15-441 Project 2, Fall 2004
Network Layer Forwarding and Routing

Out: Mon, 27 Sep 2004

Due: Mon, 25 Oct 2004, 11:59pm

Lead TAs: Pratyusa Manadhata, Arvind Kannan

1 Overview

While Project 1 gave you experience with the application layer in networking, this project will give you experience with one of the lower layers of the networking stack. Specifically, you will design and implement forwarding and routing in an IP-capable network layer.

Note that Project 3 depends significantly on Project 2. To successfully complete Project 3, you will need the facilities that you implement in Project 2 (forwarding). It is vital that you keep this in mind when choosing your project partner, and while working on the project.

2 Preliminaries

2.1 Support Code

As you know, a network layer is not useful in isolation. At minimum, we need a link layer below the network layer, and transport layer above the network layer. We also need facilities for configuring the network.

In the "real-world", the lower layers (link and physical) would be provided by hardware. To simplify your task in this project, however, we provide a machine simulator which implements these layers and the transport layer as part of a Unix process. To further simplify your task, we also provide a socket library. The socket library, as you will recall from Project 1, is the interface between network applications and the operating system's networking code.

The relationship between these components is illustrated in Figure 1. The TFTP server and TFTP client are included as examples of network applications. The gray box in the machine simulator is the component which you must implement. The white boxes are components that we provide.

Details about the machine simulator, including information about how to compile your components with the components that we provide, are given in the simulator handout.

The project directory for this project is:

`/afs/cs/academic/class/15441-f04/project2`

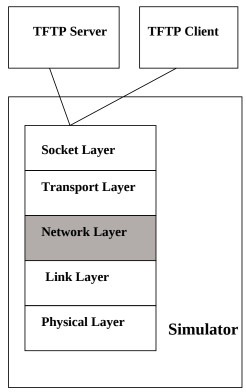

Figure 1: Components of the network stack.

In this handout, we will use `$PDIR` to denote this directory. The simulator libraries and header files are under `$PDIR/lib` and `$PDIR/include` respectively. For your convenience, we provide you a template for the code that you can start with. This includes Makefile, skeleton definitions of some important structures, skeleton prototypes of some interface functions, etc. All the template files are in `$PDIR/template/kernel`. Copy the template directory into your working space and get started with programming. `$PDIR/utils` contains utilities to help you in debugging your code. Read the `README` file under the directory to learn how to use those utilities.

2.2 Project Updates and Announcements

We reserve the right to change the support code as the project progresses to fix any bugs and to introduce new features that will help you debug your code. We will post information about such changes to the course bboard. Please make sure that you are subscribed to the bboard. *We will assume that all students in the class will read and be aware of all information posted to the bboard.*

2.3 Work Directory

So that we can gauge your progress, please use your handin directory as your work directory for this project. If, for some reason, you really cannot work in your handin directory, please copy your files to your handin directory periodically (twice a week).

3 Your Assignment

There are two main parts to Project 2. These are: the network layer and the routing daemon. We explain our requirements for each of these components below.

Destination	Interface
1.1.1.2	1
1.1.2.2	2

Table 1: Forwarding table at Node 1

Destination	Interface
1.1.1.1	1
1.1.2.2	1

Table 2: Forwarding table at Node 2

3.1 Network Layer

In the first part of the project, you will implement basic network layer functionalities.The network layer for this project will be modeled after IPv4 [3]. The network layer provides an addressing mechanism, and the ability to forward packets between nodes. Your network layer must provide both of these facilities. For addressing, we *require* that your network layer use IPv4 addresses.

Section 4.1.4 of the text book describes datagram forwarding in IP. The forwarding table entries consist of (destination IP, interface) pair. Note that this is different from the forwarding table entries described in the text book. The prototype for forwarding table entries is given in `$PDIR/template/kernel/rtable.h`. In the example network given in the simulator handout, the forwarding tables at the nodes 1, 2 and 3 are given in Table 1, Table 2 and Table 3 respectively.

When your forwarding layer receives a packet, it will look for an exact match of destination IP in the forwarding table, and send the packet on the interface obtained from the forwarding table entry.

The network layer for this project will be modeled after IPv4. Your implementation does not need to handle IP fragmentation, multicast, IP header options, and type-of-service (ToS). 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. 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.

Destination	Interface
1.1.2.1	1
1.1.1.2	1

Table 3: Forwarding table at Node 3

Section 2.4.2 of the text book has an example implementation of Internet checksum algorithm, which you may use in this project. Prototypes of the IP functions that you need to implement are provided in `$PDIR/template/ipforward.h`.

The simulator transport layer does not support ICMP. Hence, in case of an error, your forwarding layer will drop the packet and write an error message to `stderr`, instead of sending an ICMP packet.

