15-441 Project 2, Fall 2001
IP Layer and Extensions

Out: Tuesday, Oct 2, 2001

Due: Thursday, Oct 25, 2001, 5:00 pm

1 Introduction

In project 1 we asked you to implement a simple FTP server using the networking functionality provided by the Solaris kernel, so by now you are familiar with the usage of network functions such as `socket()`, `bind()`, `listen()`, `accept()`, etc. In this project, your task is to implement the IP layer of the network stack. You will link your IP layer to our simulator, which implements the basic pieces of an operating system as well as the physical, link, transport, and socket layers of the simulated network.

The resulting system is very realistic, and will provide you with useful experience in network programming. Details of the simulator and how to use it are described in a separate handout.

When implementing the IP layer, you will also have to add support for network address translation (NAT) and for stateless firewalling.

You will also implement the Dynamic Host Configuration Protocol (DHCP), which allows a host to acquire a dynamic IP address.

In project 3, you will add stateful firewalling to the IP layer implemented in this project.

2 Overview

In this section we present a high-level overview of the project. We will go into more detail in later sections. The networking code in the kernel is organized into several layers which you have already seen in class. The organization of the network code in the simulated kernel is shown in Figure 1. Note that the application layer has been broken into two "mini" layers, the first one consisting of the actual user programs and the second one of the socket layer.

- In order to access the kernel's networking functionality, user programs use the socket API, which includes *system calls* such as `Socket()`. In this project, you will implement a DHCP client and server application that use the *socket layer*. DHCP allows a client host that does not have a static IP address assigned to it to acquire a dynamic IP address from the server host.
- The *socket layer* is a protocol-independent interface to the transport protocol. It is already implemented in the simulator. The functionality provided by the *socket layer* is described in the simulator handout.
- The *transport layer* contains the implementation of the transport protocols UDP and TCP. These protocols are already implemented in the simulator.
- The *network layer* contains an implementation of the IP protocol. For this project, you will implement basic IP functionality of a router, that is, input and output processing of datagrams and forwarding of datagrams. In addition, you will also add support for NAT and firewalling.
- The *link layer* and the physical layer are implemented in the simulator. Your *network layer* will interact with the *link layer* to send/receive packets. The functionality provided by the *link layer* is described in the simulator handout.

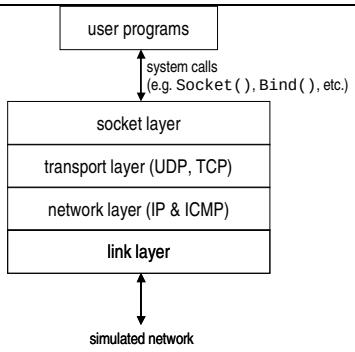

Figure 1: Networking code organization in the simulated kernel

The project directory for this project will be:
`/afs/cs.cmu.edu/academic/class/15441-f01/projects/project2/`

In this handout, we will use `$PDIR` to denote this directory. For your/our convenience, we provide you a template for some of the code that you must write. This includes a Makefile, skeleton definitions of some important structures, skeleton prototypes of some interface functions, etc. In this handout, we will reference the names of some functions/structures defined in the template files. All the template files are in `$PDIR/template`.

3 The Network Layer: IP

The IP layer provides a common interface between different transport-layer protocols and different link-layer devices. The network layer for this project will be modeled after IPv4 [1]. You can also read Chapter 3 of Stevens' *TCP/IP Illustrated, Volume 1* [2] or Chapter 4 of the textbook, since they are probably easier to read than the RFC.

Your implementation *does not* need to handle IP fragmentation, multicast, IP header options, and type-of-service (ToS). Therefore, your fragment offset field should always be zero. The identification field is used to uniquely identify each IP packet sent by a host. It is typically used for fragmentation and reassembly. Although we do not ask you to implement fragmentation, you should set the identification field according to the specification. A simple heuristic such as "incrementing by one each time a packet is sent" is sufficient for our purposes.

You are, however, responsible to correctly set, handle, and verify IP version number, IP header length, IP packet length, time to live (TTL), protocol number, checksum, and source and destination addresses.

