CHAPTER 1

Designing and Implementing Network Protocols

1.1. Introduction

The GINI toolkit includes a bare-bones all-software IP router. This chapter discusses projects, where you design and implement protocols on top of the functionality provided by the GINI router. Before attempting these projects, you need to understand the basic structure of an IP router and gain intimate knowledge of certain portions of the GINI router's implementation (see Chapters **??**, **??**).

1.2. Implementing the User Datagram Protocol

1.2.1. Overview

The *user datagram protocol* (UDP) is a simple protocol that allows an application running in one machine to to send a piece of information to another application running in the same or different machine. A UDP implementation provides three functions: (a) encapsulating information in UDP datagrams, (b) computing the checksums, (c) demultiplex receiving datagrams among the sockets that are open for receiving, and (d) handover the datagrams to the IP layer.

1.2.2. Objective

In this project, you are required to implement a standards compliant UDP service in the GINI router. Your implementation of the UDP service should reuse the functions already provided in the GINI router as much as possible. After the UDP service is implemented, the GINI router should be able to exchange UDP packets with a UML machine or another GINI router. The UML machine uses a Linux implementation of the UDP. Your UDP implementation should provide a socket API that meets the minimum specifications given below.

1.2.3. Background Material

Additional information on UDP including packet formats and how it is structured with respect to the IP protocol can be obtained from most computer networking textbooks. However, for this implementation, it is best to read the requirements outlined in the following Internet RFCs.

RFC 768 — User Datagram Protocol

RFC 1122 — Requirements for Internet Hosts - Communication Layers

RFC 1716 — Towards Requirements for IP Routers

1.2.4. Description

Your UDP implementation should provide a simplified socket API consisting of the following functions:

```
int socket(int type);
int bind(int sockid, int port);
int sendto(sockid, int destip, int dport, char *message, int len);
int recvfrom(sockid, int *srcip, int *sport, char **message, int len);
```

Applications running in the GINI router can use the above functions to send or receive data.

The `socket()` function creates a socket. On error, the `socket()` call returns -1. Otherwise, it returns a non-negative integer that points to a *protocol control block*. The `type` parameter is set to 1 for datagram sockets and 2 for stream sockets; UDP uses a datagram socket. The `bind()` function associates the socket with a local port. The `bind()` function returns success (return value 0) for first call on a socket. Subsequent calls on the same socket yields an error (return value -1). All datagrams sent out after the association will have their source port set to the value given in `bind()`. If a datagram is sent out on an unassociated socket a random port value is chosen by the implementation. Before a socket could receive a datagram, it should be bound to a specific port. If a datagram arrives and no socket is bound for that port, a *port unreachable* message would be sent by the UDP implementation to the source. The `sendto()` function takes a message of a given length and sends it to a target application. On error, `sendto()` returns -1. Otherwise, it returns the number of bytes actually sent to the destination. The `recvfrom()` function takes a pointer to a message buffer. The memory for the message buffer should be allocated by the application. The length parameter specifies the maximum size of the message buffer. On error, `recvfrom()` returns -1. Otherwise, it returns the number of bytes actually received from the source. The source IP address and port number are set by the `recvfrom()` function.

A possible architecture for a UDP implementation is shown in Figure 1.1. The `socket()` function allocates a *protocol control block* (PCB) for the socket. If there are too many active sockets, it could fail to allocate a PCB and return -1. The PCB structure has variables such as local port, input buffer, and number of bytes in the input buffer. The `bind()` function modifies the local port value in the PCB structure. The `send()` function processes the given message according to the following pseudo-code and sends it as one or more UDP datagrams.

```
outputProcessing(int sockid, char *message, int len) {
      createUDPDatagram(fragment, destIP, destport, PCB[sockid].localport)
      createPseudoHeader(datagram)
      computeChecksum(datagram, pseudoheader)
      sendToIP(datagram)

      return bytessent
}
```

Figure 1.2 shows the format of the UDP header. The source port number is obtained from the PCB of the socket. The destination port is given by the application as part of the parameter of the `send()` function. The length is the total size of the UDP packet including the UDP header. The

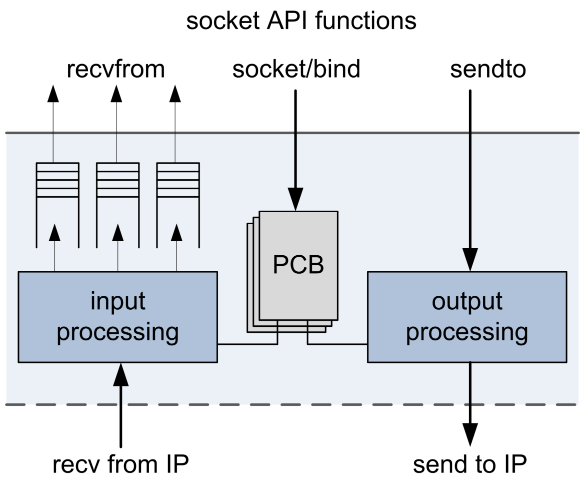

Figure 1.1: Key components of a UDP implementation.

checksum is computed by pre-pending a pseudo header (shown in Figure 1.3) to the UDP packet. The checksum field needs to be initialized to 0 before the checksumming function (can reuse the function provided in gRouter) is applied on the UDP packet. If the checksum field is set to 0 in a UDP datagram, the receiver will skip the checksum validation. This is an easy way of debugging the UDP implementation.