In the first part of the project, the network layer forwarding table can be filled up statically. We provide user space program (in `$PDIR/utils`) that can relay forwarding information to your kernel code via a "routing socket", as explained in the routing section of the simulator handout. Your code will be responsible for placing this information into your network layer's forwarding tables. You should modify the user level program provided in `$PDIR/utils` to read the forwarding information from a file and relay it to the kernel. In the second part of the project, you will implement the routing daemon and other infrastructure required to compute the forwarding table dynamically.

3.2 Routing Daemon

In the second part of the project, you will implement a simple routing daemon. The routing daemon is intended to be implemented as a user-level application. Thus each node should have one routing daemon running.

The job of the routing daemon is to build a node's forwarding table so that packets can be successfully forwarded to other nodes from that node.

3.2.1 Distance Vector Routing

Your routing daemon will use a standard distance vector routing protocol to update the forwarding tables of the nodes. The distance vector routing protocol is described in your textbook on pp. 274-282 [2].

As described in the textbook, distance vector routing works by having each routing daemon periodically exchange routing updates with its neighbors. Each routing update contains the node's entire routing table. Upon receiving a routing update, a node updates its routing table with the "best" routes to each destination. In addition, each routing daemon must remove entries from its routing table when they have not been updated for a long time. The routing daemon will have a loop that looks similar to the following:

```
while (1) {
  /* each iteration of this loop is a ‘‘cycle’’ */
  message_received = Recvfrom(MSG_NOBLOCK);
  if (message_received) {
    process_incoming_advertisements_from_neighbor();
  }
  currentTime = time();
  if (currentTime >= lastAdvertiseTime + ADVERTISE_PERIOD) {
    advertise_all_routes_to_all_neighbors();
    check_for_down_neighbors();
    set_hop_counts_to_infinity_for_all_routes_through_down_neighbor();
    expire_old_routes();
```

```
    delete_very_old_routes();
    lastAdvertiseTime = currentTime;
  }
  sleep(T)  /* sleep for T seconds*/
}
```

Let's walk through each step. First, your routing daemon checks if any messages have arrived by calling a non-blocking Recvfrom(). If a message was received, then it updates its routing table if any advertised route is new, or better than the current route based on hop counts. In the event of a tie in hop counts for a given destination it always chooses the route where the nodeID of the next hop has the lowest numerical value.

Next, your routing daemon will advertises all of its routes to all of its direct neighbors if enough time has elapsed. If your routing daemon has not received any advertisements from a certain neighbor for a number of advertisement cycles, it is time to consider a neighbor to be *down*. It considers communication with the neighbor to be broken and sets all routes through that neighbor to have a hop count of infinity. The routing daemon propagates this information the next time it advertises its routes.

Next, if a route has not been updated in a long time, then we "expire" it by marking it invalid and setting its distance to infinity. However, we keep advertising it to our neighbors (as infinity). Note that the routing daemon will never expire the routes to the users on its own node.

Finally, if a route has been expired (had infinity distance) for many advertisement cycles already, then you can delete the route permanently from its routing table. The reason we wait before deleting this information is because we want to be able to tell everyone else that the route is down (by advertising routes with infinite distance).

In addition to this basic operation, you should handle the *count-to-infinity* problem by using *Split Horizon with Poisoned Reverse* and *Triggered Updates* as described in the RIP RFC [1] and the class textbook. If you implement this correctly, routing should re-converge if communication with a node in the network is lost (e.g., a node goes down or a link goes down) or if communication with a node is reestablished (e.g., a node comes back up).

3.2.2 Implementation

Your routing daemon must communicate routing information with all directly connected neighbors. One way to do this is to use a single UDP socket that can receive routing information from all neighbors. This means you will be using the `Sendto()` function, specifying the destination address and port, and the `Recvfrom()` function, which will tell you who sent the packet. By default UDP is blocking, meaning it will not return until a packet is received. However, your routing daemon must send out periodic routing information, as well as, listen for incoming routing information all in one thread. To achieve this you will use a *non-blocking* UDP socket, which will return immediately even if no packets have been received. To make the `Recvfrom()` call *non-blocking* use the MSG_NOBLOCK flag in each call. At this point, you should be able to implement a loop that can send periodic messages and receive messages from multiple neighbors.

