Assignment 2 – Transport Layer Lab

Instructions:

Please complete this assignment by individuals. Both graduate and undergraduate students are expected to complete this assignment.

Getting Started

The objective of this assignment is to give you hands-on experience with the transport layer, including the creation of IPv4, UDP, and TCP headers, transport-layer multiplexing, and the TCP three-way handshake.

Update Cougarnet

Make sure you have the most up-to-date version of Cougarnet installed by running the following in your cougarnet directory:

```
$ git pull
$ python3 setup.py build
$ sudo python3 setup.py install
```

Remember that you can always get the most up-to-date documentation for Cougarnet here.

Resources Provided

The files given to you for this lab are the following:

- **headers.py** - a file containing a stub implementation IPv4 and TCP header classes, as well as a working version of a UDP header class. This is where you will do your work.
- **test_headers.py** - a file containing unit tests, which will be used to test IPv4 and TCP header implementations. You will also do your work here!

- **host.py** - a file containing a basic implementation of a host. Note that this is pared down version of the Host class you will implement in the next assignment in which the send_packet() method simply picks an outgoing interface, creates a frame with the broadcast address as its destination, and sends the frame out the interface.
- **transporthost.py** - a file containing a stub implementation of a host with transport-layer capabilities. It inherits from Host and overrides the handle_udp() and handle_tcp() methods. You will also do your work here!
- **mysocket.py** - a file containing a stub code for both a UDP and a TCP socket. You will also do your work here!
- **echoserver.py** and **nc.py** - files containing classes for applications that use your UDP socket implementation, specifically, an echo server and a very simple Netcat, respectively.
- **scenario1.cfg** and **scenario2.cfg** - network configuration files for testing your implementations. Both contain the same network topology, but they each run different scripts in which different packets are sent.
- **scenario1.py** and **scenario2.py** - scripts that run various tests in conjunction with the network configuration files.

Topology

The files scenario1.cfg and scenario2.cfg describe two different scenarios, but the network topology is the same for both: hosts a, b, and c are connected to each other via a single switch, s1.

```
          +----+
          | a  |
          +----+
            |
            |
            |
+---+     +----+
| b | --- | s1 |
+---+     +----+
            |
            |
            |
          +----+
          | c  |
          +----+
```

s1 has switching functionality already built in, and all hosts have the basic functionality of taking a packet, encapsulating in an Ethernet frame, and sending it to the appropriate host on its LAN/subnet. What you will be adding is the transport-layer functionality--building TCP and UDP headers for application-layer data, adding an IPv4 header, and associating those packets with their appropriate sockets.

Starter Commands

Take a look at the contents of scenario1.cfg. Then run the following to start it up: Make sure you have iptables installed.

```
$ su-
# apt install iptables-persistent
# exit
```

Run the following command:

```
$ cougarnet --disable-ipv6 --display scenario1.cfg
```

If you receive a "permission denied" error, run the following before the starter command:

```
$ chmod 755 ./scenario1.py
```

At this point, the only output will be log messages indicating that messages are being sent from the NetcatUDP instance on hosts a and c to a remote address and port. While eventually (after the first message) there is an instance of EchoServerUDP running on host b, none of the messages will actually be sent, nor will they be received or returned by that EchoServerUDP instance on b. That is because the socket implementations that are responsible for sending and receiving have yet to be fleshed out--by you! In the end, you will log messages indicating that your UDP messages were sent by a and c and received by b, and the responses were seen by a and c.

Part 1 - IPv4, UDP, and TCP Headers

In this part of the lab, you will write the code that will build and parse IPv4, UDP, TCP headers. While you have been able to get by with poking existing headers up to this point, you will want a reliable code base for reading and writing headers because you will be using them a lot more.

Instructions

Complete steps 1 and 2 below to develop a code base that correctly builds and parses IPv4, UDP, and TCP headers. Read both steps before you begin, as it might be easier for you to do them at the same time.

Step 1 - Write the Code for Building and Parsing Headers

In the file headers.py, flesh out the from_bytes() and to_bytes() methods in both the IPv4Header class and the TCPHeader class. This will allow you to create headers for each protocol from raw bytes instances, i.e., as read from the wire. Note that UDPHeader.from_bytes() and UDPHeader.to_bytes() are fleshed out for you, to give you an idea of how this should go.

IPv4 Header

Please note that the diagram describing the IPv4 header is 32 bits (columns) wide.