Prototypes of the IP functions that you need to implement are provided in `$PDIR/template/ip_io.c`. When a node boots, it calls `ip_init()`. If your implementation of the IP layer requires initialization code, you should place it there.

3.1 Sending Packets

This section describes what happens when a user program sends a packet using the UDP transport protocol. The steps look similar when using the TCP transport protocol. Figure 2 depicts the functions involved. First, the user program calls `Sendto()` to send a packet. The simulated kernel invokes the corresponding socket layer handler `k_sendto()`.

One important task of `k_sendto()` is to convert the data (passed in by the user using the pointer argument) into a `pbuf` data structure (or a chain of `pbuf`'s). This is because, inside your kernel, a packet is passed around using the `pbuf` structure. The `pbuf` structure is very similar to the `mbuf` structure used in real BSD-style networking stacks. It is described in detail in the simulator handout. For now, just remember that, inside the kernel, a packet is represented by a pointer to a `pbuf`.

After verifying arguments and converting the data into `pbuf`'s, `k_sendto()` invokes the UDP-layer interface function `udp_usrreq_send()`, which invokes the UDP function `udp_send()`. `udp_send()` is the function that

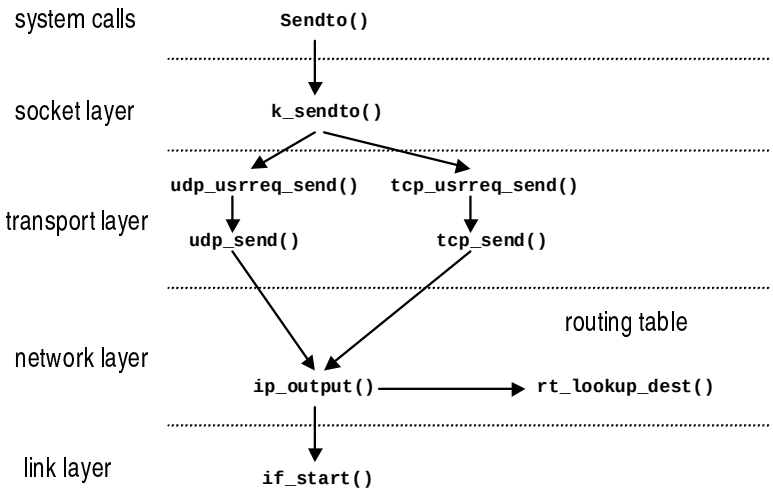

Figure 2: A user program sends a packet by `Sendto()`

actually performs the send operation of the UDP layer, for example, adding the UDP header, doing the checksum, etc. There are corresponding functions for TCP.

After `udp_send()` finishes its work, `ip_output()` is invoked. `ip_output()` will construct the IP header, calculate the checksum, query the routing table (by calling `rt_lookup_dest()`) to find the output interface, and so on. When `ip_output()` finishes constructing the packet, it invokes the link-layer function `if_start()` on the appropriate interface, which then sends out the packet. In this figure, `if_start()` is provided by our simulated kernel, and you will implement the functionality provided by `ip_output()`. Details of how to interact with the link layer are described in the simulator handout.

The output function `ip_output()` is passed a pointer to a `pbuf`, which contains the data packet to be sent. You can assume that space for the IP header has already been allocated in this `pbuf` and that the `p_data` member of the passed `pbuf` structure points to this IP header.

3.2 Receiving Packets

This section describes what happens when a packet is received from the simulated network. Figure 3 depicts the sequence of steps that take place. After a packet is received from the network by the network interface device, and processed by the link layer, the processed packet is delivered to the network layer. More specifically, the link layer will invoke the IP routine `ip_input()` (more details on how to interact with the link layer can be found in the simulator handout).

When `ip_input()` is called, it will check the packet for errors, such as IP version mismatch, IP checksum error, etc. If the packet is error-free, there are two possible scenarios:

1. The packet is destined to *this* host: In this case, `ip_input()` will deliver the packet to UDP, TCP, or ICMP. This is done by invoking `udp_receive()`, `tcp_receive()`, or `icmp_receive()`, depending on the type of the packet. (Note that although `icmp_receive()` is at the transport layer in the figure, ICMP is not really a transport protocol.). If `tcp_receive()` or `udp_receive()` fail because there is no receiver binding to the port the packet was sent to, an ICMP message of type `ICMP_UNREACH` and code `ICMP_UNREACH_PORT` needs to be sent back to the sender by the network layer. This failure condition can be detected by checking the return value of `tcp_receive()` or `udp_receive()`, which is going to be set to `FAILURE_PORT_UNREACH`. To avoid ICMP floods, ICMP messages must not be sent as a reaction to failed `icmp_receive()` calls. Interaction with ICMP is described in the simulator handout.

