CS168 Programming Assignment 2: IP over UDP

Assignment Out:	September 26, 2012
Milestone:	October 4, 2012
Assignment Due:	October 11, 2012, 11:59 pm

1 Introduction

In this assignment you will be constructing the network layer of a Virtual IP Network, using UDP as your link layer. The network will use the IP protocol to communicate and the RIP protocol (a distance vector algorithm) to handle packet routing.

Every node in the network will run the **node** executable - this is the program you will write. When you are done, you will be able to start up an arbitrary number of processes running your program. These processes will comprise a virtual network with the ability to route packets within itself. The topology of the network will be determined by external configuration files, which will be passed to the nodes at runtime (more on this later).

For this project, you will be using UDP as your link layer. The communication between processes will be handled using UDP sockets, and your network layer packets (header and all) will be encapsulated as the payloads of UDP packets. Make sure you understand what we mean when we say "UDP is your link layer".

This is a partner project. You should find a partner to work with right away and email the TAs to inform us of your pairing. If you are having trouble finding a partner, make a post on Piazza or come talk to 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

Before you start coding, make sure you have your head wrapped around what you're supposed to do. It may take a little while for everything to click, but once it does, it will all be very straightforward, we promise.

There are two main parts to this assignment. The first is *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 between nodes to populate the routing tables you need for forwarding.

The third aspect of this project is much smaller, but worth mentioning because both forwarding and routing use it. This is IP in UDP encapsulation, your foundational message-sending implementation.

Files you will need to bootstrap your network are available in `/course/cs168/pub/ip/`. You will write a network topology file (we've supplied two examples) describing the virtual topology of your intended network. After running our script `net2lnx` on the complete topology, you'll have 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. Your `node` executable must accept the name of that node's link file as its first argument on the command line.

3 Implementation

Upon initialization, your nodes will 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 as well as 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. In the next assignment, you'll deliver them to your TCP implementation (the transport layer of your network).

3.1 Forwarding

You will use UDP as your link layer for this project. When a node comes up, it will read in a link (.lnx) file. The .lnx file contains one line for each of the node's neighbors, specifying the parameters of the connection between the node and that neighbor. The node will open one interface (a UDP socket) for each neighbor, and will use these to send and receive messages.

All of the IP packets a node sends should be 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 64k buffer, since that's the largest allowable IP packet. 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 virtual IP addresses/ports associated with it, and pass these to functions which wrap around your socket calls.

The IP packet header is available in <`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. 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 decapsulated from your link layer (the UDP sockets), then decide what to do with the packet: local delivery or forwarding.

You will need to write an interface between your network layer and upper layers for local delivery. In this project, some of your packets need to be handed as RIP; any 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 (for example, the protocol number for TCP is 6). 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*[1] for a given protocol number. We'll leave its specifics up to you.

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 (hardcode the routing tables or do something similar) and make sure that your forwarding works. Send data from one node to another one that requires some amount of forwarding. Integration will go much smoother this way.

3.2 Routing - RIP

The second part of this assignment requires you to implement routing using the RIP protocol described in class.

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. Remember to convert them back to host byte order once they are pulled off the wire.

Once a node comes online, it must send a request for routing information on each of its interfaces (links). Additionally, a node must send an update of its routing information on each interface every 5 seconds. An entry in a node's routing table should expire if it has not been refreshed for 12 seconds[3]. If a link goes down, then the network should be able to recover by finding alternate routes to the ones that use that link.

As part of the routing algorithm, you must implement split horizon with poisoned reverse, as well as triggered updates.

3.3 Driver

Your driver program, `node`, will be used to demonstrate all features of the network. `node` must support the following commands.

interfaces Print information about each interface, one per line.

[1]Suggested C prototypes (where the declaration of interface_t is up to you):
```
typedef void (*handler_t)(interface_t *, struct ip *);
void net_register_handler(uint8_t protocol_num, handler_t handler);
```
[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 mess with these intervals to assist in debugging

routes 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).

You should feel free to add any additional commands to help you debug or demo your system, but these are the commands we will test.

