Project 2: IP and UDP header processing

Goal: To build basic IP and UDP header parsing modules.

Task: This task will build on the IP box you built for project 1. Specifically, you will use the **send-config** streams socket, and three UDP sockets, **app, iface1, and iface2**. Similar to project 1, the **send-config** socket will be used to control the test (though this time we won't use the receive-config socket, because that socket is actually redundant). The **IP box** will have to do three things:

1. Receive an upper-layer packet from **app**, encapsulate the packet in UDP and IP, and send the resulting packet over **iface1** (Figure 1);
2. Receive a UDP packet from **iface1**, decapsulate UDP and IP, and send the remainder of the packet to the **app** socket (which will send it to the Test box) (Figure 2).
3. Receive an IP packet from **iface2**, process the IP header (decrement TTL and modify checksum), and send the packet out over **iface1** (Figure 3).

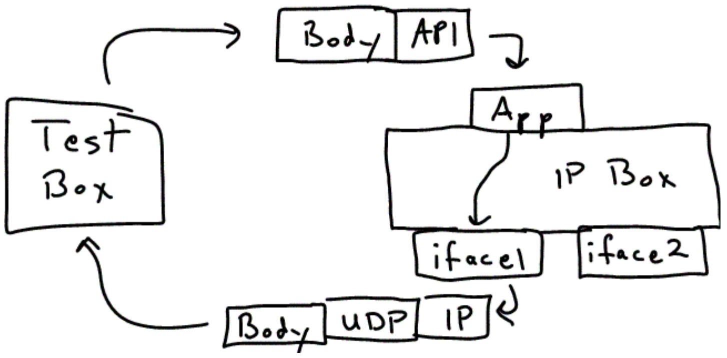


Figure 1

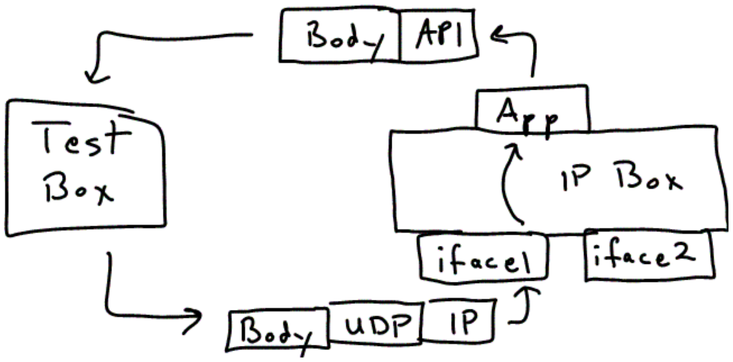


Figure 2

Note that in all of the discussion here, the terms 'IP' and 'UDP' refer to the header in user space that you will generate and parse, not the "real" IP and UDP headers that the kernel is dealing with under sockets. Your code will never see the real IP and UDP headers (though obviously your code deals with the IP addresses and UDP port numbers of the real IP and UDP headers). When I talk of IP addresses and UDP headers, it should be clear from the context which I'm referring to (i.e. the user-space ones or the kernel ones).

In Figures 1 and 2, the UDP and IP headers shown are those that the IP box operates on (not those used by the kernel on the linux box). The "Body" part of the packet is random bits produced by the text box, and are used to generate/check the UDP header checksum, but are not otherwise operated on. The packet header labeled "API" represents the information that would normally be handed to the kernel by the application through the sockets interface (including the sockopts interface). The API header is structured as a number of binary fields as follows:

| src-ip (32) | dst-ip (32) | src-port (16) | dst-port (16) | TTL (8) | ToS (8) | Length (16) |

The length of each field (in bits) is shown in parenthesis. Dst-ip and src-ip are the IP addresses, src-port and dst-port and the UDP port numbers, and TTL and ToS are the values of the respective IP fields. These fields arrive off the "wire" in order of left-to-right (the src-ip field is first) in "network order". The Length field indicates the number of bytes in the Body part, which immediately follows.

When a packet arrives via the **app** socket, the IP box must use these fields to populate the corresponding fields of the IP and UDP headers. When a packet arrives via the **iface1** socket, the IP box must generate the API header from the corresponding fields in the received IP and UDP headers before passing the packet on to the **app** socket. Note that a common source of error is not setting the binary fields to "network order".

