CISC 4615 Data Communication & Networks

Group Programming Assignment 1: IP Routing and Forwarding

Introduction

In this assignment you will be constructing a Virtual IP Network using UDP packets. Your network will support dynamic routing. Each node will be configured with its (virtual) links at startup and support the updates, such as changing the cost, 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.

Requirements

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.

We will simulate a network of nodes with a set of cooperating processes running on the same host as you do in lab 1 and lab2. You will write a network topology file (we've supplied two examples) describing the virtual topology of your intended network.

Implementation Details

Static routing tables

In this setting, you node will be feeded with all topology files. By loading the files, each node must build a static routing table and forward the package based on this table.

For example, if you executable program is named node and there are 3 nodes in the network, a, b, c. To launch the nodes, you should start three processes,

```
./node NodeA NodeB NodeC
```

```
./node NodeB NodeA NodeC
```

```
./node NodeC NodeA NodeB
```

Please note that the first file consists of the configurations of the node itself. The following files are the configuration files of other nodes. By reading the configuration files, you should be able to generate a static routing table by using the Bellman-Ford Equations.

Each record in the table will be a struct of `forwarding_entry_t`

```
typedef struct forwarding_entry {
    char entry_src_addr[IP_ADDR_LEN];   // Your own IP address
    char dest_addr[IP_ADDR_LEN];            // Destination IP Address
    int interface_id;       // Interface ID associted with the IP
    int cost;                   // link cost
    time_t last_updated;    // last updated time
} forwarding_entry_t;
```

Then, your forwarding table should be,

```
typedef struct forwarding_table {
    int num_entries;
    forwarding_entry_t forwarding_entries[MAX_NUM_ROUTING_ENTRIES];
} forwarding_table_t;
```

Dynamic routing tables

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 the protocol 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. The interface struct is like,

```
typedef struct interface {
    int interface_id;
    char my_ip[IP_ADDR_LEN];
    uint16_t my_port;
    char my_vip[IP_ADDR_LEN];
    char other_vip[IP_ADDR_LEN];
    int mtu_size;
    bool is_up;
    int send_socket;
} interface_t;
```

All the interfaces should be stored in the table for management.

```
typedef struct ifconfig_table {
    int num_entries;
    interface_t ifconfig_entries[MAX_NUM_ROUTING_ENTRIES];
} ifconfig_table_t;
```

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 packet larger than 1400 bytes. However, be liberal in what you accept. Read link layer packets into a 64KB 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.

Dynamic Routing - RIP (Distance Vector)

The first 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. The IP address will be an IPv4 address.

Then, your RIP packet should be,

```
typedef struct rip_packet {
    uint16_t command;
    uint16_t num_entries;

    struct {
        uint32_t cost;  // link cost to the following IP address
        uint32_t address;   // Your neighbor's IP address
    } entries[MAX_NUM_ROUTING_ENTRIES];
} rip_packet_t;
```

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. 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 (use 16 as infinity), as well as triggered updates.

Forwarding - IP

You will design a network layer that sends and receives IP packets using your link layer. 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 (like what you do in lab2)

- Decrement the TTL

Functions (just suggestions)

1. void initialize_interface(interface_t * interface)
    - create a socket for each interface (just create, not specification)

2. void send_packet_with_interface(interface_t * interface, char * data, int data_size, struct iphdr * ip_header)
    - Construct the sockets with specification
    - Use the IP ip_header struct to contruct the full packet (IP header + data)
    - Send the packet

3. interface_t* get_interface_by_id(int id)
    - Given the ID, return the interface by searching your IFCONFIG_TABLE.

4. forwarding_entry_t* get_forwarding_entry_by_dest_addr(char * dest_addr)
    - Search your forwarding table and return a forwading table entry.

5. void create_ifconfig_entry(int ID, uint16_t port, char *myIP, char *myVIP, char *otherVIP)
    - Base on the information, create an entry of interface and add it into the table
    - In addition, you should call initialize_interface

6. void update_forwarding_entry(char * src_addr, char * next_addr, char * dest_addr, int cost)
    - You should decide how to update your forwarding entry (when you build the table from the configuration file and when you receive a RIP packet)

7. void build_tables(FILE *fp)
    - Read the configuration file and build interface table as well as forwarding table

8. bool is_dest_equal_to_me(char * dest_addr)
    - Check the dest_addr, process the message and print out a message to indicate it arrived.

