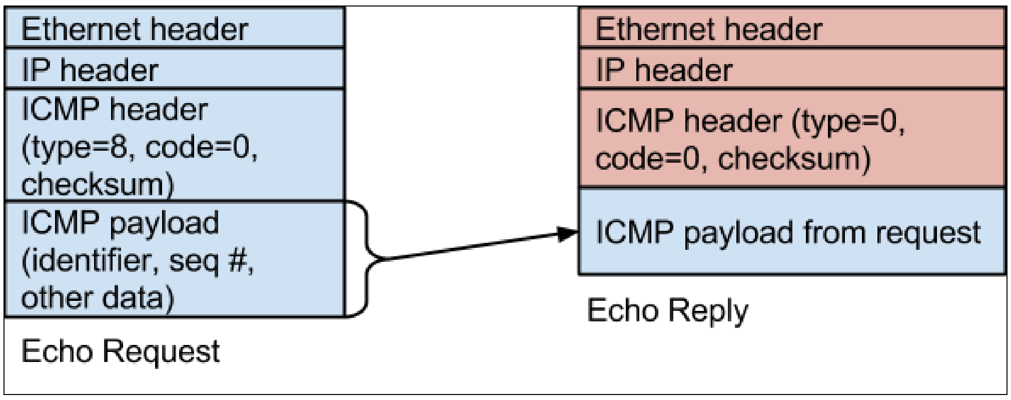
Simple Router

Overview

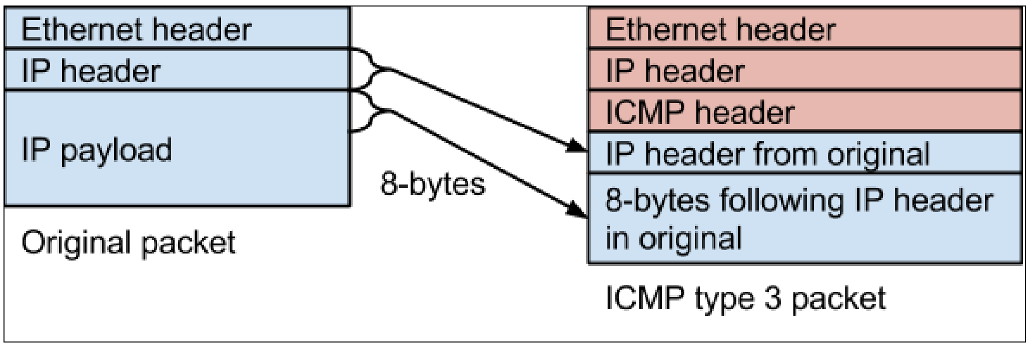
For this assignment, you will implement both the control and data planes of a simple IP router. Your control plane will implement a distance vector routing protocol (RIP) to build a forwarding table. Your data plane will forward packets based on entries in the forwarding table, and, when necessary, generate ICMP messages to notify packet senders of forwarding problems.

Clarifications

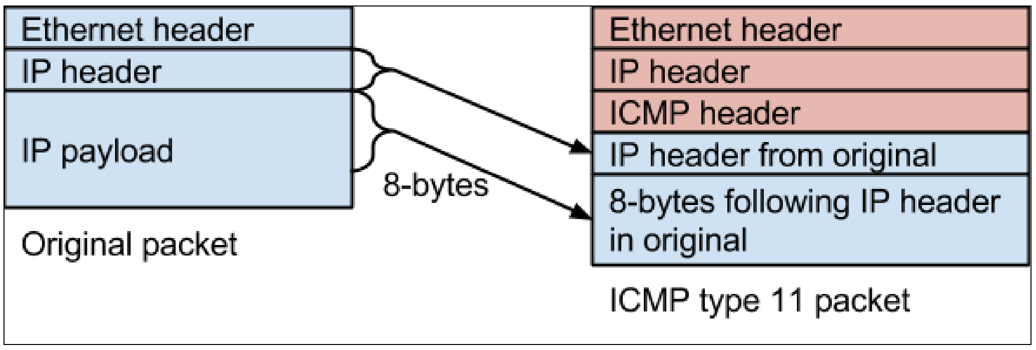
- **ARP lookups:** When your router sends IP packets, the packets must have an Ethernet header with the appropriate source and destination MAC addresses. The destination MAC address should be set according to the next hop IP address for the packet. ARP is used to determine the MAC address associated with a given IP address. If a lookup in the ARP table is successful, then the destination MAC address can be set in the Ethernet header and the packet can be sent immediately. Otherwise, the packet must be queued (by calling the `waitForArp(...)` method in the `ArpCache` class). Assuming the ARP resolution eventually succeeds, any packets waiting for the resolution must be updated to have the appropriate destination MAC address in their Ethernet header, and then be sent.
- **ICMP packets:** All ICMP packets must contain an Ethernet header, IP header, and ICMP header. Depending on the type of ICMP packet, extra data may also need to be included after the ICMP header.
  If the router receives an ICMP echo request sent to the IP addresses of one of its interfaces, then the echo reply sent in response must contain the ICMP payload from the echo request. This is illustrated in the figure below:


Ethernet header
IP header
ICMP header (type=8, code=0, checksum)
ICMP payload (identifier, seq #, other data)
Echo Request
Ethernet header
IP header
ICMP header (type=0, code=0, checksum)
ICMP payload from request
Echo Reply

When the router generates ICMP messages for destination net unreachable (type 3, code 0), destination host unreachable (type 3, code 1), or port unreachable (type 3, code 3), the packet must contain the following after the ICMP header: (1) the original IP header from the packet that triggered the error message, and (2) the 8 bytes following the IP header in the original packet. This is illustrated in the figure below:

- 

When the router generates ICMP messages for time exceeded (type 11, code 0), the packet must contain the following after the ICMP header: (1) the original IP header from the packet that triggered the error message, and (2) the 8 bytes following the IP header in the original packet. This is illustrated in the figure below:



- **Checksums:** The IP checksum should only be computed over the IP header. The length of the IP header can be determined from the header length field in the IP header, which specifies the length of the IP header in 4-byte words

(i.e., multiple the header length field by 4 to get the length of the IP header in bytes). The checksum field in the IP header should be zeroed before calculating the IP checksum.
The ICMP checksum should be calculated over the ICMP header and payload (i.e., any data that follows the ICMP header). The checksum field in the ICMP header should be zeroed before calculating the ICMP checksum.

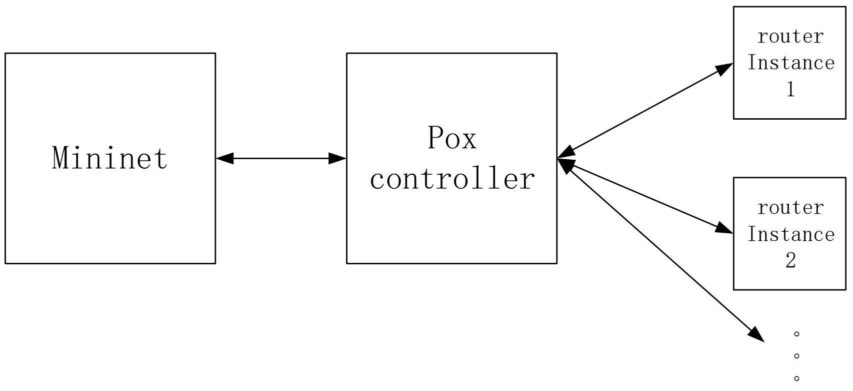
- **POX:** POX is used “under the hood” in this assignment to direct packets between Mininet and your router instances (i.e., Java processes). You do not need to understand, modify, or interact with POX in any way, besides executing the `run_pox.sh` script.