0	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Version				IHL				Differentiated Services								Total length															
Identification																Flags			Fragment offset												
TTL								Protocol								Header checksum															
Source IP address																															
Destination IP address																															
Options and padding:::																															

(See https://datatracker.ietf.org/doc/html/rfc791#page-11)

- Version - IP version. This will always be 4 for the IPv4 header.

- IHL - Internet header length in 4-byte words. The IPv4 header is 20 bytes long without options, so this field will always be 5.
- Differentiated Services - This will always be 0 for our purposes.
- Total length - This is the length of the entire IP datagram, including IP header and payload.
- Identification - The ID field for reassembling fragmented packets. We will not be handling fragmentation in this, so this field can be 0.
- Flags - Same.
- Fragment offset - Same.
- TTL - Time-to-live value. We will initialize this to 64 for any newly-create IPv4 packets.
- Protocol - The protocol associated with the next header. For example, TCP (IPPROTO_TCP = 6) or UDP (IPPROTO_UDP = 17).
- Header checksum - The checksum of the IPv4 header. For the purposes of this lab, we will not be calculating a checksum, so 0 can be used here.
- Source IP address
- Destination IP address
- Options and padding ::: - Not used.

UDP Header

Please note that the diagram describing the UDP header is 32 bits (columns) wide.

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Source Port																Destination Port															
Length																Checksum															

(See also https://www.ietf.org/rfc/rfc768.txt)

- Source Port
- Destination Port
- Length - This is the length of the entire UDP datagram, including UDP header and payload.
- Checksum - The checksum of a pseudo IPv4 header. For the purposes of this lab, we will not be calculating a UDP checksum, so 0 can be used.

TCP Header

Please note that the diagram describing the TCP header is 32 bits (columns) wide.

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Source Port																Destination Port															
Sequence Number																															
Acknowledgment Number																															
Data Offset				reserved			ECN			Control Bits						Window															
Checksum																Urgent Pointer															
Options and padding :::																															

(See also https://datatracker.ietf.org/doc/html/rfc9293#name-header-format)

- Source Port
- Destination Port
- Sequence Number
- Acknowledgment Number
- Data Offset - The length of the TCP header in 4-byte words. The TCP header is 20 bytes long without options, so this field will always be 5.
- reserved - always 0
- ECN - Explicit Congestion Notification. This will not be used in this lab, so this field can always be 0.
- Control Bits (flags) - Each flag is listed below from left to right (most significant to least significant). However, note that only the `SYN` and `ACK` flags are likely to be used for this lab.
    - `URG`
    - `ACK`
    - `PSH`
    - `RST`
    - `SYN`
    - `FIN`
- Window - the receive window advertised by the sending host. We will not be using this field in the lab, but 64 is a reasonable value for this field, nonetheless.
- Checksum - The checksum of a pseudo IPv4 header. For the purposes of this lab, we will not be calculating a TCP checksum, so 0 can be used here.
- Urgent Pointer - Not used for this lab, so this field can always be 0.

Step 2 - Complete and Tests Code against Unit Tests

The file `test_headers.py` contains a suite of tests that use python's unittest module. Like the doctest module, the `unittest` module provides a way for automated testing of different aspects of your code. For each header class, there is a `bytes` instance containing the raw bytes of the header, to be converted to an object, as well as an instantiated header object, to be converted to bytes.

Run the following to test your header-handling code:

```
$ python3 test_headers.py
```

or, alternatively:

```
$ python3 -m unittest test_headers.py
```

If you have made any changes to the code in headers.py, the tests will most likely fail. That is because test_headers.py must be populated with the correct expected values. Each place in which correct_value is assigned a placeholder value, that placeholder must be replaced with the correct value. Note that the correct values of correct_value for testing the UDPHeader class have already been included, to give you an idea of how this should go.

Part 2 – UDP Sockets

The mysocket.py file contains a stub implementation of a UDP socket in the UDPSocket class. Instantiating the class involves passing local address (local_addr) and local port (local_port), which are the equivalent of creating a socket with (socket()) and calling bind() on it. Thus, an instantiated UDPSocket object knows the destination address and port associated with all UDP/IP packets arriving to it, and the source address and port associated with all UDP/IP packets that are sent from it.

Note that, in addition to the local address and port, two functions are passed as arguments to UDPSocket.__init__() and assigned as callable members variables. That is, they can be used just like methods. UDPSocket._send_ip_packet() is what is called to send an IP datagram that is ready to be encapsulated in an Ethernet frame and sent out of one of the host's interface. In fact, this function is just a pointer to the bound Host.send_packet() method associated with the host for which the UDPSocket was created. UDPSocket._notify_on_data() is simply a function called by the socket instance to let the application know there is data.