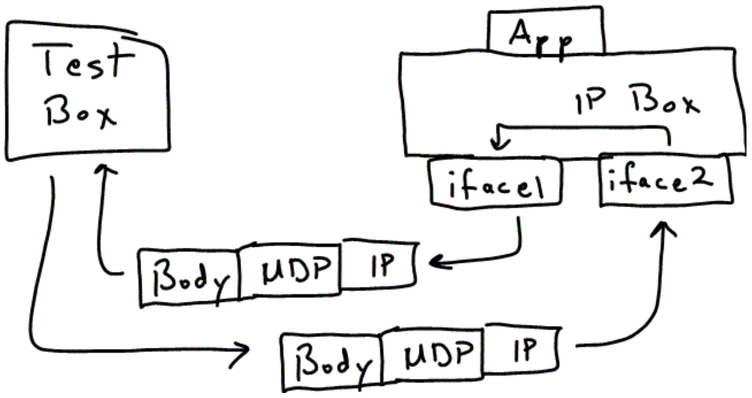
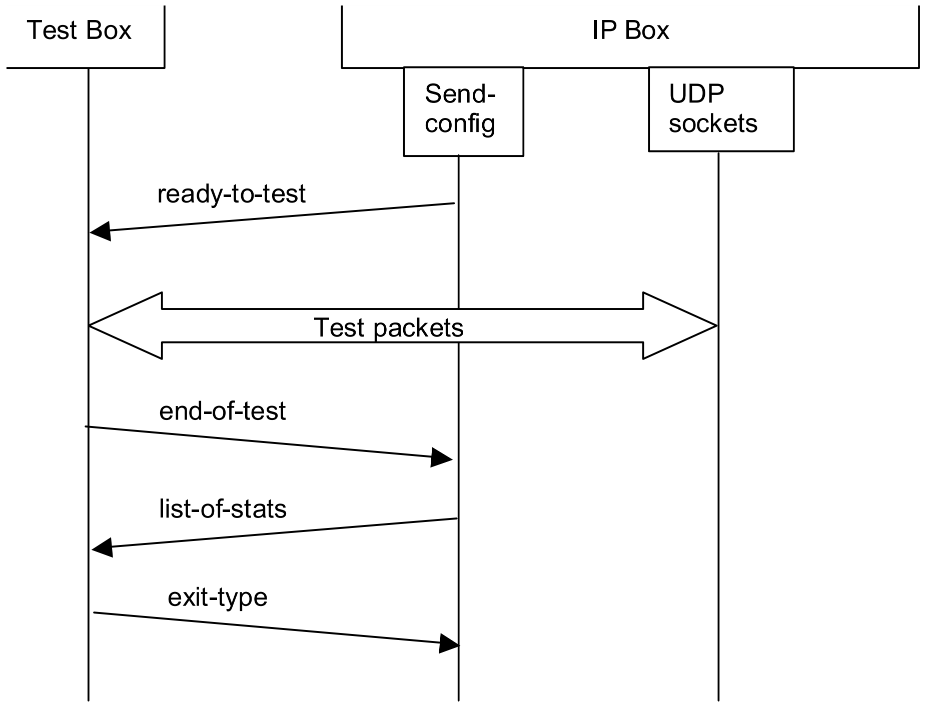


Figure 3

Note that if the IP box receives an IP packet with a malformed IP or UDP checksum, it must drop the packet. Likewise, if the IP box receives an IP packet with a TTL of 0, it must also drop the packet. If the IP box receives a fragmented IP packet, it must reassemble the IP packet before sending it to the **app** socket. If not all fragments that constitute a single packet are received, the IP box must drop all of the received fragments. Note that the fragments do not necessarily arrive in order—the IP box should wait up to two seconds from the time the last fragment was received before determining that fragments are missing. Likewise, if the IP box receives a packet via the **app** socket that is larger than the MTU of the interface, it must fragment the IP packet it transmits. For this purpose, assume that the MTU of the interface is 1500 bytes. Finally, when an IP packet is received over **iface2,** the IP Box must correctly decrement the TTL and modify the IP checksum. You do not have to implement IP options, but other than this, you will be implementing a near-fully functional IP (host and router) and UDP processing machine. You do not have to implement ICMP, however.

The UDP checksum should be calculated as described in RFC 768 (which indicates the four fields from the IP header, the so-called pseudo-header, that must be included in the checksum calculation). IP is described in RFC791. Page 91 of the class textbook gives code for the 1's complement checksum.



The test operates as follows (and is summarized in the figure above):

1. The IP box opens the four sockets described above.

2. The **send-config** socket connects to the **Test box** (on an IP address and port number hard coded into your program) and sends out a "ready-to-test" command.

