Project 2: IP over UDP

Due: 11:59 PM, Oct 16, 2017

Contents

1 Introduction **2**

2 Requirements **2**

2.1 Core Requirements 2

2.2 Capstone Requirements 2

3 Implementation **3**

3.1 Abstract Link Layer 3

3.2 Routing - RIP 4

3.3 Forwarding 4

3.4 Driver 5

3.5 Fragmentation (Capstone Only) 6

3.6 Traceroute (Capstone Only) 6

3.7 Multicast (Capstone Only) 6

4 Getting Started **7**

4.1 Executables 7

4.2 Sample Networks 7

4.3 Utilities for C 8

5 Getting Help **8**

6 Grading **9**

6.1 Milestone - 20% 9

6.2 Functionality - 65% 9

6.3 Code Quality - 10% 9

6.4 README - 5% 9

7 Handing In and Interactive Grading **10**

8 A Warning **10**

1 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 post your pairing on Piazza to inform the TAs 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 or ask us. Once the groups are set, you'll be assigned a mentor TA to help you through this project and the next, TCP. TCP will build on this project, so your effort on design will pay off twice.

2 Requirements

2.1 Core 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 two main parts to this assignment. The first 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. The second is *routing*, the process of exchanging information to populate the routing tables you need for forwarding. This will be done with the Routing Information Protocol (RIP) which will be shown in class, and is described in section 4.2.2 of the textbook.[1]

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 your github repository. You will write a network topology file (we've supplied three examples) describing the virtual topology of your intended network. After running our script `net2lnx` on the complete topology, you'll have a file for each node that specifies only that node's links. You will run your process, which must be called `node`, for each virtual node, and must accept the name of that node's link file as its first argument on the command line. An example invocation would be:

```
node <linksfile>
```

2.2 Capstone Requirements

In addition to everything mentioned in the `Core Requirements` section, there is a small additional requirement for students taking cs168 for a capstone requirement. Below is a list of additional features to implement, and each capstone student is required to implement one of them. So if both

[1]To be clear, RIP is an indepedent protocol apart from IP. IP as a protocol does not prescribe any particular routing method. Read RFC 2453, the specification for RIP version 2, for more information.

students in a pair working on IP are taking cs168 for a capstone credit, then 2 of the following need to be implemented:

a. **IP Packet Fragmentation**
Refer to RFC 791 for the proper Fragmentation and Reassembly procedures. You will be required to implement both procedures, except for the part about header options. Your packets should make it from endpoint to endpoint regardless of the intermediate nodes' MTUs (which can be anything between 1 and 1400).

b. **Traceroute** You will be implementing a subset of ICMP within your IP layer in order to implement traceroute functionality. The requirements are that you should be able to report the nodes in shortest path from the current node to any other endpoint in the network. This functionality requires you to implement the `ICMP Time Exceeded Message` from RFC 792.

c. **Multicast** You will be implementing a simple version of multicast addressing. Each node in the network can have any number of their interfaces `Join` and `Leave` a multicast group. When a host sends a packet to that multicast group, it is received by every host that has an interface subscribed to that group. Since multicast was designed for a different network topology (i.e: one with a gateway), we will not be abiding exactly by the specs. Instead, you will be responsible for developing your own protocol for communicating which nodes are in which multicast groups throughout your network.

3 Implementation

In brief, your nodes will come up, create an abstract link layer, 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. Finally, when IP packets arrive at their destination, if they aren't RIP packets, you will implement an interface to deliver them to an upper layer. In the next assignment, you will deliver them to your TCP implementation when appropriate. In this current assignment, you will simply print the packets out in a useful way.

From a network stack persective, you will implement a link layer interface over UDP sockets with the ability to disable and enable individual interfaces. You will then implement a virtual IP layer over these interfaces, and then build RIP as a client protocol that uses your virtual IP layer.

3.1 Abstract Link Layer

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 link layer frames 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 (link layer frame) larger than 1400 bytes. However, be liberal in what you accept. Read link layer frames 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. We also require that you provide the functionality to activate/deactivate a link layer interface; this would be equivalent to

```
ifconfig eth0 up/down
```

or disabling your Ethernet/wireless card.)