Part 1: Getting Started

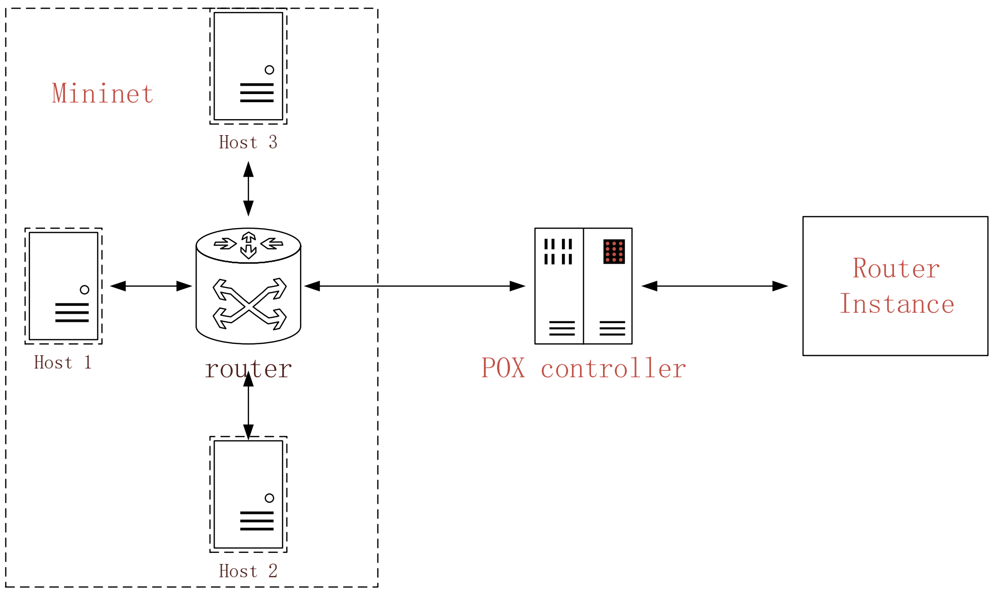
You will be using Mininet, POX, and skeleton code for a simple router to complete the project.

Platform introduction

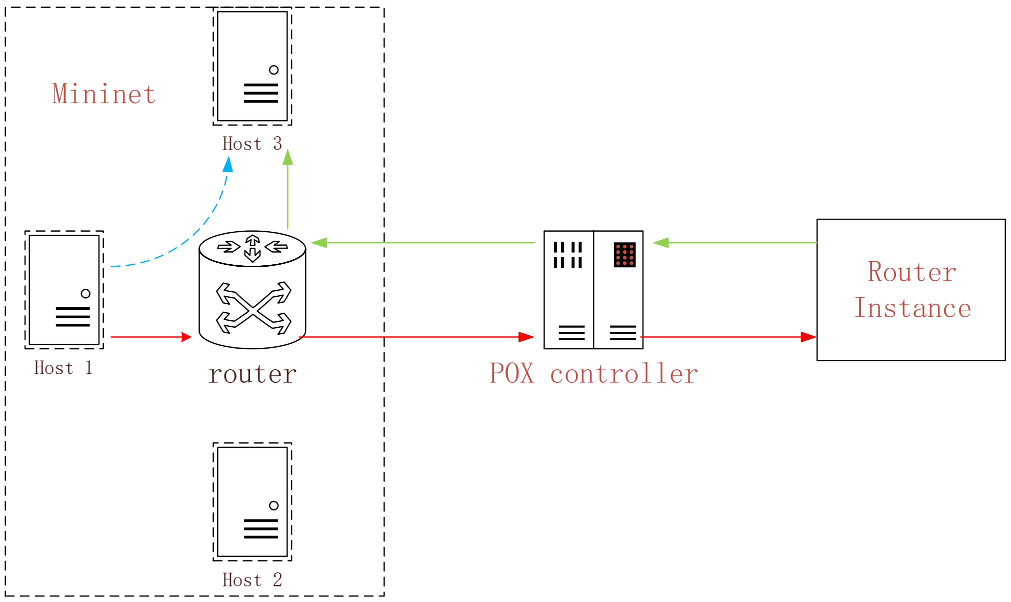
Mininet creates a realistic virtual network on a single machine.
POX is a controller platform written in python. It can tell the router to forward the arrived packets to which port, or drop them.
In this program, Mininet connects to POX controller, and POX controller connects to router instances. One instance controls one router in Mininet.



An example is shown in the following figure which has only one router.