The two methods UDPSocket.sendto() and UDPSocket.recvfrom() represent the functions of the same names in the socket API, for sending and receiving data respectively. The idea is that UDPSocket.sendto() and UDPSocket.recvfrom() are called by the applications, as a socket-like API, just as would be done with real sockets. These methods, in turn, call other methods, which you will flesh out as part of your UDP socket implementation, such as UDPSocket.handle_packet() and UDPSocket.send_packet(). For example:

```
sock.sendto(b'abc123', '192.0.2.1', 1234)
```

and:

```
data, remote_addr, remote_port = sock.recvfrom()
```

Packets Issued

With scenario1.cfg, the following packets are sent at the times noted: Each sub-bullet describes the purpose of the primary bullet under which it is listed.

- 5 seconds: Host a instantiates a UDPSocket, s, as part of a NetcatUDP application and calls: s.sendto(b'...', '10.0.0.2', 1234)
    - There is no service listening on 10.0.0.2 port 1234

- 6 seconds: Host b starts an instance of EchoServerUDP on port 1234 ○ There is now a service listening on 10.0.0.2 port 1234

- 7 seconds: Host a instantiates a UDPSocket, s, as part of a NetcatUDP application and calls: s.sendto(b'...', '10.0.0.2', 1234)
  It waits for a message in return and calls s.recvfrom().
    - There is now a service listening on 10.0.0.2 port 1234 to respond to the UDP msg

- 8 seconds: Host c instantiates a UDPSocket, s, as part of a NetcatUDP application and calls: s.sendto(b'...', '10.0.0.2', 1234)

It waits for a message in return and calls s.recvfrom(). ○ There is a service listening on 10.0.0.2 port 1234 to respond to the UDP msg

Instructions

In the file mysocket.py, flesh out following the skeleton methods:

- create_packet(). This method is used create a packet to be sent to a remote host. Use the src, sport, dst, dport, and data arguments to create a UDP datagram with IP header, UDP header, and UDP payload (i.e., data). The method should return a byte instance that is ready to be sent on the wire--after an Ethernet frame header is added.

- send_packet(). This method is used to create and send a UDP datagram. It is what is called by UDP.sendto(). Use the remote_addr, remote_port, and data arguments to create and send an UDP datagram with IP header, UDP header, and UDP payload (i.e., data). Note that this function should largely be implemented by calling UDPSocket.create_packet() and UDPSocket._send_ip_packet().

- handle_packet(). This method takes the following as an argument:

    ○ pkt: an IP packet, complete with IP header. Generally, this could be either an IPv4 or an IPv6 packet, but for the purposes of this lab, it will just be IPv4.

  The method should simply parse the packet, append the data to the socket's buffer, along with the remote IP address and port from which it originated, and call self._notify_on_data() to let the application know that there is data to be read. When the application calls UDP.recvfrom() it will return the contents of the earliest received UDP datagram that has not been read.

At this point, you should be able to run the following command to run scenario 1:

```
$ cougarnet --disable-ipv6 --display --wireshark=a-s1 scenario1.cfg
```

After a few seconds, you should see a log message indicating that host a is sending the message, you should see that packet in your Wireshark output, and you should see a log message indicating that it was received by host b.

Note: If packets are being sent before Wireshark is ready to capture them, you can add --start=10 to add a 10-second delay to the scenario, so Wireshark can catch up.

You should ***not*** expect to see a log message that the message has been received by the application running on b (i.e., the UDP echo server). Nor should you see a return packet in the Wireshark output. That is because you are missing the transport-layer multiplexing--the glue that passes the incoming packets to their correct socket.

In the file transporthost.py, flesh out following method:

- handle_udp(). This method takes the following as an argument:
    - pkt: an IP packet, complete with IP header. Generally, this could be either an IPv4 or an IPv6 packet, but for the purposes of this lab, it will just be IPv4.

  It is called by handle_ip() when the packet is determined to be a UDP packet. handle_udp() should look for a open UDP socket corresponding to the destination address and destination port of the incoming packet. If such a mapping is found, then the handle_packet() method is called on that socket. This is the one you fleshed out earlier! See several examples of where it is called for both client and server in scenario1.py.

  If there is no mapping found, then call the no_socket_udp() method. At this point, you do not need to flesh out the no_socket_udp() method; it is simply a placeholder.