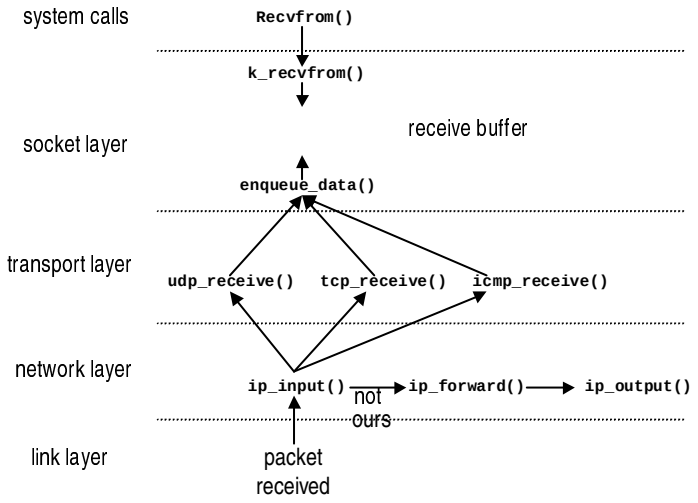

Figure 3: Receiving a packet from the simulated network

2. The packet is destined to some other host: Since the destination is not *us*, IP needs to forward the packet to its destination. Forwarding a packet to its destination is called "routing". Routing is typically done only by routers and not by end hosts. To forward a packet, `ip_forward()` is invoked, which performs some additional work such as decrementing the IP TTL. It then calls `ip_output()` to actually send out the packet (`ip_output()` is described in Section 3.1). If the TTL in an IP packet expires, an ICMP message of type `ICMP_TIMXCEED` and code `ICMP_TIMXCEED_INTRANS` needs to be sent back to the sender.

To complete our description of what happens when a packet is received, let's look at what takes place above the network layer, when the packet is a UDP packet. In this case, `udp_receive()` will process the packet (checking for errors, etc.) and then invoke the socket layer function `enqueue_data()` to insert the packet into the socket receive buffer of the socket this packet is addressed to.

The socket layer function `enqueue_data()` then inserts the packet into the socket receive buffer of the specified `socket` structure. The `enqueue_data()` function must wake up any processes blocked on the `Recvfrom()` call on the socket. `Recvfrom()` grabs a packet from the socket receive buffer and return immediately only if the buffer is not empty. If it is empty, the process that called `Recvfrom()` must be blocked, waiting for the arrival of packets. Therefore, when `enqueue_data()` is called, it must check whether there is any process blocked on `Recvfrom()`, and if there is, it must wake up the blocked process(es). Then `Recvfrom()` can grab the packet and return immediately to the user process.

3.3 Handling Broadcast Packets

Your IP layer should be able to handle broadcast packets (since the DHCP protocol relies on them). More specificially, it needs to provide support for IP packets sent to the *limited broadcast address* 255.255.255.255. Your implementation should follow these rules:

- Datagrams destined to this address must never be forwarded by a router.
- Locally generated datagrams with this destination address must be sent out over all interfaces.
- Received datagrams with this destination address must be forwarded to the transport layer. However, unreachable ports must not result in ICMP messages.

