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.

Teamwork  This is a 2-person assignment. To form a group, you should fill out the group assignment form, located here: https://forms.gle/2cqUUdrt9U41LCaUX6

If you prefer to work with a specific person, you can indicate it here—both students on a team should fill out the form, and only mutual requests will be honored. Otherwise, we will match each student to a partner based on their language preferences and in-person/remote status.

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.

Capstone students  If you are using this class to fulfill your capstone requirement, you will need to complete either of two extra components. If both students are taking the class as a capstone, both capstone components must be completed.

2 Development Environment

We will be providing a container-based development environment that you can use to work on this assignment, which you can use to run Wireshark. Wireshark is an industry-standard packet capture tool that you can use to observe network traffic in real-time and debug your work.

You can install the container environment on your own system and use this to develop locally. If developing on a non-department system would be problematic for you and you did not already indicate this on HW0, please email the TAs list ASAP and we will provide a solution. (If you already noted this on HW0, we will contact you.)

To install the container environment, follow the instructions here: https://cs.brown.edu/courses/csci1680/s22/pages/container_setup/

Otherwise, you can still work on this assignment on the department machines if you want, but this is not recommended as you will not have access to Wireshark. When submitting your work, please indicate which environment you used in your readme.
3 Requirements

3.1 Core Requirements

There are two main parts to this assignment:

- The first is IP in UDP encapsulation, and the design of forwarding—this includes 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. We will do this using RIP, the Routing Information Protocol (v2), a relatively simple distance-vector routing protocol described in RFC2453\(^1\). Our version of the protocol will be modified slightly from the RFC version, as described in Section 4.3.

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 some 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>
```

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

1. **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.

2. **Route Aggregation and Longest Prefix Match** In large networks it is necessary to aggregate the networks known to a router in order to keep the size of the routing tables reasonably small. You will be implementing a mechanism that automatically aggregates the networks a router learns from its neighbours. As a result of network aggregation, in order for your

routers to route correctly you will also need to implement Longest Prefix Matching. See the implementation section for more details.

4 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 perspective, 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.

4.1 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.

4.2 Forwarding

In addition, you will design a network layer that sends and receives IP packets using your link layer. Overall, your network layer will read packets from your link layer, then decide what to do with the packet: local delivery or forwarding.

In general, your IP forwarding process should follow the IPv4 protocol, as described in RFC791, but you are not required to implement all features of this standard, as described below. When
you receive a packet, your node must validate its checksum, inspect the destination address to
determine how it should be processed, and (if necessary) forward it to the appropriate next hop.

Your IP packets must use the standard IPv4 header format, described in Section 3.1 of RFC791. When forwarding, you must decrement the packet’s TTL value and recompute its checksum—you can use a starting TTL value of 16, which is more than sufficient for all the networks we will test in this assignment. A struct for 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.

Your node is NOT required to support advanced IP features such as fragmentation, IP potions. In addition, all other IP advanced header fields (IP Identification, Type of Service, flags) may be set to zero when you generate packets, as we will do not use any of these features. However, even though your node does not support these features, your design should be able to receive these packets and act on them in a sensible way. For example, you are not required to send packets with IP options, but you must be able to accept packets with options—you can simply ignore the options. Similarly, you are not required to send fragmented packets, or implement reassembly of fragmented packets that you receive. Instead, just forward these packets normally.

Overall, your IP node must be able to interoperate with the reference node when forwarding packets. Make sure you test your node in networks that contain the reference node. As you work to implement forwarding, remember that you have Wireshark to help you! You can use wireshark to view the contents of your IP packets (and those sent by the reference node) to make sure they are formatted properly. See the class lecture videos for more information on how to do this, or feel free to ask on EdStem.

Interface to higher layers 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, which we will call “test packets” sent using the send command, will simply be printed. In the next assignment, you will 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.

4.3 Routing with RIP

One part of this assignment is to implement routing using a modified version of RIP protocol, which is defined by RFC2453\(^3\). We will discuss the RIP protocol in detail during class within the first week of the assignment’s release. You can also read more about the RIP protocol in Sections 13.1–13.2 of the Dordal textbook\(^4\) or in Section 4.2.2 of the Peterson textbook.

Our version of RIP uses a slightly modified packet structure, which is described on the next page.

\(^3\)http://tools.ietf.org/html/rfc2453
RIP message format

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

```c
uint16_t command;
uint16_t num_entries;
struct {
    uint32_t cost;
    uint32_t address;
    uint32_t mask;
} 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. `mask` is a bitmask to denote the size of the network. In the standard version of the assignment, all routing entries will refer to single IP addresses and thus will use a mask of 255.255.255.255 (ie. all 1’s) to denote a /32. If you are implementing link aggregation (capstone only), you will adjust the mask based on how you decide to represent each network.

As with all network protocols, all fields must be sent on the wire in network byte order.

In the standard version of RIP, packets are typically sent to routers via UDP. In our version, RIP messages are sent directly inside our (virtual) IP packets, using IP protocol 200.

RIP operation

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. 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.

4.4 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, li` Print information about each interface, one per line.
- `routes, lr` 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)

5 If you are writing in C or C++, consider using flexible array members for allocation of your packet structure.

6 When testing your project, feel free to make these times longer if it assists with using a debugger.
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.

4.5 Traceroute (Capstone Only)

Your driver should include the following command for demonstrating traceroute:

ttraceroute 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.