Testing

Test your implementation against scenario 1. After the service has been started, the output should show that a sent the UDP datagram and that b received the UDP datagram--both at the host level and the application levels (it will look a little redundant, but the logging happens at both layers). The output should also show that a received the echoed message, both at the host level and the application level. After the exchange between hosts a and b, you should see output for a similar exchange between hosts c and b.

When it is working properly, test also with the --terminal=none option:

```
$ cougarnet --disable-ipv6 --terminal=none scenario1.cfg
```

Part 3 – TCP Sockets and Three-Way Handshake

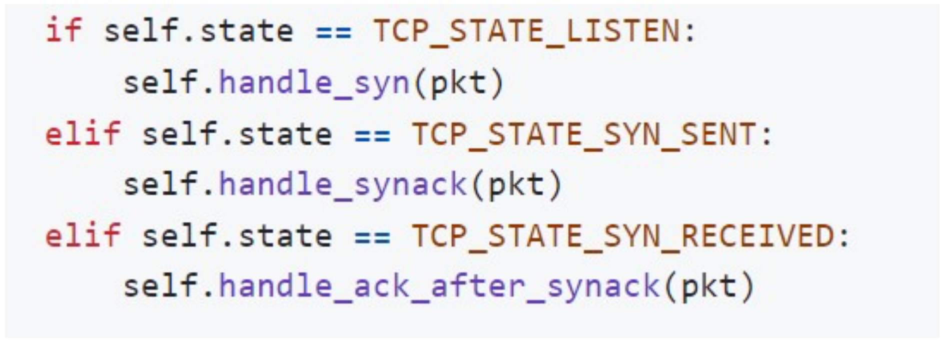
The mysocket.py file also contains a stub implementation of a TCP socket in the TCPSocket class. Typically the TCPSocket class is instantiated in one of two ways:

- With call to the class method TCPSocket.connect(), in which:
  - a new TCPSocket instance is created with a given local address, local port, remote address, and remote port, and an initial state of CLOSED; and
  - the initiate_connection() method is called on the newly created socket, starting the TCP three-way handshake. Note that this method will change the state to SYN_SENT. Additionally, the 4-tuple of local address, local port, remote address, and remote port is mapped to the newly created TCPSocket instance on the host, but that is not done within the method.
- When an instance of the TCPListenerSocket class calls its own handle_packet() method. Because a new socket is required for every TCP connection, the TCPListenerSocket class exists to simply handle packets that are for new TCP connection requests, instantiating a new TCPSocket with each request, so each 4-tuple (remote address, remote port, local address, local port) uniquely maps to its own TCPSocket instance. In the handle_packet() method:
  - a new TCPSocket instance is created with a given local address, local port, remote address, and remote port, and an initial state of LISTEN;
  - the 4-tuple of local address, local port, remote address, and remote port is mapped to the newly created TCPSocket instance on the host; and ○ the handle_packet() method is called on the newly created socket,

handling the first packet received on the server side of the TCP three-way handshake. Note that this method will change the state to SYN_RECEIVED. From this point on, when a packet, pkt, is received by a TCPSocket instance, sock, the following is called:

```
sock.handle_packet(pkt)
```

The handle_packet() method first checks the state of the connection. If the connection is not ESTABLISHED, then it calls the continue_connection() method. The continue_connection() method looks like this:



```python
if self.state == TCP_STATE_LISTEN:
    self.handle_syn(pkt)
elif self.state == TCP_STATE_SYN_SENT:
    self.handle_synack(pkt)
elif self.state == TCP_STATE_SYN_RECEIVED:
    self.handle_ack_after_synack(pkt)
```

The implementation of the three helper functions (handle_syn(), handle_synack(), and handle_ack_after_synack()) is your task. Once you flesh them out, the TCP three-way handshake will be complete, and the connection established!

The whole process should look something like this, which is an example taken directly from RFC 793:

```
    TCP A                                                TCP B

1.  CLOSED                                               LISTEN

2.  SYN-SENT    --> <SEQ=100><CTL=SYN>               --> SYN-RECEIVED

3.  ESTABLISHED <-- <SEQ=300><ACK=101><CTL=SYN,ACK>  <-- SYN-RECEIVED

4.  ESTABLISHED --> <SEQ=101><ACK=301><CTL=ACK>       --> ESTABLISHED

5.  ESTABLISHED --> <SEQ=101><ACK=301><CTL=ACK><DATA> --> ESTABLISHED

(https://tools.ietf.org/html/rfc793#section-3.4)
```

Packets Issued

With scenario2.cfg, the following packets are sent at the times noted: Each sub-bullet describes the purpose of the primary bullet under which it is listed.

