line. The print out for each route should have this format: "[Destination]\t[Next_hop_interface_id]\t[cost]" e.g. "12.12.43.23 2 5"

- **down** Brings an interface ``down''. The interface_id is the number you get from ifconfig.

- **up** Brings an interface ``up'' (it must be an existing interface, probably one you brought down).

- **send** Send an IP packet with to the virtual IP address vip (dotted quad notation). The payload is simply the characters of $string$ (as in snowcast, do not null-terminate this).

You should feel free to add any additional commands to help you debug or demo your system, but the above the commands are required.

All Nodes should be able to receive the command line. Let's take Node A as an example,

- If you type "ifconfig" into Node A, it should print out:
  1 10.116.89.157 up

- If you type "route" into Node A, it should print out:
  10.10.168.73 1 1
  10.42.3.125 1 1
  14.230.5.36 1 2

- If you type "down 2" into Node A, it should print out the following error because it has no interfaces with id 2:
  Interface 2 not found.

- If you type "up 1" into Node A, it should print out the following acknowledgement:
  Interface 1 up.

- If you type "down 1" into Node A, it should print out the following acknowledgement:
  Interface 1 down.

- If you type "send 10.10.168.73 how are you doing?" into Node A, Node B should print out "how are you doing?"

- Similarly, If you type "send 10.42.3.125 hey hey friend." into Node A, Node B should print out "hey hey friend."

- If you type "down 1" into Node A, it should print out the following acknowledgement:
  Interface 1 down.

Extra Credit

You must implement IP fragmentation for extra credit. The MTU for each link should be set as an additional command line interface option as:
[mtu integer0 integer1] Sets the MTU for the link integer0 to integer1 bytes.

How to Submit

You can finish this lab in groups of two. Submit a compressed file on Sakai, with **main.c**, and a README file containing your name in it.