First, Mininet creates a virtual network which has one router and three hosts. Then Mininet connects to POX controller. POX connects to one router instance. If there are more than one router in Mininet, you will need the same number router instances.



When Host 1 wants to send a packet to Host 3 (blue line). First, it sends a packet to router (red line). When router gets the packet, router sends it to POX controller. POX controller sends the packet to the corresponding Router Instance. The Router Instance knows that it should send the packet to Host 3 according to the route table. Then the packet arrives at POX controller, then the virtual router, then Host 3.
What you have to finish is the Router Instance code.

Reference

Mininet http://mininet.org/
POX https://github.com/noxrepo/pox

POX and Mininet installation http://www.sdnap.com/sdnap-post/1270.html

Preparing Your Environment

Before beginning this project, there are some additional steps you need to complete to prepare your VM:

1. Install required packages

```
sudo apt-get update
sudo apt-get install -y python-dev vim-nox python-
   setuptools flex bison traceroute ant openjdk-7-jdk
   git
```

2. Install ltprotocol

```
cd ~
git clone git://github.com/dound/ltprotocol.git
cd ltprotocol
sudo python setup.py install
```

3. Checkout the appropriate version of POX

```
cd ~
git clone https://github.com/noxrepo/pox
cd ~/pox
git checkout f95dd1a81584d716823bbf565fa68254416af603
```

4. Install mininet

```
cd ~
sudo apt-get install mininet
```

5. Download the starter code from course website

```
cd ~
wget http://cs.wisc.edu/~akella/CS640/F14/assign2/code.tgz
tar xzvf code.tgz
```

6. Symlink POX and configure the POX modules

```
cd ~/project2
ln -s ../pox
./config.sh
```

7. **test mininet and pox**

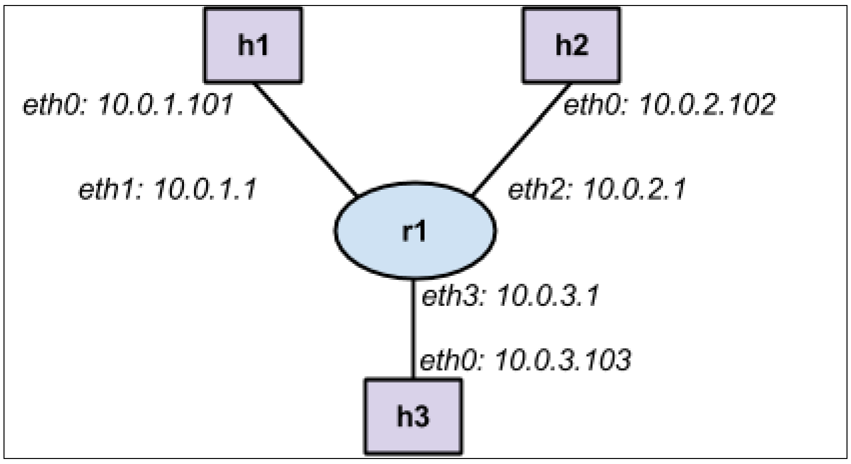
```
cd ~/pox
./pox.py forwarding.l2_learning
```

Start a new terminal

```
sudo mn --controller=remote,ip=127.0.0.1,port=6633
```

Sample Configuration

The first sample configuration consists of a single router (`r1`) and three emulated hosts (`h1`, `h2`, and `h3`). The hosts are each running an HTTP server. When you have finished implementing your router's data plane, one host should be able to fetch a web page from any other host (using wget or curl). Additionally, the hosts should be able to ping and traceroute each other, as well as the router.



This topology is defined in the configuration file `single.topo`.