There is one more important mechanism for the routing daemon you must implement: the UDP socket used to send routing information must only send to directly connected neighbors and must only send routing information to them if the direct link is up. For example, if a link between two nodes, A and B goes down, an alternate route may be chosen by your routing protocol from A to B going through a third node C. In this case, A should not send routing information to B since they are not directly connected anymore, however, using the IP forwarding table will not achieve this behavior since the routing information will end up going from A to B through C. To fix this problem you will have to set the UDP socket to send packets only to direct neighbors. This can be done by setting the SO_DONTROUTE flag using `Setsockopt()` on the UDP socket. Once this is done, every call to `Sendto()` on this UDP socket will force UDP to call `ip_output()` with the IP_NOROUTE flag. As mentioned in the simulator handout, when this flag is set, the `ip_output` routine, which you will implement, should use the kernel's network interface list, which only has directly connected neighbors, to forward the packet *instead* of the forwarding table, which can reach all nodes in the network.

To update the entries in the node's routing table, the routing daemon should use routing sockets, as described in the simulator handout. Note that you only have to implement the routing socket routines to add an entry, delete an entry, and change an entry. No information will be returned via the routing socket.

For this section, when running the routing daemon, the method described in section 3.1 should **not** be used to populate the routing table. Instead, the routing daemon will assume that the routing table is initially empty. It will read the same network configuration file that the kernel uses upon startup to find out its immediate neighbors in the network topology. It will then use the distance vector routing protocol to update the routing table.

4 Grading Criteria

4.1 General Information

If your project generates compiler warnings, you will lose credit; if your project dumps core during our testing, you will lose substantial credit. Handling invalid input reasonably and generating defensible error codes are fundamental parts of writing any program.

Poor design, documentation, or code structure will probably reduce your grade by making it hard for you to produce a working program and hard for the grader to understand it; egregious failures in these areas will cause your grade to be lowered even if your implementation performs adequately. Putting all of your code in one module counts as an egregious design failure.

It is better to have partial functionality working solidly than lots of code that doesn't actually do anything correctly.

We may test your code to see if you don't leak memory or other system resources. We will send corrupt packets to your network stack.

4.2 Rough Grade Breakdown

Your grade on this project will be determined approximately as follows:

(40%) Forwarding layer We will test your forwarding layer by checking that it can, for example, forward between neighboring nodes, and across multiple nodes.

(55%) Routing daemon We will test your routing daemon by checking that it periodically sends advertisements to neighbors only, it correctly implements the distance vector protocol, it converges to a stable routing state in the event of no failures, and it also handles the case of communication failures.

(5%) Style Poor design, documentation, or code structure will affect the result of your next project as well as your grade in this project.

5 Your Project Submission

5.1 Late Submissions and Extensions

Late submissions will be handled according to the policy given in the syllabus. With regard to extensions, we generally do not expect to grant any extensions for this project. However, if an extenuating circumstance does arise, contact us *immediately.* For example, do not wait until the day before the project is due to tell us 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. As a further note, a disaster that occurs during the last few days before the deadline (such as `rm -rf`) will probably *not* warrant an extension, since you still would have had the majority of the project duration to work.

5.2 Code Requirements and Restrictions

We will run your program on x86 computers running Linux. We recommend that you use similar machines for development. Such machines are available in the Wean clusters. Additionally, several Linux servers (unix44.andrew.cmu.edu-unix49.andrew.cmu.edu) are available for remote login. If you have your own Linux system, you are welcome to use it for this project. Note, however, that we will test your code on the Andrew systems. Thus, *you must make sure your code runs correctly on the Andrew Linux machines.*

You must write your server in C or C++. You may only use the following libraries: `libc`[1], `libstdc++`[2]. (You must contact us in advance if you want to use other libraries.) Your code must compile with `gcc` or `g++`. You should also consider using the -Wall flag to generate full warnings and help to debug.

Note that the machine simulator uses threads internally. To avoid any possibility of conflict between the simulator's threading model and your code, we highly recommend that your code not create additional threads. If, after reading the description of the simulator's threading model, you believe that you need to create your own threads, please contact us.

5.3 Project Writeup

Each group should create a brief report describing their efforts, in one of the following formats: plain text, postscript, or pdf. Your report should include the following:

[1] Specifically, the GNU C Library.

[2] The GNU C++ library. This includes the Standard Template Library.

- A description of the design of your forwarding layer and routing daemon. Note that if there are errors in your implementation of some functionality, we may still be able to give you credit for having worked through the design issues for that functionality.
- A description of what works and what does not (use a table for this). For things that do not work, give your thoughts on what the problem might be.
- A breakdown of what each group member did (use a table for this as well).
- A description of the test cases you used, and any interesting testing strategies that you used.
- Your thoughts on the project: was anything too difficult? What would improve the project? Were there parts of the project that worked particularly well, and shouldn't be changed?

5.4 Hand-in Procedure

Your code files and project writeup (Section 5.3) are to be placed in `/afs/cs/academic/class/15-441-f04/usr/{youruserid}/p2` by 25 Oct 2004, 11:59pm. Your project writeup must be named `report.{txt,ps,pdf}`.