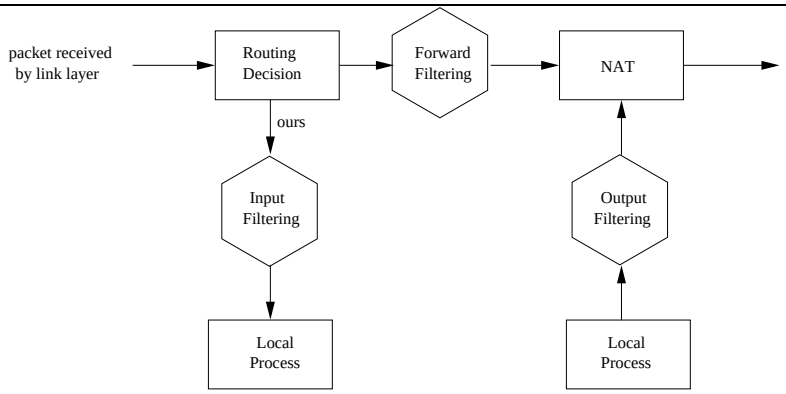

Figure 4: Processing of a packet in the IP layer.

4 Firewalling

For this project, you should implement a simple firewall based on packet filtering. In packet filtering, packets are dropped when they fulfill certain criteria. Figure 4 shows the processing of packets in the IP layer. A packet originates either from the link layer or it is generated by a local process. As can be seen from the figure, filtering can take place at three places: packets destined to the local host go through input filtering, packets received from another host and destined to a remote machine go through forward filtering, and locally generated packets go through output filtering.

4.1 Filter Rules

Your firewall should drop packets based on criteria like the protocol of a packet or its destination address. The default policy is to accept a packet. Explicit rules specify which packets should be dropped.

The syntax of a rule looks as follows:

```
<queue>: <protocol> <src address> <src port> -> <dst address> <dst_port>  \
         (<rule options>)
```

For example, the rule

```
input: tcp 1.2.3.4 any -> 5.6.7.8 80 (flags: SF; ttl: 30)
```

specifies that all TCP packets from any port on 1.2.3.4 to port 80 on 5.6.7.8 should be dropped in input filtering, if they have the SYN and FIN flag set and a TTL value of 30.

The keyword `any` can be given for addresses, ports, and the protocol. For port numbers, a range of port numbers is permitted. For example, `80:100` will match on ports between 80 and 100.

Possible values for `<queue>` are `input`, `output`, or `forward`. The value specifies the queue to which a filter should be applied.

Possible values for `<protocol>` are `tcp`, `udp`, and `icmp`.

Rule options are optional. They match on particular fields in the IP or TCP header. The exact header layout can be retrieved from RFC 791 [1] and RFC 793 [3], respectively. All rule options are separated from each other using the semicolon ";" character. Rule option keywords are separated from their arguments with a colon ":" character.

The following rule options apply to fields in the IP header:

`ttl: <number>;`	Time to Live
`id: <number>;`	Identification
`dsize [>|<] <number>;`	Total Length (The greater/less than signs can be used to indicate ranges and are optional.)

The following rule options apply to fields in the TCP header:

`seq: <number>;`	Sequence Number
`ack: <number>;`	Acknowledgment Number
`flags: <flag values>;`	TCP Flags

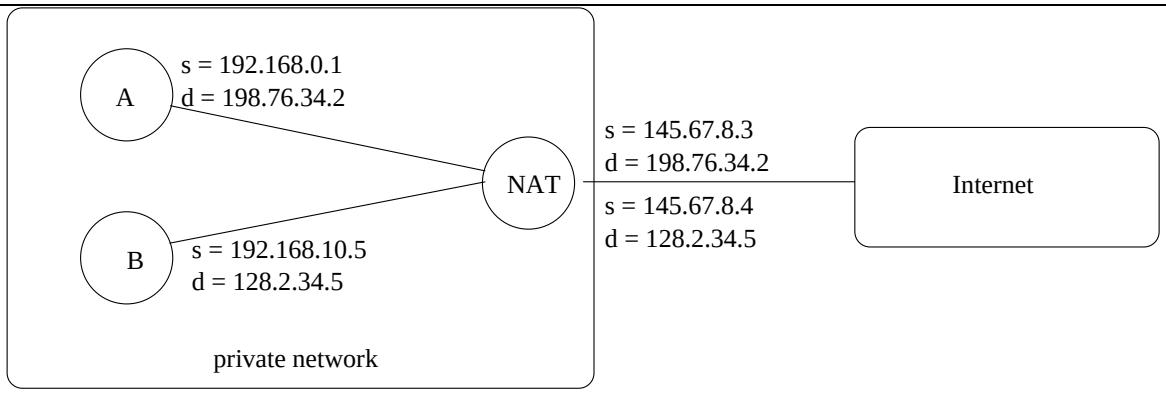

Figure 5: Network Address Translation.