Running the Simple Router

1. Start Mininet emulation by running the following commands:

```
$ cd ~/project2/
$ ./run_mininet.sh single.topo
```

You should be able to see some output that ends like the following:

```
*** Adding links:
(h1, r1) (h2, r1) (h3, r1)
*** Configuring hosts
h1 h2 h3
*** Starting controller
*** Starting 1 switches
r1
*** Configuring host h1
Setting ip for h1-eth0 to 10.0.1.101/24
Adding route 10.0.1.1/32 dev h1-eth0
```

```
Deleting route 10.0.1.0/24 dev h1-eth0
Adding default gw 10.0.1.1 dev h1-eth0
*** Starting SimpleHTTPServer on host h1
*** Configuring host h2
Setting ip for h2-eth0 to 10.0.2.102/24
Adding route 10.0.2.1/32 dev h2-eth0
Deleting route 10.0.2.0/24 dev h2-eth0
Adding default gw 10.0.2.1 dev h2-eth0
*** Starting SimpleHTTPServer on host h2
*** Configuring host h3
Setting ip for h3-eth0 to 10.0.3.103/24
Adding route 10.0.3.2/32 dev h3-eth0
Deleting route 10.0.3.0/24 dev h3-eth0
Adding default gw 10.0.3.2 dev h3-eth0
*** Starting SimpleHTTPServer on host h3
*** Starting CLI:
mininet>
```

Keep this terminal open, as you will need the mininet command line for debugging. Use another terminal to continue the next step. (Don't press `ctrl-z`.)

2. Start the controller, by running the following commands:

```
cd ~/project2/
./run_pox.sh
```

You should be able to see some output like the following:

```
POX 0.0.0 / Copyright 2011 James McCauley
DEBUG:.home.mininet.cs640.project2.pox_module.cs640.ofh
andler:*** ofhandler: Successfully loaded ip settings
for hosts
[…]
Ready.
POX>
```

You have to wait for Mininet to connect to the POX controller before you continue to the next step. Once Mininet has connected, you will see output like the following:

```
INFO:openflow.of_01:[Con 1/1] Connected to 00-00-00-00-
00-01
DEBUG:.home.mininet.cs640.project2.pox_module.cs640.ofh
andler:Connection [Con 1/1]
DEBUG:.home.mininet.cs640.project2.pox_module.cs640.ofh
andler:dpid=1
DEBUG:.home.mininet.cs640.project2.pox_module.cs640.srh
andler:cs640_srhandler catch RouterInfo(info={'eth3':
('10.0.3.2', '16:83:f7:de:00:f5', '10Gbps', 3), 'eth2':
```

```
('10.0.2.1', '86:39:42:d4:30:96', '10Gbps', 2), 'eth1':
('10.0.1.1', '92:23:9a:41:a2:39', '10Gbps', 1)},
rtable=[], swid=r1, dpid=1)
INFO:.home.mininet.cs640.project2.pox_module.cs640.srha
ndler:created server
DEBUG:.home.mininet.cs640.project2.pox_module.cs640.srh
andler:SRServerListener listening on 8001
```

Keep POX running. Open yet another terminal to continue the next step. (Don't press `ctrl-z`.)

3. Build and start the simple router, by running the following commands:

```
cd ~/project2/
ant
java -jar SimpleRouter.jar -p 8001 -v r1 -r rtable.r1
```

You should be able to see some output that ends like the following:

```
Loading routing table
---------------------------------------------
Destination        Gateway          Mask
Iface
10.0.1.101         0.0.0.0              255.255.255.255
eth1
10.0.2.102         0.0.0.0              255.255.255.255
eth2
10.0.3.103         0.0.0.0              255.255.255.255
eth3
---------------------------------------------
Router interfaces:
eth3       HWaddr 16:83:F7:DE:00:F5
          inet addr 10.0.3.2
eth1       HWaddr 92:23:9A:41:A2:39
          inet addr 10.0.1.1
eth2       HWaddr 86:39:42:D4:30:96
          inet addr 10.0.2.1
<-- Ready to process packets -->
```

4. Go back to the terminal where Mininet is running. To issue an command on the emulated host, type the hostname followed by the command in the Mininet console. For example, the following command issues 2 pings from the `h1` to `h2`:

```
mininet> h1 ping -c 2 10.0.2.102
```

The pings will fail because the router’s data plane is not fully implemented. However, in the terminal where your simple router is running, you should see the following output:

```
*** -> Received packet:
         arp
         dl_vlan: untagged
         dl_vlan_pcp: 0
         dl_src: c6:cb:bb:81:8e:57
         dl_dst: ff:ff:ff:ff:ff:ff
         nw_src: 10.0.1.101
         nw_dst: 10.0.1.1
*** -> Received packet:
         arp
         dl_vlan: untagged
         dl_vlan_pcp: 0
         dl_src: c6:cb:bb:81:8e:57
         dl_dst: ff:ff:ff:ff:ff:ff
         nw_src: 10.0.1.101
         nw_dst: 10.0.1.1
```

This indicates packets are arriving at the router and need to be processed by the router's data plane.