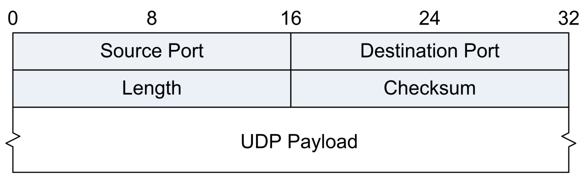

Figure 1.2: The structure of the UDP protocol header.

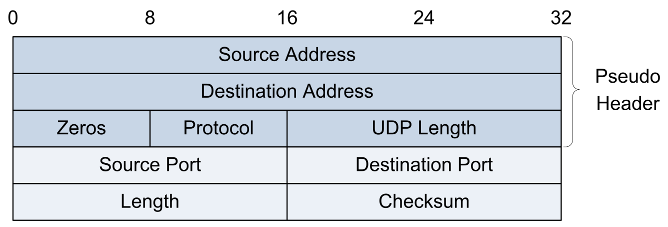

Figure 1.3: Using UDP pseudo header for checksumming.

The IP layer of the GINI router will send an incoming UDP datagram to your UDP implementation by calling a function your provide for processing UDP packets (e.g., `UDPProcess()`). The `UDPProcess()` function processes the incoming packets according to the following pseudo-code.

```
createPseudoHeader(datagram)
computeChecksum(datagram, pseudoheader)
if (checksum not 0)
      reject(datagram)
if (datagram.destport is not open)
```

```
        send ICMP error message
    getInputQueue(datagram.destport)
    enqueue(inputqueue, datagram.data)

    return bytesrecv
```

Figure 1.4 shows an example topology for demonstrating the UDP protocol implementation. Because the virtual machines are user-mode Linuxes, they already have a Internet standards compliant TCP/IP stack that includes the UDP service. The GINI routers, however, do not have the UDP service but provide almost complete IP and ICMP services.

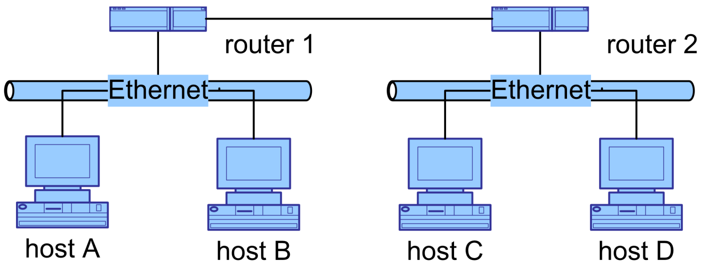

Figure 1.4: Example network topology for UDP experiments.

1.2.5. Useful Tools

You need to implement the UDP service as C functions and integrate it with the GINI router code. The following tools should be helpful in injecting test packets, establishing communication sessions, and visualizing the packets.

- `nc` — create arbitrary UDP connections
- `hping2` — create and inject arbitrary UDP packets
- `tcpdump` — a terminal-based packet visualizer
- `wireshark` — a graphically packet visualizer

1.2.6. Suggested Implementation Plan

1. Read the background material on the UDP protocol.
2. Design and implement the UDP service. The UDP service provides three basic functions: encapsulating, multiplexing, and checksumming.
3. In the UDP service, if a packet is received for a unknown port, it should generate an error according to the Internet standards (check RFCs for detail). This feature is used by some application such as `traceroute` to carry out their function. Similarly, if the local UDP service sends a packet for a port that is not available on the remote machine, an error message will be received. Check whether it is necessary to handle this error message.
4. The UDP service should compute checksums on each outgoing and incoming packet. The UDP checksum includes a pseudo header of the IP packet to bind the payload to the header. Routines provided in the GINI router for IP checksumming could be reused for this process.

5. Provide a testing function that exercises your implementation using the socket API. The application interface shown above is merely a suggestion. The eventual application interface created as part of this project could be similar but it should be complete (handle both client and server needs) and simple.

6. Use the topology shown in Figure 1.4 to test the implementation. The UDP service runs on the routers. Use the packet injection tool `hping2` at `host A` to inject UDP packets and ensure that the UDP service running at `router 1` (`router 2` should work as well if the packet is forwarded) is processing them according to the specifications. Various different types of packets such as bad checksum packets, zero checksums, and undefined ports can be injected by `hping2`. Using `nc` run a UDP server at `host A` at port 9000.

   ```
   nc -l -u -p 9000
   ```

   Connect to the server at `host A` from a client running at `router 1`. This client is a simple C program that is running at the router and using the application interface provided by the UDP service. Once the client connection is working, run a UDP server at `router 1` at port 9050. Again the UDP server is a simple C program that is developed using the application interface provided by the UDP service. Connect to the UDP server running at the router from the machine using the following command.

   ```
   nc -u router_1_IP_addr 9050
   ```

   This should connect to the server and allow simple message communication between `host A` and `router 1`.

7. Run the traceroute from `host A` to `host C` in the UDP mode. The traceroute should provide the expected output.

1.2.7. Expected Deliverables

1. The UDP service built on top of the IP/ICMP services that are already part of the GINI router. The UDP service should interoperate with the one provided by the Linux networking stack.

2. An application programming interface for the UDP service in the form of function calls and their usage.

3. Demonstration of UDP service with applications such as traceroute (using UDP).

4. Simple UDP based telnet-like communication between a router and machine. The client or server at the machine can be implemented using `nc`.