The `flags` rule tests the TCP flags for a match. There are six flags variables: `S` (SYN), `F` (FIN), `R` (RST), `P` (PSH), `A` (ACK), and `U` (URG). There are also logical operators that can be used to specify matching criteria for the indicated flags: `+` (ALL flag, match on all specified flags plus any others), `*` (ANY flag, match on any of the specified flags), and `!` (NOT flag, match if the specified flags are not set in the packet). For example, `flags: SF*` matches on all packets that have at least either the SYN or FIN bit set. There can be at most one logical operator. If no logical operator is given, there has to be an exact match between the flags in the packet and the flags specified in the filter.

4.2 Rules Instantiation

To instantiate the rules discussed in the previous section, you will use an existing mechanism in the simulator for passing information from the user layer to the network layer. This mechanism is based on the function `Setsockopt()`. You should write a user program `setfilter` that parses a file containing filter rules and that configures the firewall through `Setsockopt()` calls. The only argument to the program is the name of a file containing filter rules (e.g., `setfilter -n 1 rules.txt`). To add a filter to the firewall from a user program, you need to call `Setsockopt()` on a routing socket with level `IPPROTO_IP` and option name `IP_FW_SET`. Routing sockets are described in the simulator handout. The fourth argument to `Setsockopt()` is a pointer to an arbitrary data structure, whose length is given in the fifth argument. You should come up with a convenient data structure for passing filter rules and use it as the fourth argument to `Setsockopt()`.

Given the level/option name combination mentioned above, the kernel will call `fw_setsockopt()`, where the actual configuration is done. You need to provide the body of this function, that is, parsing its arguments, whereas one of them is the data structure you defined, and instantiating the filtering rules.

Prototypes of the firewalling functions that you need to implement are provided in `$PDIR/template/firewall.c`. When a node boots, it calls `fw_init()`. If your firewalling implementation requires initialization code, you should place it there.

5 Network Address Translation

With Network Address Translation (NAT), IP addresses are mapped from one realm to another, in an attempt to provide transparent routing to hosts. Traditionally, NAT devices are used to connect an isolated address realm with private unregistered addresses to an external realm with globally unique registered addresses. There are many variations of address translation. For the project, you are going to implement Basic NAT [4].

Figure 5 demonstrates the usage of Basic NAT in an example scenario. Hosts A and B have private IP addresses 191.68.0.1 and 192.168.0.5, respectively. Private IP addresses are unique within the local network, but not globally. All the packets going from the private network to the Internet have to go through a NAT box. The NAT box is assigned two globally unique IP addresses (145.67.8.3 and 145.67.8.4) for address translation. Assume host A wants to communicate with a host in the Internet that has the globally unique address 198.76.34.2. It generates a packet with this destination address. The source address is set to its private address. When the NAT box receives this packet, it

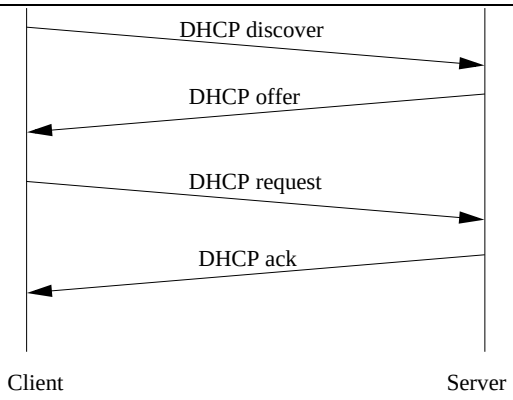

Figure 6: DHCP.

replaces the private source address with the global address 145.67.8.3. Similarly, if host B wants to communicate with a host on the Internet, the NAT box uses 145.67.8.4 as source address for packets coming from B. For IP packets on the return path, a similar address translation is required.