5. You can stop your simple router by pressing `ctrl-c` in the terminal where it's running. You can restart the simple router without restarting POX and mininet, but it's often useful to restart POX and mininet to ensure the emulated network starts in a clean state.

Part 2: Code Overview

The simple router code consists of four packages:

- `edu.wisc.cs.sdn.sr` — code for the router's structures (interfaces, route table, ARP cache, etc.) and functionality (packet forwarding, ARP resolution, etc.)
- `edu.wisc.cs.sdn.sr.vns` — code to communicate with POX and mininet
- `net.floodlight.packet` — code for parsing and creating packets with various protocol headers
- `net.floodlight.util` — code for representing MAC addresses

You should only need to modify (and possibly create) classes in the `edu.wisc.cs.sdn.sr` package to complete this assignment.

`Router.java`

The `Router` class has the full context for the router (interfaces, route table, ARP cache, etc.) and the primary functions for receiving and sending packets.
The `Router` class has several important class variables:

- `Map<String,Iface> interfaces` — list of the router's interfaces; maps interface name's to`Iface` objects which contain details about an interface (e.g., the interface's IP and MAC addresses)
- `RouteTable routeTable` — routing table for the router
- `ArpCache arpCache` — ARP cache for the router (i.e., a cache of IP address to MAC address mappings)

When packets are received on any of a router's interfaces, the `handlePacket(...)` function is called (by the underlying code that communicates with POX and mininet). In the code we provide, this `handlePacket` simply prints out the received packet, but you'll add code to this function to either forward the packet out another interface, pass the packet to the ARP or RIP subsystems, or respond with an ICMP packet. Part 3 discusses this in more detail.
You can send a packet out one of the router's interfaces by calling the `sendPacket(...)` function in the`Router` class.

`Iface.java`

There is an `Iface` object for each interface on the router (stored in the `interfaces` variable in the `Router` object). Each interface has a name, MAC address, and IP address.

`RouteTable.java` & RouteTableEntry`.java`

The router maintains its route table in a `RouteTable` object (referenced via the `routeTable` variable in the`Router` object). A `RouteTable` object has a list of route entries (the class variable `entries`).
You can request the list of entries by calling the `getEntries(...)` function. You can add, update, and remove entries, by calling the `addEntry(...)`, `updateEntry(...)`, and `removeEntry(...)` functions, respectively. When you are implementing the router's data plane (Part 3 of the assignment), you will use a static routing table that is automatically read from a file, so you will not need to add, update, and remove entries. You will only need these functions when you implement the router's control plane (Part 4 of the assignment).
A `RouteTableEntry` object has a destination address, gateway address, mask address, and interface name.

`ARPCache.java`, `ARPEntry.java` & `ArpRequest.java`

The router maintains a mapping of IP address to MAC addresses in an `ARPCache` object (referenced via the`arpCache` variable in the `Router` object). An `ARPCache` object has a list of cache entries (the class variable`entries` maps an IP address to an entry) and a list of pending requests to determine the MAC address corresponding to a particular IP address (the class variable `requests` maps an IP address to a request).
You can check if there is an entry for an IP address in the ARP cache by calling the `lookup(...)` function. If no entry is found, then you can call `waitForArp(...)` to send ARP request packets into the network to determine the MAC address for a given IP address. The packet that needs the mapping will be placed on a queue until an ARP response packet is received. Once the ARP response is received, you will need to send any packets that were waiting on the mapping by completing the `handleArpPacket(...)` function in the`Router` class. If five ARP requests are sent for a particular IP address without and no response has been received, then the router gives up on trying to resolve the packets. An ICMP error message should be sent for any packets that were waiting on the response; you'll need to complete the`updateArpRequest(...)` function in the `ARPCache` class to send these ICMP packets.
Note that there is already a thread (started by the constructor in the ARPCache class) that: sends a new ARP request packet every second for any pending requests, and times out entries from the ARP cache that are more than 15 seconds old.

