IP for Minet

Project Part C

Overview

In this part of the project, your group will build a minimal implementation of IP for the Minet TCP/IP stack. You will be provided with all parts of the stack except for the IP module. You may implement the module in any way that you wish so long as it conforms to its interfaces to other Minet components, and to the reduced IP standard described here. However, Minet provides a considerable amount of code, in the form of C++ classes, that you may use in your implementation. You may also earn extra credit by implementing additional parts of the IP standard.

The Minet TCP/IP Stack

The Minet TCP/IP Stack is documented in a separate handout. After completing project part B, you should be quite familiar with Minet. You will be given the source code to all of the Minet modules, except for ip_module and tcp_module. The ip_module source you will be given is only a framework – it simply connects with the rest of the stack. You are responsible for filling it out. You can either use your tcp_module implementation or copy the /home1/pdinda/netclass-execs binary.

Updating Your Copy of the Stack

Before you begin working on ip_module, do a "cvs update" to bring down the latest versions of all the files. The main new file I want you to get is ip.cc, which implements the IPHeader class. Originally, I was going to have you implement this yourselves, but I've decided to just give you the code. If you don't want to update all the files, simply run "cvs update ip.cc" to fetch only this critical file.

Compilation

To compile, make sure that /usr/local/bin/g++ is first on your path, and then execute "make project_c".

IP Specification

The core specification for IP is RFC 791, which you can and should fetch from www.ietf.org. In general, you will implement IP as defined in that document, except for the parts listed below.

- You do not have to support the Type Of Service field, other than copying it. Your service may be completely FIFO.
- You do not have to implement any IP options.
- You do not have to support IP fragmentation. You should limit yourself to sending 576 byte packets and always set the Don't Fragment flag. This size IP

packet is guaranteed not to be fragmented by the network, and all IP implementations must support at least this size packet.
- You may assume that the MTU is 1500 (ie, that everything is an Ethernet).
- You do not have to send or handle ICMP packets, although you may do so using the ICMPPacket class.
- You must implement routing and support at least two interfaces. You must initialize your routing table from the file "./minet_static_routes". By default, this table should contain:

```
10.10.1.0/24      eth0
129.105.99.0/24   eth0
127.0.0.0/8       lo
default           129.105.99.1    eth0
```

where 10.10.1.0 should be replaced with your network address. The last line indicates that the packet should be sent to the gateway at 129.105.99.1 using eth0. You should be able to support arbitrary minet_static_routes files in this format.

Chapter 4 of your textbook covers IP within the context of routing. In addition, you will find the readings from Stevens, RFC 791, and RFC 1180 to be useful.

Interfaces

You must fully support the following two interfaces.

- eth0 : The first Ethernet card
- lo : The local loopback device

In addition, you must implement support for an arbitrary number of Ethernet devices (eth1, eth2, etc). The local loopback device forwards IP packets right back up the stack. It is used to allow two processes that are sharing the same Minet stack to communicate using IP packets. The IP address of lo is 127.0.0.1. eth0 will be the destination for most packets bound for other machines. You must assign it the IP address in the environment variable MINET_IPADDR. Its physical address is in MINET_ETHERNETADDR.

Routing

You must implement support for routing over all interfaces. This means the following:

- Locally generated packets bound for local addresses (ie, the address of any of your interfaces) should be routed to lo so that they go back up the stack.
- Locally generated packets bound for remote addresses should be routed through the most appropriate interface according to the longest prefix matching rule.
- Remotely generated packets bound for local addresses should be routed up the stack.
- Remotely generated packets bound for remote addresses that are routable with your routing table should be routed out the appropriate interface according to the longest prefix matching rule.

- Remotely generated packets bound for remote addresses that are not routable should be dropped.

IP Module / IP Multiplexor Interface

The IP module talks to the IP multiplexor using Packets that have extracted IP Headers. The Packets may have other headers, such as Ethernet, UDP, TCP, or ICMP headers, however, an IP header must always be present. When a received IP packet is bound for a local address, it should be sent to the IP multiplexor. Similarly, the IP multiplexor will send outgoing IP packets to the IP module.

IP Module / Ethernet Multiplexor Interface

The IP module communicates with the Ethernet multiplexor using RawEthernetPackets. To send a packet, it sends a RawEthernetPacket to the multiplexor. To receive a packet, it receives a RawEthernetPacket from the multiplexor.