Figure 4 shows where in the IP layer NAT takes place. For this project, we assume that the NAT box has a globally unique IP address for each host in the private network and that the mapping between a private and a global address is static. In your implementation, you need to implement Basic NAT. You do not need to change port numbers in TCP/UDP packets and you do not need to deal with ICMP packets. However, in addition to the source address (or the destination address for the reverse path), it might be required to also modify some other fields in the TCP/UDP/IP header to make end-to-end communication work.

You should write a user program `setnat` that parses a file containing the mappings from private to public IP addresses. An example entry is given below:

```
192.168.0.1 -> 128.2.4.5
```

The only argument to the program is the name of a file containing the mappings (e.g., `setnat -n 1 mapping.txt`). To add a mapping to the NAT box from a user program, you need to call `Setsockopt()` on a routing socket with level `IPPROTO_IP` and option name `IP_NAT_SET`. It is up to you to come up with a convenient data structure for the option value. Given this level/option name combination, the kernel will call `nat_setsockopt()`, where the actual configuration is done. You need to provide the body of this function.

Prototypes of the NAT functions that you need to implement are provided in `$PDIR/template/nat.c`. When a node boots, it calls `nat_init()`. If your firewalling implementation requires initialization code, you should place it there.

6 Dynamic Host Configuration Protocol

The Dynamic Host Configuration Protocol (DHCP) [5] provides configuration parameters to Internet hosts. DHCP consists of two components: a protocol for delivering host-specific configuration parameters from a DHCP server to a host and a mechanism for allocation of network addresses to hosts. DHCP is built on a client-server model, where designated DHCP server hosts allocate network addresses and deliver configuration parameters to dynamically configured hosts. In this project, you should implement a DCHP client and a DHCP server.

Figure 6 gives a brief overview of the DHCP protocol. A host that does not have a static IP address assigned to it invokes the DHCP client. The client broadcasts a DHCP discover msg. A or multiple DHCP servers on the same network receive this message and broadcast a DCHP offer message. The message contains the IP address that is offered to the client by a server and further configuration parameters such as the IP address of a router. The client picks one of the offered IP addresses and broadcasts a DHCP request message to let the DHCP servers know of its choice. Finally, the DHCP server whose IP address got picked by the client confirms this choice with a DHCP ack packet.

Typically, a dynamic IP address is assigned a finite lease, allowing for serial reassignment of network addresses to different clients. If a lease expires, a client should renew its lease or get a new IP address. If a client no longer needs its address, it should issue a message back to the server to release it.

Both the DHCP client and DHCP server are user-level applications. Your DHCP server should be called `dhcp_server` and have the following interface:

```
dhcp_server --topology=FILE --config=FILE --default_lease=LENGTH --max_lease=LENGTH
```

The topology file contains the topology of your network. This file is identical to the one you need for starting up the simulator. You need to pass this file to `read_config()` (defined in `$PDIR/include/rconfig.h`). The configuration file lists each interface of the server that supports DHCP. Also, for each interface, it contains the IP address that should be handed out on that interface. An example configuration file is given below:

```
1 192.168.0.1
2 192.168.0.5
```

The fact that only one address can be handed out on an interface is due to a limitation of the simulator. Typically, a DHCP server connected to an Ethernet can hand out multiple IP addresses to different hosts on the interface to the Ethernet. However, the simulator does not provide Ethernet support. It supports only direct links between hosts. Therefore, there can be only one client host per interface that requests an address from the DHCP server.

The default lease argument gives the default lease time of addresses handed out to clients and the maximum lease argument their maximum length.

Your DHCP client should be called `dhcp_client` and have the following interface:

```
dhcp_client --lease_length=LENGTH --release_time=LENGTH
```

The lease length gives the suggested length of the lease. A server should return a shorter length if the suggested length is longer than the maximum lease length. The release time indicates after how many seconds a client should release its assigned IP address. If the release time is after the expiration time of a lease, the client needs to renew the lease before.