3.2 Routing - RIP

One part of this assignment is to implement 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:[2]

```
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[3]. 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. Triggered updates do not contain the entire routing table, just the routes that are updated.

3.3 Forwarding

In addition, 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[4]. Although you are not required to send packets with IP options, you must be able to accept packets with options (ignoring the options). Your network layer will read packets from your link layer, then decide what to do with the packet: local delivery or forwarding.

[2] If you are writing in C or C++, consider using flexible array members for allocation of your packet structure.
[3] When testing your project, feel free to make these times longer if it assists with using a debugger.
[4] RFC 791, the IPv4 specification, would be a good place to start!

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. Next time, you'll be handing packets off to your TCP implementation. 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. We ask you to design and implement an interface that allows an upper layer to register a *handler* for a given protocol number. We'll leave its specifics up to you. An example of how you might go about doing this in C (for some `some_data_t`):

```
typedef void (*handler_t)(some_data_t *, struct ip *);
void net_register_handler(uint8_t protocol_num, handler_t handler);
```

For example, for RIP packets, the RIP packet should be the payload for the IP packet. As a protocol, an RIP handler should be able to be registered with the following in order to receive incoming packets with an IP protocol field of 200:

```
net_register_handler(200, RIP_handler);
```

Likewise, RIP as a protocol should be able to send packets over a particular interface through IP. Keep in mind that this will require a clean abstraction of your link layer interfaces.

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.

3.4 Driver

Your driver program, `node`, will be used to demonstrate all features of the system. You must support the following commands within a command line interface.

interfaces,i Print information about each interface, one per line.

routes,r Print information about the route to each known destination, one per line.

down *integer* Bring an interface "down".

up *integer* Bring an interface "up" (it must be an existing interface, probably one you brought down)

send *vip proto string* Send an IP packet with protocol *proto* (an integer) 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).

q Quit the node by cleaning up used resources

You should feel free to add any additional commands to help you debug or demo your system, but the above the commands are required. It would be to your advantage to add bandwidth-intensive test commands to help prepare your implementation for TCP.

3.5 Fragmentation (Capstone Only)

Your driver should include the following commands for demonstrating your fragmentation capabilities:

setmtu *interface mtu* This will set the mtu for outgoing packets on the interface.

fragment *yes/no* This will set the DF flag on all outgoing packets. *yes* will enable fragmentation at this node, and *no* will disable it.

interfaces In addition to what this already does, you need to print the mtus alongside each interface

From these driver commands, we should be able to observe packets being dropped if the MTU of any node in the path is too low, and framentation is not supported. Additionally, if we enable fragmentation, the packets should not get dropped anymore.

3.6 Traceroute (Capstone Only)

Your driver should include the following command for demonstrating traceroute:

traceroute *vip* prints out the sequence of hops in the following format: `<hop num> <vip>`. So an example output would be:

```
Traceroute from 192.168.0.2 to 192.168.0.5:
1 192.168.0.2
2 192.168.0.8
3 192.168.0.5
Traceroute finished in 3 hops
```

From the driver command, we should be able to see changes in the path when any node in the network is brought up or down. If a host is not in the network, or is unreachable, you should print that information.

3.7 Multicast (Capstone Only)

Your driver should include the following commands for demonstrating multicast:

join *interface multi_ip* This will tell your IP to receive packets for the multicast group, *multi_ip*, on the specifiec interface.

leave *interface multi_ip* This will tell your IP to stop receiving packets for the multicast group, *multi_ip*, on the specifiec interface.

groups This will list all of the multicast groups that each of your interfaces are subscribed to.

From these driver commands, we should be able to have any number of nodes in your network join a multicast group, and all those nodes should receive messages designated for any multicast group they are assigned to. One requirement is that each node should only receive a multicast-addressed packet 1 time for every interface subscribed to that multicast group. So if a node only has 1 interface subscribed to the group `224.0.0.10`, then your node should only display packets sent to this group 1 time. If it has 2 interfaces subscribed to `224.0.0.10`, then any time a message is sent to that group, the node should display it twice.

4 Getting Started

We've created a few tools that you can use to help you with your project. They are available in the following github classroom invitation link:

`https://classroom.github.com/g/7yiS2pOm`

4.1 Executables

- `bin/ref_node` - The reference node that you should be able to communicate with.
- `bin/net2lnx` - A tool to convert a .net file into a series of .lnx files that each node can read separately.

4.2 Sample Networks

These are found in the nets/ subdirectory.

- `AB.net` - Simple network with three nodes. It may looks like this:

  ```
  node A localhost
  node B localhost
  node C localhost
  A <-> B
  B <-> C
  ```

  which tells you the physical location of each node and how they are connected. After running net2lnx on it, you will have something look like:

  ```
  A.lnx:
    localhost:17000
    localhost:17001 10.116.89.157 10.10.168.73
  B.lnx:
    localhost:17001
    localhost:17000 10.10.168.73 10.116.89.157
    localhost:17002 10.42.3.125 14.230.5.36
  C.lnx:
    localhost:17002
    localhost:17001 14.230.5.36 10.42.3.125
  ```

  which you can feed each node as their link information. These files mean that 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.

- `loop.net` - More complicated network with the following shape:

```
src -- srcR --  short  -- dstR -- dst
        |                   |
        \-- long1 -- long2 -/
```

A useful test for routing is to start the network and make sure `src` goes to `dst` through `short`. Then stop the `short` node and see what happens.

4.3 Utilities for C

We've provided several utility files for C with useful functions in `main.c` and the `support` directory:

- Debugging: `dbg.c dbg.h`. Print colored debugging messages. You can enable and disable categories of messages based on the environment variable `DBG_MODES`. See `node.c` for an example of how to use them in your code. By default, `node` enables only error messages. If you want to enable only, say, net layer and routing messages, then you can run:

  ```
  DBG_MODES=net,route ./node file.lnx
  ```

  See `dbg_modes.h` for a full list of debugging modes - feel free to add your own!

- IP checksum calculation: `ipsum.c`. Use this function to calculate the checksum in the IP header for you.
- Linked list: `list.h`.
- Hash table: `htable.c htable.h`.
- IP header: `ip.h`, equivalent to `<netinet/ip.h>`.
- parseLinks: `parseLinks.c parseLinks.c`, implementation of parsing the lnx file. Feel free to use it directly or modifying it.

5 Getting Help

This project isn't intended to be painful, and you have many resources to help you. Make sure you've read this handout and really understand what we mean when we say that UDP is your virtual network's link layer. Piazza is always a good place to get help on general topics, and the TAs will, of course, be holding TA hours and scheduling appointments.

Make sure that you work together with your group partner, and try to split the project up so that neither of you has too much to handle. An obvious way to split things up is for one person to implement routing (RIP) and the other to be responsible for everything else (packet forwarding, send/recv interface, etc), but you can do whatever you feel is appropriate. It will *not* be possible for you to go off into separate rooms, implement your half, and "just hook them up." You should work together, there is a lot that should be designed together. The routing table is the most obvious example.

We request you use Git well so that you can update each other periodically (commit often, but only when the build succeeds!). However, please note that your Git repos should be private, and

you are not allowed to share code with other groups. You can talk to other groups about concepts, algorithms, *etc.*, but each group's code must be their own.

Finally, each group will have a mentor TA. This means that you'll have one of the TAs as your group's advisor. You'll need to set up a milestone appointment to meet with your mentor TA during the first week of the project to discuss the project design. Once you've got the ok on this, you should stay in contact with your mentor, who will be grading your project and will be able to explain what the project ultimately should be doing. Your mentor also will do his best to help outside of TA hours, debugging, discussing design, *etc.* Just because your mentor is helping you out, however, doesn't mean that he/she is at your beck and call. Understand that the TA staff is busy too, and while they'll try to help you as much as possible, there may be times when they simply won't be able to.

6 Grading

6.1 Milestone - 20%

You will schedule a milestone design meeting with your TA by Monday the 9th. At this milestone, you should have a clear design for your program and be ready to ask questions you don't understand. Be ready to answer specific questions about your design, for instance:

- What objects will you use to abstract link layers, and what interface will it have?
- What fields in the IP packet are read to determine when to forward a packet?

6.2 Functionality - 65%

Most of your grade will be based on how well your program conforms to our specification. As in the Snowcast project, you will be expect to interoperate with the reference implementation as well as with other pairs' projects. Among other details, your program will be expected to maintain forwarding tables, handle packet forwarding, network loops, and maintain interfaces that may go up or down at any time without causing the network to crash.

6.3 Code Quality - 10%

Because part of the specification prescribes the programming interfaces for the link layer and for the upper layer handlers to IP, we will be evaluating the design of those interfaces as well as general code quality.

6.4 README - 5%

Please include a README that describes your design decisions, including how you abstract your link layer and its interfaces, the thread model for your RIP implementation, and the steps you will need to process IP packets. List any known bugs or notable design decisions. If you identify known bugs, we will take off less points than if we have to find them ourselves.

7 Handing In and Interactive Grading

Once you have completed the project you should commit and push your Git repo to deliver your code. Your mentor TA will arrange to meet with you for your interactive grading session to demonstrate the functionality of your program and grade the majority of it. This meeting will take place at some point shortly after the project deadline. Between the time you've handed in and the demo meeting, you can continue to make minor tweaks and bug fixes (and you should, since it will be the code base for your next project). However, the version you've handed in should be nearly complete since it could be referenced for portions of the grading.

8 A Warning

You should start on this project *now*. We expect all of the projects in CS168 to take the full amount of time we give you. It can be tricky so we want to make sure that you stay on top of it. The milestone design meeting is meant to encourage you to plan your design. Ask questions now if in doubt. Start talking with your partner right away, and get ready to get connected!

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS168 document by filling out the anonymous feedback form:

`https://piazza.com/brown/fall2017/cs168`.