- 5 seconds: Host a calls:
  TCPSocket.connect('10.0.0.1', <randomport>, '10.0.0.2', 1234, ...))
  which both creates a TCPSocket instance and initiates the TCP three-way handshake.
  - There is no service listening on 10.0.0.2 port 1234
- 6 seconds: Host b creates a TCPListenerSocket:
  TCPListenerSocket('10.0.0.2', 1234, ...)
  - There is now a service listening on 10.0.0.2 port 1234
- 7 seconds: Host a calls:
  TCPSocket.connect('10.0.0.1', <randomport>, '10.0.0.2', 1234, ...)) which both creates a TCPSocket instance and initiates the TCP three-way handshake.

    - There is now a service listening on 10.0.0.2 port 1234
- 8 seconds: Host a sends a TCP packet from a TCPSocket instance, with a new randomly-selected source port, pretending that it already has an established connection (i.e., client-side socket has state ESTABLISHED and packet does not have SYN flag set). ○ There is no TCP socket associated with the 4-tuple of the incoming TCP packet.
- 9 seconds: Host c calls:

  TCPSocket.connect('10.0.0.3', <randomport>, '10.0.0.2', 1234, ...)) which

  both creates a TCPSocket instance and initiates the TCP three-way handshake.

    - There is a service listening on 10.0.0.2 port 1234

Instructions

In the file mysocket.py, flesh out following the skeleton methods:

- create_packet(). This method is used create a packet to be sent to a remote host. Use the src, sport, dst, dport, and data arguments to create a TCP packet with IP header, TCP header, and optional TCP segment (i.e., data). The method should return bytes instance that is ready to be sent on the wire--after an Ethernet frame header is added.

- send_packet(). This method is used to create and send a TCP packet. It will be used both for control packets (e.g., those associated with the three-way handshake) as well as any packets with segment data. Note that this method should largely be implemented by calling TCPHeader.create_packet() and TCPHeader._send_ip_packet().

- initiate_connection(). This method initiates the TCP handshake. The method should send a TCP packet with the SYN flag set and transition to state SYN_SENT.

- handle_syn(). This method takes the following as an argument:

    - pkt: an IP packet, complete with IP header. Generally, this could be either an IPv4 or an IPv6 packet, but for the purposes of this lab, it will just be IPv4.

Handle an incoming TCP SYN packet. Ignore the packet if the SYN flag is not sent. Save the sequence in the packet as the base sequence of the remote side of the connection. Send a corresponding SYNACK packet, which includes both our own base sequence number and an acknowledgement of the remote side's sequence number (base + 1). Transition to state SYN_RECEIVED.

- handle_synack(). This method takes the following as an argument:

    - pkt: an IP packet, complete with IP header. Generally, this could be either an IPv4 or an IPv6 packet, but for the purposes of this lab, it will just be IPv4.

  Handle an incoming TCP SYNACK packet. Ignore the packet if the SYN flag is not yet, the ACK flag is not set, or if the ack field does not represent our current sequence (base + 1). Save the sequence in the packet as the base sequence of the remote side of the connection. Send a corresponding ACK packet, which includes both our current sequence number (base + 1) and an acknowledgement of the remote side's sequence number (base + 1). Transition to state ESTABLISHED.

- handle_ack_after_synack(). This method takes the following as an argument:

    - pkt: an IP packet, complete with IP header. Generally, this could be either an IPv4 or an IPv6 packet, but for the purposes of this lab, it will just be IPv4.

  Handle an incoming TCP ACK packet. Ignore the packet if the SYN flag is set, the ACK flags is not set, or the ack field does not represent our current sequence (base + 1). Transition to state ESTABLISHED.

At this point, you should be able to run the following command to run scenario 2:

```
$ cougarnet --disable-ipv6 --display --wireshark=a-s1 scenario2.cfg
```

After a few seconds, you should see log messages for packets the TCP SYN packets sent in the scenario, and you should also see those packets in the Wireshark output.

Note: If packets are being sent before Wireshark is ready to capture them, you can add --start=10 to add a 10-second delay to the scenario, so Wireshark can catch up.

You should not see any return packets (e.g., the SYNACK). That is because you are missing the transport-layer multiplexing-the glue that passes the incoming packets to their correct socket.

In the file transporthost.py, flesh out following method:

- handle_tcp(). This method takes the following as an argument:

    - pkt: an IP packet, complete with IP header. Generally, this could be either an IPv4 or an IPv6 packet, but for the purposes of this lab, it will just be IPv4.