The DHCP protocol is specified in RFC 2131 [5]. DHCP options are specified in RFC 2132 [6]. Your protocol should support at least the DHCPDISCOVER, DHCPOFFER, DHCPREQUEST, DHCPACK, and DHCPRELEASE messages. Some more requirements/suggestions:

- The simulator does not support hardware addresses. Therefore, the `chaddr` field in DHCP messages should always be zero and the DHCPDISCOVER and DHCPREQUEST messages should contain a client identifier option. The client identifier needs to be unique per client. We suggest that you use the node number of a client for this purpose. The function `get_node_number()` (defined in `$PDIR/include/rconfig.h`) returns this number.
- Nodes in the simulator can receive only messages that are sent to their destination address or to the broadcast address 255.255.255.255. Therefore, a client should set the BROADCAST flag in DHCPDISCOVER and DHCPREQUEST messages.
- Neither the server nor the client should probe the network before allocating an address and receiving an address, respectively.
- A client can immediately accept an offered address and does not have to wait for other offers.
- You do not need to handle reboots of clients/servers.
- In DHCPOFFER and DHCPACK messages, the server should return a router option that is set to the IP address of the interface over which the server received the DHCP request. The client should use this address as default gateway for routing. Information about the default gateway of a node is available in the simulator handout.
- The server should be able to handle DHCP requests from multiple interfaces. Note that you are not allowed to create new processes or threads.

- The client and server should be able to deal with lost DHCP messages.

You can set the IP address of an interface using `Setsockopt()`. The socket must be a routing socket, the level is `IPPROTO_IP`, the option name `IP_IF_SET`, and the option value a pointer to `struct if_info` (defined in `$PDIR/include/route.h`).

7 Logistics

7.1 Groups

This project is to be done in groups of two. Send an email to uhengart+441@cs.cmu.edu listing the names of the two group members and their Andrew user IDs as soon as possible. The body of the email should look as follows:

```
member1     <andrew user ID>        <member name>
member2     <andrew user ID>        <member name>
```

For example,

```
member1     uh        Urs Hengartner
member2     rbalan    Rajesh K. Balan
```

You are not permitted to share code of any kind with other groups, but you may help each other debug code. Each member of the group is responsible for both sharing the work equally, and for studying the work of their partner to the point they understand it and can explain it unassisted.

7.2 Equipment and tools

For this project (and all subsequent projects in the course) you will use computers from Sun Microsystems running the Solaris operating system. Such machines are available in several clusters, notably the Wean Hall clusters. You may work from any workstation you have access to, be it a Solaris machine or not, by telnetting to unix*XX*.andrew.cmu.edu. The support code for this project uses the Solaris threads package internally, so *you will not be able to compile, test or run your network stacks except on Solaris machines.*

Since your partner and you will both work on the project, it is essential that you make sure you are not modifying the same file at the same time! Therefore, we highly recommend that you use some form of version control, such as RCS or CVS. You can find a brief tutorial for RCS on the project web page. It also includes pointers to information regarding CVS.

All programs must be written in either C or C++. Note that the TAs do not provide support for problems occuring due to the usage of C++. We strongly recommend you compile your code with `gcc` and debug it with `gdb`. The template `Makefile` is already set up to use `gcc`. Also strongly recommended is the use of the `-Wall` and `-Werror` flags for `gcc`. This will force you to get rid of all possible (well, almost) sources of easily avoidable bugs (so that you can concentrate your debugging efforts on the unavoidable ones). Again, the template `Makefile` is already set up with these flags.

7.3 Provided Code

All of the provided code and infrastructure is available in `$PDIR`, which has the following contents:

- `README`: a file in the project2 directory describing the contents;
- `include/`: the header files for your interface with the simulator — **DO NOT** copy or modify these!!;
- `lib/`: the archive files containing the simulator that you link to;
- `template/`: a template for the code you must write. It also includes a `Makefile`, the `startkernel.pl` script, etc. — you should copy all these files to your own working directory;

- `utils/`: a collection of utilities for testing and configuring your network, explained in the README file in this directory.

It is very important that you *do not* copy the libraries that we provide into your own directory. You should link against the library where it resides in the official project2 directory. This is already taken care of in the template `Makefile` that we provide. We reserve the right to make changes to the library as needed, and you will not receive the updates if you link against your own copy.

The code in the `$PDIR/template/` directory is merely a suggestion: you are free to modify any of the code there or start from scratch. However, it will make it easier for us to help you if you have used the template and retain the function prototypes that we have suggested. Also, the template was designed with a C implementation in mind. It may be more difficult to follow this template if you intend to use C++ to implement your networking layer.