9. void send*packet(char * dest*addr, char * msg, int msg_size, int TTL, int protocol)
    - Search the forwarding table to find the next hop (call get_forwarding_entry_by_dest_addr)
    - Search interface table by calling get_interface_by_id() function to determine which interface should be used
    - Construct the IP_header
    - Call send*packet*with_interface() to send it out

10. void set_as_up(int ID, int Cost)

- Update the interface table
- Update the forwarding table

11. void set_as_down(int ID)
    - Update the interface table
    - Update the forwarding table to mark the appropriate link to infinity (cost = 16)

12. void print_routes()
    - Print out the forwarding table

13. void print_ifconfig()
    - Print out the interface table

14. void send_forwarding_update(char * dest_addr)
    - Construct RIP packet with command 2 to indicate it is a response a RIP request packet.
    - You need to call send_packet in this function

15. void activate_RIP_update()
    - If you have a RIP update (e.g. link cost change, or interface up/down), you should call send_forwarding_update() for all the interfaces.

16. void request_routes()
    - Construct a route request message
    - Send it to all your interfaces

17. void check*for*expired_routes()

    - Periodically call this function
    - Mark expired routes to infinity

18. void choose_command(char * command)
    - Extract the command the call appropriate functions
    - For example, if the user set an interface to "up", you should call `set_as_up(ID)` and `activate_RIP_update()`.

19. int init*listen*socket(int port, fd*set * running*fd_set)

    - Start listening on sockets.

20. void handle*packet(int listen*socket)

    - When you received a packet, you should call inet_ntop() function to convert message from binary to text form. `inet_ntop(AF_INET, &(recv_header->saddr),src_addr,INET_ADDRSTRLEN)`

- Whenever you received a message, you should decide how to handle it.
- For example, you should check the IP checksum and TTL.

```
if(received_ip_checksum != calculated_ip_checksum){
        printf("Broken checksum, dropping packet\n");
        return;
    }

if(recv_header->ttl <= 0) {
        printf("TTL surpassed, dropping packet\n");
        return;
    }
```

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 one 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] Cost

e.g. `localhost:17001 10.116.89.157 10.10.168.73 10`

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 10
- Node B's Input:
  - localhost:17001
  - localhost:17000 10.10.168.73 10.116.89.157 10
  - localhost:17002 10.42.3.125 14.230.5.36 9
- Node C's Input:

- localhost:17002
- localhost:17001 14.230.5.36 10.42.3.125 9

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:

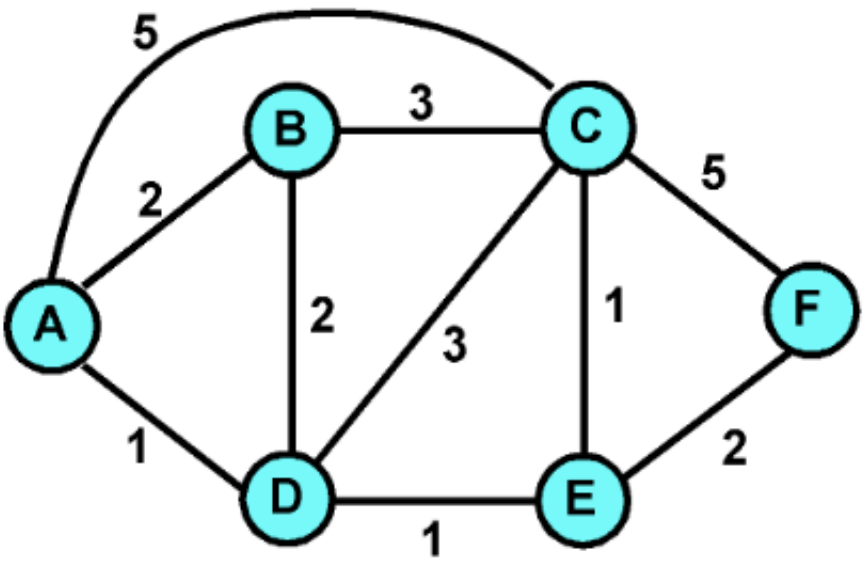
> 10.10.168.73 1 1
>
> 10.42.3.125 1 1
>
> 14.230.5.36 1 2

- 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

Add a command "update remote-ip-address cost" to update the link cost and the whole network will notice this update. You can test your program with a fairly complicated network topology. (e.g. Node c need 5 interfaces to connect with A, B, D, E, and F)



Grading Rubric

50% Static Routing

- 25% Build the tables
- 15% Search the routing table
- 10% Send the package

20% IP Forwarding

———————————————

- 5% — create well formed headers and obey protocol
- 5% — handle packet header (TTL, checksum)
- 5% — fwd using correct RIP entry
- 5% — Send test data

30% — RIP

——————————

- 5% — Base case arrive at stable with simple topology
- 5% — deal with taking links up/down — recalculate and converge
- 5% — split horizon (deal with count to infinity problem)
- 5% — triggered updates
- 5% — route timeouts
- 5% — print interface/routes

Extra Credit 15%