These folders will be locked after this time. Note that *submissions by e-mail will not be accepted*. Since you will need these files in order to work on Project 3, we will leave the read permissions set. In order to modify the files, however, you will need to copy them to a new directory.

The code files must include a Makefile, and your source code. Your `Makefile` should be written such that we can build the following binaries from source by simply running `make` (with no arguments) in your submission directory. The *exact* names of the binaries that must be produced are:

- `kernel` - Your kernel
- `srouted` - Your *s*imple *rout*ing *d*aemon

`srouted` should run according to the following semantics:

usage: `./srouted -n` *i* `-node` *i* `-f` *config_file*

`-n` ***integer***
: **Node ID for simulator**. As the simulator handout mentions, this flag is necessary for all user programs and will not be visible as a command line parameter to the routing daemon.

`-node` ***integer***
: **Node ID for routing daemon**. This flag will be the first command line parameter to `srouted`. *It must specify the same node as the* `-n` *flag specifies.*

`-f` ***filename***
: **Configuration file**. This specifies the network configuration file for the current network. *This file should be the same configuration file that the kernel uses upon startup.*

If your code does not build according to this procedure, your submission *will* lose points. The source code must include all the the `*.c`, `*.h`, `*.cc`, etc. files required to build your binary from source. You do not need to include any library files (such as the C library, or the simulator's libraries).

6 Getting Started

We recommend that you implement the forwarding layer first and then the routing daemon. First, you should familiarize yourself with the simulated kernel. Quickly familiarize yourself with the Makefiles and which of the supplied libraries are used by which parts of the code.

From here, you can implement the network layer without routing: test only a two-node network configuration, where forwarding is not necessary. Then, implement and test the forwarding functionality.

After you have implemented and tested your forwarding layer, you should start designing and implementing your routing daemon.

Use the outline suggested in Section 3.2.1 to start on your routing daemon. First implement the simple distance vector protocol. After you test this, then handle the count-to-infinity problem.

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 project*. Below we suggest a timeline, which breaks up the project in three roughly equal parts:

Week 1: Read all handouts in detail. Try building the simulated kernel and observe the interaction between user programs and the kernel. Design your overall architecture, data structures, etc. Be sure to keep notes on your design, as you will need them for your project writeup.

Week 2: Develop and test your forwarding layer.

Week 3-4: Implement the routing daemon. Throughly test your stack and prepare for the final submission.

We will provide a few test programs to aid in your testing. For example, we will provide user lever server and client programs for file transfer. The primary responsibility for developing test cases, however, is yours.

7 Getting Help

- Talk to your classmates. While you need to write your own original program, we expect conversation with other people facing the same challenges to be very useful.
- Post to the main class bulletin board (academic.cs.15-441) if appropriate. This is probably best for detailed questions the answers to which could help many other students. Questions of the form "I do not understand this project." are not likely to get you much help.

- For clarifications on this assignment, post to the main class bulletin board (academic.cs.15-441). But do not expect an immediate reply, especially at odd hours.
- Read the class bulletin boards. This is really very important.
- Come to office hours. This is particularly useful if you have questions about how to structure your code, or questions about other aspects of your design.
- For private questions, send mail to `staff-441@cs.cmu.edu` and not to individual TAs.

8 Suggestions

Start early. Given that this project involves a substantial amount of design, it is unrealistic to expect that you will complete the project in just a few days. Starting early means you can create a cleaner design, which will be easier to implement and test. Starting late means almost invariably results in a complex design, which is difficult to understand, and even more difficult to implement and test.

Starting early is all the more important if you don't know where to begin. If you find yourself in that situation, start by figuring out which parts of the assignment make sense to you, and which parts don't. Is the problem that some parts of the assignment don't make sense, or that you can't see how the pieces fit together? Come to office hours to get help with the parts that don't make sense.

Design before you code. The project is sufficiently large that we do not expect you to design every detail before writing any code. You should, however, work out the high-level design before coding. You may find it helpful to alternate between design and coding phases. For example, after you've worked out the high-level design, start implementing it. As you do, you'll probably encounter many new questions. At this point, work out the design in more detail. Be sure to consider whether the new questions invalidate some aspects of your earlier design.

Plan ahead. Specifically, keep in mind that Project 3 depends on the functionality you implement in this project. 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.

Use version control. 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.

References

[1] G. Malkin, *RFC 2453 - RIP Version 2*, http://www.faqs.org/rfcs/rfc2453.html

[2] L. L. Peterson and B.S. Davie, *Computer Networks: A Systems Approach*, 3rd edition, 2003

[3] J. Postel, *RFC 791 - Internet Protocol*, http://www.faqs.org/rfcs/rfc791.html