4 Getting Started

We've created a few tools that you can use to help you with your project. They are available in

```
/course/cs168/pub/ip/
```

4.1 Bootstrap Scripts

- `net2lnx` - A tool to convert a .net file into a series of .lnx files that each node can read separately.
- `runNode` - Takes a .lnx file as input and runs that node, ssh-ing to the remote machine it is specified to run on, if necessary.
- `runNodeWin` - runNode, but in a different xterm window.
- `runNetwork` - Given a .net file, starts all nodes that are part of that network. Much more convenient than starting all nodes manually!

4.2 Sample Networks

- `AB.net` - Simple network with two nodes.
- `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 C support code / Utilities

We provide some utilities in the `util` folder and suggest others to assist you with the more mundane and less instructive portions of the project.

- `parselinks.c parselinks.h` - Functions to parse a `.lnx` file into a lnxlink struct. Non-C users can look at this for reference.
- Debugging: `dbg.c dbg.h dbg_modes.h colordef.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, `runNode` enables all possible debugging messages. If you want to enable only, say, net layer and routing messages, then you can run:

  ```
  ./runNode file.lnx net,route
  ```

  See `dbg_modes.h` for a full list of debugging modes - feel free to add your own!
- IP checksum calculation: `ipsum.c ipsum.h`. Use this function to calculate the checksum in the IP header for you.
- Linked list: `list.h`. See `parselinks.c` for examples on how to create a list, add elements, and iterate through it.
- Hash table: We recommend you use the glib hash table when you need a hash table - you can find the docs at `http://developer.gnome.org/glib/unstable/glib-Hash-Tables.html`. Include it from `<glib.h>`, and pass the flags

  ```
  -I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include -lglib-2.0
  ```

  to gcc so it can find and link to the library. Alternatively you may use `uthash.h` which is available at `uthash.sourceforge.net`.

  You are welcome to use a different hash table implementation instead, as long as it is a completely generic hash table.

4.4 Reference Implementation

A reference implementation of the project is available in `/course/cs168/pub/ip/node`. Copy the executable into your project directory, then do:

```
./runNetwork loop.net
```

and watch it run!

5 .lnx Files

The usage of the .lnx files can be a little confusing, largely because they mix real hosts and ports with virtual ip addresses. .lnx files are structured so that each line represents a connection with another node in the network; if a .lnx file has two lines, the node that runs it will have two immediate neighbors in the network.

A line in a .lnx file looks like:

```
<host>:<port> <virtual ip address> <host>:<port> <virtual ip address>
```

where the first `<host>:<port> <virtual ip address>` triple represents the local end of the connection, and the second triple represents the remote end of the connection. A line that reads

```
localhost:17001 10.116.89.157 localhost:17000 10.10.168.73
```

indicates that the local end of the connection will run on localhost on port 17001, with the virtual ip address 10.116.89.157; the remote end of the connection will be run on localhost on port 17000, with the virtual ip address 10.10.168.73. These virtual ip addresses are what you will use for routing packets, and exist to simulate a multi-machine network where every node has a different ip address. The host-port pairs are the info that you will use when actually creating your UDP sockets.

6 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.

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 recommend a revision control system such as Git so that you can update each other periodically (commit often, but only when the build succeeds!). We are going to give out github repos to teams. Register accounts with github and give your account names to the tas. These repos are 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 that will get in touch with you by email within the next couple of days. This means that you'll have one of the three TAs as your group's advisor. 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.

7 Graduate Credit

You must implement IP fragmentation for graduate credit. The MTU for each link should be set as an additional command line interface option as:

mtu *integer*0 *integer*1 Sets the MTU for the link integer0 to integer1 bytes.

8 Milestone

The milestone will consist of two parts. First you will provide a detailed review of your design including a threading rationale, the use of select(), struct specifications, and a description of the algorithm used to populate routing tables. Second, you will demonstrate a working driver (the underlying functionality may be unimplemented) with a fully functional `interfaces` command. This meeting should be scheduled within one day of the milestone date (10/4). The milestone will constitute 20% of your grade for IP.

9 Handing In and Interactive Grading

Once you have completed the project you should run the electronic handin script `/course/cs168/bin/cs168_handin ip` to deliver us a copy of 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.

10 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. Start talking with your partner right away, and get ready to get connected!