  It is called by handle_ip() when the packet is determined to be a TCP packet. handle_tcp() should look for a open TCP socket corresponding to the 4-tuple of the incoming packet. If such a mapping is found, then the handle_packet() method is called on that socket. If no mapping of the 4-tuple is found, then a mapping is looked for with only the local address and local port--i.e., a TCPListenerSocket instance, and the handle_packet() method is called on the corresponding socket.

  See several examples of where install_socket_tcp() and install_listener_tcp() are called for client and server, respectively, in scenario2.py.

  If there is no mapping of either type found, then call the no_socket_tcp() method. At this point, you do not need to flesh out the no_socket_tcp() method; it is simply a placeholder.

Testing

Test your implementation against scenario 2. Before the service has been started, the output should reflect that there is no active socket on the given IP address and port. However, after the service has been started, the output should reflect the TCP three-way handshake, but only when an initial packet has the SYN flag set. After the exchange between hosts a and b, you should see output for a similar exchange between hosts c and b.

When it is working properly, test also with the --terminal=none option, and make sure it works for both scenarios:

```
$ cougarnet --disable-ipv6 --terminal=none scenario1.cfg $
cougarnet --disable-ipv6 --terminal=none scenario2.cfg
```

Part 4 - ICMP Port Unreachable and TCP Reset (Extra Credit)

In Parts 2 and 3 you implemented transport-layer multiplexing, which guided an incoming packet to the appropriate socket on a host. However, when a packet did not match any open sockets on the system, it was simply ignored. This is considered a "stealthy" configuration. An alternative is to let the remote host know that there is no open socket by either sending an ICMP Port Unreachable message or a TCP Reset.

Instructions

Write the code for building and parsing an ICMP header. Then send ICMP error messages and TCP Reset messages at the appropriate times and circumstances.

ICMP Message Header

In the file headers.py, create a class ICMPHeader with both from_bytes() and to_bytes() methods, similar to those you created for IPv4 and TCP in Part 1. A diagram of the ICMP header used for the Destination Unreachable type is below. Please note that the diagram describing the IPv4 header is 32 bits (columns) wide.

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Type								Code								ICMP header checksum															
Unused																															

In the file test_headers.py create a method test_icmp_header(), which tests both the ICMPHeader.to_bytes() and ICMPHeader.from_bytes() methods. Use the following bytes string as input to from_bytes() for testing:

b'\x15\x02\x00\x00\x00\x00\x00\x00'

And use the following as arguments for instantiating an ICMPHeader instance for testing the ICMPHeader.to_bytes() method:

- type: 5
- code: 12
- checksum: 0

Run the following to test your header-handling code:

```
$ python3 test_headers.py
```

or, alternatively:

```
$ python3 -m unittest test_headers.py
```

ICMP Port Unreachable

Whenever a UDP packet arrives for which there is no matching socket, return an ICMP message to the sender with the following characteristics:

- type: 3 (Destination unreachable)
- code: 3 (Destination port unreachable)
- checksum: 0
- unused: 0
- payload: the UDP datagram that was received by the host, complete with its original IP and UDP headers.

TCP Reset

When a TCP packet arrives for which there is no matching socket or when a TCP packet reaches a socket in the LISTEN state, but the packet does not have the SYN flag set, return a TCP packet with only the RST flag set.

Testing

Test your implementation against scenarios 1 and 2. Where you previously received no response from the server after sending a packet that did not match any socket, you should now see responses from the server.

```
$ cougarnet --disable-ipv6 --terminal=none scenario1.cfg $ cougarnet --disable-ipv6 --terminal=none scenario2.cfg
```

General Helps

- You can modify scenario1.py, scenario2.py, and the corresponding configuration files all you want for testing and for experimentation. If this helps you, please do it! Just note that your submission will be graded using only your headers.py, test_headers.py, mysocket.py, and transporthost.py files. The other files used will be the stock files you were provided.
- Save your work often, especially after you move from part to part. You are welcome (and encouraged) to use a version control repository, such as GitHub. However, please ensure that it is a private repository!

What to turn in:

Please note that you should submit your assignment on Gradescope. (We will not accept submissions on Canvas.)

Use the following commands to create a directory, place your working files in it, and tar it up:

```
$ mkdir transport-lab $ cp headers.py test_headers.py transporthost.py mysocket.py transport-lab $ tar -zcvf transport-lab.tar.gz
transport-lab
```