IP Module / ARP Module Interface

The interface between these modules is request response. To make an ARP request, the IP module sends an ARPRequestResponse with the flag REQUEST to the ARP module. In response, the ARP module will send back an ARPRequestResponse with the flag RESPONSE_OK (meaning that the IP address was found in the cache and that the mapping in the response is valid), or RESPONSE_UNKNOWN (meaning that no mapping was in the cache and thus the response's mapping is invalid.) Generally, when a RESPONSE_UNKNOWN is returned, the ARP module will have also made a broadcast ARP request. The IP module may retry the request at any time.

Recommended Approach

There are many ways you can approach this project. The only requirements are that you meet the IP specification detailed above, that your IP module interfaces correctly to the rest of the Minet stack, and that your code builds and works on the TLAB machines. We recommend, however, that you use C++ and exploit the various classes and source code available in the Minet TCP/IP stack. Furthermore, we recommend you take the roughly the following approach.

1. Read Chapter four of your textbook, and skim Stevens.
2. Skim RFC 791, and glance at RFC 1180.
3. Reread the "Minet TCP/IP Stack" handout.
4. Build Minet using make project_c. We have provided an `ip_module.cc` that consists of the framework code needed to start up and connect to the appropriate fifos.
5. Extend this IP module so that it simply prints incoming RawEthernetPackets from the Ethernet multiplexor and locally generated Packets from the IP multiplexor. You should be able to run the stack with udp_client or tcp_client as the application and see packets printed when you type and when you send your stack traffic from netcat (nc).
6. Write a simple lo interface and router that forwards IP packets bound for 127.0.0.1 back up the stack. You should be able to test this using udp_client.

7. Write a generic interface class and specialize it for Ethernet. This class will store state about the interface such as: (0) it's name, (1) whether it is up or down, (2) its IP address, (3) its physical layer (Ethernet) address, and (4) how to talk to it.
8. Write a routing table class. A routing table consists of mappings of network addresses (i.e., 10.10.1.0/24) to interfaces. Your routing table should be able to initialize itself with the mappings stored in a file (the minet_static_routes file). Your routing table should support dynamic changes (like those induced by RIP messages).
9. Implement the longest matching prefix rule within your routing table class. When given an IP address, your routing table should be able to respond with an interface.
10. Configure your minet_static_routes file as described above.
11. Add code to construct an IP Packet from each RawEthernetPacket you receive from the Ethernet multiplexor.
12. Add code so that each Packet you construct from an incoming Ethernet packet is routed according to your routing table.
13. Test incoming packets using netcat in UDP mode.
14. Add code so that each Packet you receive from the IP multiplexor is routed using your routing table.
15. Interface with the ARP module so that you can find hardware addresses for these outgoing packets. You'll need to put these destination addresses into an Ethernet header and prepend it to the outgoing packet.
16. Add code to construct a RawEthernetPacket from such an outgoing packet and to send it out to the Ethernet multiplexor.
17. Test outgoing packets using udp_client.
18. Test bi-directional communication using tcp_client.
19. Test your routing table by adding clones of your eth0 interface bound to different addresses.
20. You're done!

Extra Credit: IP Fragmentation Support

For extra credit, you may implement support for IP fragmentation. If you do so, you should implement support both for fragmenting packets and for reassembling them.

Extra Credit: RIP Support

For extra credit, you may implement support for the RIP routing information protocol, so that you can advertise your routes and take advantage of the route advertisements of others. .

Mechanics

- Your code must function as a `ip_module` within the Minet TCP/IP Stack, as described in the Minet Stack handout.
- Your code should be written in C or C++ and must compile and run under Red Hat Linux 6.2 on the machines in the TLAB. In particular, we will compile your code using GCC 2.95.2 and GNU Make 3.78.1, which are installed on the lab machines. You must provide a Makefile. We will expect that running "make" will generate the

executable `ip_module` and that this module will meet the specification described in this document and in the "Minet TCP/IP Stack" handout.

Things That May Help You

- RFC 791 is essential.
- RFC 1180 is a handy tutorial on IP and the lower layers it talks to.
- Chapter 4 of your book. Section 4.4 is a reasonable introduction. Remember that you don't have to implement ICMP, routing protocols, or fragmentation.
- Rick Stevens, "TCP/IP Illustrated, Volume1: The Protocols"
- The handout "Unix Systems Programming in a Nutshell"
- The handout "Make in a Nutshell"
- The handout "The TLAB Cluster"
- The C++ Standard Template Library. Herb Schildt's "STL Programming From the Ground Up" appears to be a good introduction
- GDB, Xemacs, CVS, etc.