This command contains the IP addresses and port numbers of the four sockets. The format of this command is as follows:

```
  addr1\nport1\naddr2\nport2\naddr3\nport3\naddr4\nport4\nstring\nnet-id\n
```

where,

addr1	the ip address of the send-config socket in string format
port1	the port of the send-config socket in string format
addr2	the ip address of the iface1 socket in string format
port2	the port of the iface1 socket in string format
addr3	the ip address of the iface2 socket in string format
port3	the port of the iface2 socket in string format
addr4	the ip address of the app socket in string format
port4	the port of the app socket in string format
string	the seed digit of a random number generator when you are still testing out your program, and is the string 'final' when you have completed your testing and want to submit it run the program for grading. If you set the seed digit to 0, then the test box will randomly generate its own seed.
net-id	your net-id (this allows us keep record of your program's performance)

\n is the newline character

Note that this format is essentially the same as project 1, except that of course there is one more UDP socket and one less TCP socket.

The seed digit is required because the test box is going to send you a random series of packets. For your testing purpose, however, you would like the same sequence to be repeated so that you can debug. Therefore, if you set the seed to a given value, the test box will use this to seed its random number generator, and you'll get the same behavior from the test box as a result. If you set the seed digit to 0, then the test box will select its own seed value (random). For your final run, the test box will select its own value for the seed.

3. The **Test box** then sends one or more UDP packets to the **iface1, iface2,** and **app** sockets (in no particular order). These packets must be handled as described above. Following is a list of packets the test box will send (note any given packet may be sent more than once):

	Incoming socket	**Outgoing socket**	**Packet description**
1	app	iface1	Small packet (does not need fragmentation)
2	app	iface1	Large packet (needs fragmentation)
3	iface1	app	Well-formed packet (packets are unfragmented unless otherwise stated)
4	iface1	none	Packet with bad IP checksum
5	iface1	none	Packet with bad UDP checksum
6	iface1	app	Fragmented packet (all fragments received)
7	iface1	none	Fragmented packet (fragment missing)
8	Iface2	iface1	Well-formed packet
9	Iface2	none	Packet with TTL = 1

Note that the last three tests (10 – 12) have been dropped.

4. The Test box then sends an "end-of-test" command via the send-config socket. The format of this command is:
"end-of-test: string\n"

The value of the string will be "ok" if the Test box found no errors. If the test box found errors, the value of string will be some text that describes the nature of the error and which packet the error occurred on (as well as possible). In other words, the string might be "pk3, bad IP checksum; pk7, no packet received", where pk3 refers to the expected response from the third packet sent by the test box, pk7 refers to the expected response from the seventh packet sent by the test box, etc. If the test box finds more than a small number of errors, it may terminate the test prematurely.

5. The **IP-box** will then send a "list-of-stats" command to the send-config socket. The first line of this command will contain:
"list-of-stats\n"

This is followed by a series of lines, each with the format:

"type: value\n"

where the following types and their corresponding values are defined as follows:

type	**value**

app-rcv	The number of packets received over the app socket
app-snd	The number of packets sent over the app socket
iface1-rcv	The number of packets received over the iface1 socket (where each IP fragment is counted as a single packet)
iface1-snd	The number of packets sent over the iface1 socket (where each IP fragment is counted as a single packet)
iface1-drop	The number of packets received over the iface1 socket that were subsequently dropped (where each IP fragment is counted as a single packet)
iface2-rcv	The number of packets received over the iface2 socket (where each IP fragment is counted as a single packet)
iface2-snd	The number of packets sent over the iface2 socket (where each IP fragment is counted as a single packet)
iface2-drop	The number of packets received over the iface1 socket that were subsequently dropped (where each IP fragment is counted as a single packet)

Note that all of the values are represented as text strings. Note that there is a colon character after the type string, and a space between the colon and the value string.

After all of these type:value lines have been sent, the IP box terminates the command with an addition '\n' character. (In other words, the command terminates when two consecutive '\n' characters are received.

6. The **Test-box** then sends an exit message to the **IP-box** at it's send-config socket. The message will have the same basic format as above ("type: value\n"), where type is exactly one of the following:

type	**value**
exit-fail	A text string indicating the nature of the failure. This would only be sent during a "test" run of the test (versus the final run that counts for your grade), and then only if there was an error in the list-of-stats message.
exit-good	The string "ok". This is also only used for a non-final test run.
exit-final	The hashed timestamp. (The hash is sent in string format implying 40 chars for the 160 bit hash) – you must **record** this as proof of your execution!

Test Criteria: The IP box passes if the headers it creates are correctly formed. The IP box passes if it is able to interpret which incoming UDP/IP packets are correctly and incorrectly formed, and forward the correct ones over the corresponding sockets. **Note that the test box will not check to see if the counts are correct.**

For your convenience, we will keep the **Test box** running throughout the assignment period at address **132.236.227.87** (snoopy.csuglab.cornell.edu). The final test itself must be done on port **9994**, and you can do practice runs on port **9992** (which sends fewer packets in a predictable pattern). Note that this port number is different from that of project 1. I will notify the class when the test box is up and running (probably Sunday Feb. 22).