4.6 Route Aggregation and Longest Prefix Match (Capstone Only)

Every router you create in this project represents a single local ip address. In contrast, real world routers represent networks that are attached directly to them. Therefore in this project you will implement a relaxed version of route aggregation:

• all the networks (a single address is a network with mask 255.255.255.255) learned from a specific neighboring router can be aggregated to the smallest network which contains all the learned networks.

You can safely assume that networks this will be tested on will have ip addresses allocated in a manner that conforms to the topology.

• The distance metric of the aggregated network will be equal to the shortest distance from the aggregated routes.

These relaxations will lead to cases where the routers behaviour will diverge from what is otherwise considered correct behaviour: a router may forward a packet whose destination is a “hole” in the network (a target address that part of the network, but does not exist in any node), and the distance metric may be incorrect for some of the nodes in the network (when aggregating we use the shortest distance, leading to some nodes to appear closer than they are).
Apart from these cases, a driver implementing route aggregation MUST behave correctly, therefore Longest Prefix Matching is necessary.

**Note:** The reference node does *not* implement route aggregation. Do not run the reference node while testing your own implementation for this step, as it will not propagate your messages correctly.

## 5 Getting Started

Once teams are defined, you will be assigned a Github classroom link to create a repository for your team. In the meantime, you can view the template repo here: [https://github.com/brown-csci1680/ip-tcp-starter](https://github.com/brown-csci1680/ip-tcp-starter)

Don’t try to fork this repository yet—we will send you a link to create your repository once teams are defined.

### 5.1 Executables

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

**Note:** If you are using an M1 mac, you should use the versions of these binaries in the `tools/arm64` directory—these binaries are compiled for your CPU architecture.

### 5.2 Sample Networks

These are found in the `nets/` subdirectory.

- **ABC.net** - Simple network with three nodes, connected as follows:

  A ---- B ---- C

  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 5000
  localhost 5001 192.168.0.1 192.168.0.2

  B.lnx:
which you can feed each node as their link information. For instance, `A.lnx` indicates that A listens on localhost port 5000 and has one link to B, which listens on localhost and port 5001. A’s IP on this interface is 192.168.0.1. It is connected to another interface (defined by the reversed tuple) with virtual IP 192.168.0.2 (which is provided by B).

- **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.

### 5.3 Utilities for C

If you are using C, we have provided several utility files 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!

- **Linked list**: `list.h`

- **Hash table**: `htable.c` `htable.h`

- **parseLinks**: `parseLinks.c` `parseLinks.c`, implementation of parsing the lnx file. Feel free to use this directly, or modify it.

In addition, you can find an C code example that outlines a structure for the driver’s command-line parsing using `readline` here: [https://github.com/brown-csci1680/lecture-examples/tree/main/ip-project-driver](https://github.com/brown-csci1680/lecture-examples/tree/main/ip-project-driver)

The `readline` library provides some helpful features for implementing a command line (like pressing up to get your last command).
If you are using a language other than C/C++, feel free to use external packages that provide similar features for logging, command line parsing, etc. If you have questions about whether a library is acceptable, just ask!

6 Getting Help

This project has a large scope, but we don’t intend for it to be painful—we have provided a number of resources to help you, and we are always happy to answer questions. To start, make sure you’ve read this handout and really understand what we mean about using UDP as your link layer. EdStem is always a good place to get help on general topics, and the TAs will, of course, be holding TA hours and scheduling appointments.

Collaboration Make sure that you work together with your group partner to share the workload. One way to do this might be to build your high-level interfaces together (e.g., via pair programming), and then have 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.” We recommend designing the core components of your work as a team, such as the layer interfaces and the forwarding table.

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 are welcome to talk to other groups about concepts, algorithms, etc., but each group’s code must be their own.

Mentor TAs Each group will have a member of the course staff (TA or instructor) assigned as their mentor. 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 have approval, you should stay in contact with your mentor, who will ultimately grade your work. Your mentor also will do their best to help answer questions about, debugging, discussing design, etc., but you should feel free to ask questions of any member of the course staff—just because your mentor is helping you out, however, does not mean that they are at your beck and call. Understand that the we (the course staff) are busy too!

7 Grading

7.1 Milestone - 20%

You will schedule a milestone design meeting with your mentor TA by Monday, March 7 at the latest. An announcement will be made on EdStem after the long weekend with instructions on how to schedule—we will be creating meeting slots in the week leading up to the milestone deadline. At this milestone, you should have a clear design for your program and be ready to ask about
any areas where you have questions. 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?

7.2 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 expected to interoperate with the reference implementation. 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.

7.3 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.

7.4 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 fewer points than if we have to find them ourselves.

In your README, please also note whether you developed your work using the container environment or on the department machines, and any instructions necessary to build your project.

Similar to Snowcast, please include a Makefile such that `make` builds your project and places the `node` binary in the top-level directory of your repository.

8 Handing In and Interactive Grading

Once you have completed the project you should commit and push your work to the main branch of your Git repo to deliver your code.

We will grade your projects interactively—after the deadline, your team will meet with your mentor to demonstrate the functionality of your program and grade the majority of it. Between the time you’ve handed in and the demo meeting, you can continue to make minor tweaks and bug fixes, but you shouldn’t be making any major changes. However, the version you’ve handed in should be nearly complete since it could be referenced for portions of the grading.
If you want to maintain a separate version of your code for grading that diverges from your development on `main`, please create a branch called `ip_submit` for the version of the code you want to be graded. We will checkout this branch if we see it—otherwise, we will use `main`.

9 A Warning

*Do not leave this project until the last minute.* This project can be tricky—the best way to stay on top of it is to start working on your design early so that you have opportunities to ask questions. 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!