`RIP.java`

If a static route table is not provided (via the `-r` argument when running `SimpleRouter.jar`), then the router's control plane uses the Routing Information Protocol (RIP) to build a route table. The `RIP` class we provide contains three functions that can be used as a starting point for implementing RIP (described in more detail inPart 4):

- `init()` — populates the route table with entries for the subnets that are directly reachable via the router's interfaces, starts a thread for period RIP tasks, and performs other initialization for RIP as necessary
- `handlePacket(...)` — processes a RIP packet that is received by the router
- `run()` — sends periodic route updates to neighbors, and times out route table entries that neighbors last advertised more than 30 seconds ago

---

Part 3: Implement Router Data Plane

Your router's data plane must:

1. Forward IP packets based on the route table
2. Invoke the appropriate control plane code for control packets (ARP and RIP)
3. Respond to ping packets destined for the router itself
4. Send ICMP messages when error conditions occur (e.g., no matching route table entry, or MAC address resolution (i.e., ARP) timeouts)

Your code to provide this functionality should go in the `handlePacket(...)` function, and other helper functions you create, in the `Router` class.
Given a raw Ethernet frame, it may contain an ARP packet or an IP packet.

IP Forwarding

If the frame contains an **IP packet that is not destined for one of our interfaces**:

- Check the packet has the correct checksum.
- Decrement the TTL by 1.
- Find out which entry in the routing table has the longest prefix match with the destination IP address.
- Check the ARP cache for the next-hop MAC address corresponding to the next-hop IP. If it's there, send the packet. Otherwise, call `waitForArp(...)` function in the `ARPCache` class to send an ARP request for the next-hop IP, and add the packet to the queue of packets waiting on this ARP request.

If an error occurs in any of the above steps, you will have to send an ICMP message back to the sender notifying them of an error.

You can start an web server in a virtual host:

```
   mininet> h1 python -m SimpleHTTPServer 80 &
```

And use "wget" command to test your router

```
   Mininet> h2 wget http://10.0.1.101
```

If it can download "index.html", your router works.

Invoking Control Plane Code & Responding to Pings

If the frame contains an **IP packet destined for one of your router's interfaces** (i.e., destined to an IP address assigned to one of your router's interfaces):

- If the packet is an ICMP echo request and its checksum is valid, send an ICMP echo reply to the sending host.

- If the packet contains a UDP packet destined for port 520, call the your control plane function that handles RIP packets (details in Part 4).
- If the packet contains a TCP payload or a UDP packet destined for any port other than 520, send an ICMP *port unreachable (ICMP type 3, code 3)* packet to the sending host.
- Otherwise, ignore the IP packet.

If the frame contains an **ARP packet**: call the `handleArpPacket(...)` function in the `Router` class. You will need to complete the `TODO` block in the `handleArpPacket(...)` function to send any waiting packets after an ARP request is successfully resolved (i.e., the MAC address associated with an IP address is determined).

Internet Control Message Protocol (ICMP)

You will need to properly **generate the following ICMP messages** in response to the sending host under the following conditions:

- Echo reply (type 0): Sent in response to an echo request (ping) to one of the router's interfaces. (This is only for echo requests to any of the router's IPs. An echo request sent elsewhere should be forwarded to the next hop address as usual.)
- Destination net unreachable (type 3, code 0): Sent if there is a non-existent route to the destination IP (no matching entry in routing table when forwarding an IP packet).
- Destination host unreachable (type 3, code 1): Sent if five ARP requests were sent to the next-hop IP without a response. This should happen in the `TODO` block in the `updateArpRequest(...)` function in the `ArpCache` class.
- Port unreachable (type 3, code 3): Sent if an IP packet containing a TCP or a UDP payload (except a UDP payload on port 520) is sent to one of the router's interfaces. This is needed for traceroute to work.
- Time exceeded (type 11, code 0): Sent if an IP packet is discarded during processing because the TTL field is 0. This is also needed for traceroute to work.