Also note that your project code will use both the Solaris standard header files and the header files specific to this project. The header files such as `<netinet/*.h>` or `<sys/*.h>` are the standard header files on typical UNIX machines. `<project2/include/*.h>` and `"*.h"` are header files specific to this project. When including a header file, you should be conscious whether the header file is a standard header file or a project-specific file.

FYI, you don't need to define data structures for the various headers. They are already defined in several standard Solaris header files. Several examples:

- The UDP header: `struct udphdr` in `<netinet/udp.h>`.
- The ICMP header: `struct icmp` in `<netinet/ip_icmp.h>`.
- The IP header: `struct ip` in `<netinet/ip.h>`.

There are some other header files in the `/usr/include/netinet/` directory that you may find useful. However, you probably don't need all of them. In fact, you don't need to use these structures at all if you prefer defining your own. We just want to let you know that they exist.

7.4 Communication

We reserve the right to change the support code as the project progresses to fix bugs (not that there will be any :-) and to introduce new features that will help you debug your code. You are responsible for reading the bboards and the project home page to stay up-to-date on these changes. We will assume that all students in the class will read and be aware of any information posted to the bboards and the project home page.

If you have any question regarding this project, please post your question to the class bboard. Please make your questions clear and specific to increase the chance that we can solve your problem with one response. As always, the course staff is available for help during office hours.

7.5 A Brief Report

Each group should create a brief report describing their efforts, in one of the following formats: plain text, postscript, or html. Please use the file name `proj2_report.{txt|ps|html}`. Your report should describe the following:

- A breakdown of what each group member did (use a table for this).
- At a high-level, describe your implementation of the firewalling support and the DHCP protocol.
- Describe any interesting testing strategies that you have used.
- Describe what works and what does not, and if not, why (use a table for this as well).
- Your thoughts on the project: was anything too difficult? What would improve the project?

7.6 Submission and Policy

You must use a `Makefile`, and your project should build completely by typing `gmake`. We will post further guidelines for how to submit your project and report.

Note that we do not expect to grant any extensions for this project. We must be notified *immediately* of any extenuating circumstances you might have, including problems with your partner. A disaster that occurs during the last few days before the deadline may **NOT** qualify you for an extension, since you still would have had the majority of the project duration to work. For example, don't come to us complaining that your partner has been bad for two weeks or you have been sick for two weeks and you need an extension. If you need help with this sort of problem, you must come to us before the damage is irreparable.

7.7 Grading

We will post the grading scheme for the project. The grading philosophy for this project is that your grade increases with the number of components that work. *It is better to have a few solidly-working components than many slightly-broken components, and this should influence your implementation plan.*

8 Advice

It is critical to realize that the third project in this course will require you to extend the IP layer implemented in this project to include additional functionality. This makes a clean design and well-written code a must, since you will live with the code you write now for the next 2 months. If you make a mess of this project, you will cause yourself no end of pain and agony when you attempt to complete project 3. Therefore, when you are designing data structures, you should make sure that your code can be easily extended.

While each student has a different set of commitments and will have to manage his or her time accordingly, this project is a significant effort, so you will have to **start early and make steady progress throughout the next three weeks**.

We suggest that you first familiarize yourself with the simulated kernel. Read the simulator handout, compile the provided dummy sources of the IP layer, and start the simulator. Then, implement the IP layer and test it thoroughly for different topologies. Next, add NAT and firewalling to the IP layer. Finally, you implement the DHCP server and client. The NAT/firewalling and DHCP implementation are very independent, making it easy to split the work between group members. *The design of all the parts should be done jointly, that is, we expect each student to be familiar with all the pieces.*

References

[1] J. Postel. Internet Protocol. RFC 791, USC/Information Sciences Institute, September 1981.

[2] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

[3] J. Postel. Transmission Control Protocol. RFC 793, USC/Information Sciences Institute, September 1981.

[4] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022, Intel Corporation, January 2001.

[5] R. Droms. Dynamic Host Configuration Protocol). RFC 2131, Bucknell University, March 1997.

[6] R. Droms. DHCP Options and BOOTP Vendor Extensions). RFC 2132, Silicion Graphics Inc, Bucknell University, March 1997.