The source address of an ICMP message can be the source address of any of the incoming interfaces, as specified in RFC 792. As mentioned above, the only incoming ICMP message destined towards the router's IPs that you have to explicitly process are ICMP echo requests. Network Sorcery's RFC Sourcebook provides a condensed reference for ICMP.
For examples of how to construct and send packets, see the `sendArpRequest(...)` and `sendArpReply(...)` functions in the `ArpCache` class.

Part 4: Implement Router Control Plane

Your router's control plane must implement version 2 of the Routing Information Protocol (RIPv2) to build, and update your, router's route table. RFC 2453 is the latest specification for RIPv2; Network Sorcery's RFC Sourcebook provides a condensed reference for RIP.
Note that the RIP only operate when a static route table is not provided (via the `-r` argument when running`SimpleRouter.jar`). The `init()` function of the `RIP` class we provide checks if the routing table has already been populated before initializing RIP.

RIP Packets

The `RIPv2` and `RIPv2Entry` classes in the `net.floodlightcontroller.packet` package define the format for RIPv2 packets. All RIPv2 packets should be encapsulated in UDP packets whose source and destination ports are 520 (defined as a constant `RIP_PORT` in the `net.floodlightcontroller.packet.UDP` class). When sending RIP requests and unsolicited RIP responses, the destination IP address should be the multicast IP address reserved for RIP (*224.0.0.9*, defined as a constant `RIP_MULTICAST_IP` in the `RIP` class) and the destination Ethernet address should be the broadcast MAC address *FF:FF:FF:FF:FF:FF* (defined as a constant `BROADCAST_MAC` in the `RIP` class). When sending a RIP response for a specific RIP request, the destination IP address and destination Ethernet address should be the IP address and MAC address of the router that sent the request.

RIP Operation

Your router should send a RIP request out all of the router's interfaces when RIP is initialized; do this in the `init()` function in the `RIP` class. Your router should send an unsolicited RIP response out all of the router's interfaces every 10 seconds thereafter (defined as a constant `UPDATE_INTERVAL` in the `RIP` class); do this in the `run()` function in the `RIP` class.
When a RIP request or RIP response is received by your router, your data plane should invoke the `handlePacket(...)` function in the `RIP` class (as described in Part 3). Your router should update its route table based on these packets, and send any necessary RIP response packets (solicited or unsolicited).
Your router should time out route table entries for which an update has not been received for more than 30 seconds (defined as a constant `TIMEOUT` in the `RIP` class). You should never remove route entries for the subnets that are directly reachable via the router's interfaces (i.e., the route table entries added by the `init()` function in the `RIP` class we provide).
Your router must implement the split horizon strategy to help combat the count to infinity problem. Assume a maximum hop count of 16.

Testing RIP

To test your router's control plane, you will need a topology with more than one router. We have provided two such topologies: `pair.topo` and `triangle.topo`. Provide one of these alternative files to the`run_mininet.sh` script to use one of these topologies.
You will need to start an instance of your simple router for each router in the topology. For the first router, use port 8001 and host r1:

```
  java -jar SimpleRouter.jar -p 8001 -v r1
```

For the second router, use port 8002 and host r2:

```
  java -jar SimpleRouter.jar -p 8002 -v r2
```

For the third router, use port 8003 and host r3:

```
  java -jar SimpleRouter.jar -p 8003 -v r3
```

And so on. You will need to run each in a separate terminal window. You should not include the `-r` option when testing your control plane code, since we want the router to build the route table rather than reading it from a file.

Submission Instructions

You must submit a single tar file of the `src` directory containing the Java source files for your simple router. Please submit the entire `src` directory; do not submit any other files or directories.

```
 tar czvf yourname.tgz src
```

Grading

1) Correctness 70%.
    Arp protocol 10
    IPv4 protocol 25
    ICMP protocol 25
    RIP protocol 10
2) Report 25%
3) Code quality 5%

Acknowledgements

This programming assignment borrows from the *Simple Router* assignment from University of Wisconsin-Madison CS 640: Computer Networks.