Lab 3: Simple Router

CS144 Lab 3 section
October 23rd, 2009
Tom Wiltzius

Based on slides by Derrick Isaacson, Ben Nham, Clay Collier, and Martin Casado
Assignment Overview

• You use a virtual network topology as part of the VNS system
• You write a router in C
• Your router will route real IP packets over the Internet from standard clients (i.e. ping, traceroute, ftp, Firefox...)
• Due Thursday, Oct 29th
• Get started!
Getting Started

• Get the assignment code from:
  http://www.scs.stanford.edu/09au-cs144/lab/router.tar.gz

• Look in the INSTRUCTIONS file for info on
  – Using your assigned topology
  – What the default topology looks like
  – Information on the routing table for your topology
Protocols You Need to Handle

• Ethernet
  – All packets given to you are raw Ethernet frames
• IP
• ARP
  – Needed to resolve IP addresses to MAC addresses
• ICMP requests/replies
  – Used by some programs to send requests (ping)
  – Needed to send control messages back to host
• See sr_protocol.h and Network Sorcery to deal with the raw bits
• Make sure you understand your pointer arithmetic!
Router Basics

ftp 192.168.128.51

192.168.128.51

Network

vns-firewall

Router

eth0:

192.168.128.6

192.168.128.50

eth1:

192.168.128.51

eth2:

192.168.129.106

192.168.129.107

eth0:

192.168.128.51

eth2:

192.168.129.106

192.168.129.107
Routing Table

- Static routing table in this assignment
- Make sure you understand LPM
- Loaded for you from the command line into the router context of type `struct sr_instance` (sr_router.h)
- Entries are a linked list of type `struct sr_rt` (sr_rt.h)
- Walk over the routing table linearly to do a longest prefix match on it

<table>
<thead>
<tr>
<th>IP address</th>
<th>Next-hop</th>
<th>Network Mask</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>192.168.128.51</td>
<td>192.168.128.51</td>
<td>255.255.255.255</td>
<td>eth1</td>
</tr>
<tr>
<td>192.168.129.106</td>
<td>192.168.129.106</td>
<td>255.255.255.255</td>
<td>eth2</td>
</tr>
<tr>
<td>0.0.0.0</td>
<td>172.24.74.17</td>
<td>0.0.0.0</td>
<td>eth0</td>
</tr>
</tbody>
</table>
Basic Forwarding Principles

• Remove IP datagram from Ethernet frame.
  – Could also be an ARP payload in Ethernet frame
• Verify checksum. If it’s not valid, drop the packet.
• Inspect the packet’s DA. Assuming the packet is destined for someone else:
  – Look up next-hop address by doing a LPM on the routing table using the packet’s DA
    • If it does not exist, send ICMP host unreachable
  – Decrement TTL, update header checksum
    • If TTL == 0 after decrementing, send ICMP time exceeded
  – From next-hop IP address, determine outgoing interface and next-hop MAC address
    • If necessary, send ARP request to determine MAC address
  – Encapsulate IP datagram in Ethernet packet
  – Forward packet to outgoing interface
Packets Destined for the Router

• If the packet’s DA is destined towards one of our interfaces:
  – If it’s an ICMP echo request, generate an ICMP echo reply
  – Otherwise if it’s a TCP or UDP packet, generate an ICMP port unreachable (needed for traceroute to work)
Checksums

- **IP checksum**
  - Need to check for all IP packet headers; drop packet if checksum is bad

- **ICMP checksum**
  - Need to validate for incoming packets destined for us
  - Need to calculate for outgoing packets
  - Ignore if forwarding

- **TCP/UDP checksum**
  - End-to-end checksum, ignore

- Use cksum func from previous lab to calculate checksums
IP Addresses and MAC Addresses

- Each IP address is associated with a single interface
- Each interface has a MAC address
- Therefore, each IP address is associated with a single MAC address
- Router with 4 interfaces has 4 IPs and 4 MAC addresses!
  - eth0 has some IP\(_0\)
  - eth1 has some IP\(_1\)
  - eth2 has some IP\(_2\)
  - eth3 has some IP\(_3\)
- Conclusion: An IP address names an *interface*, not a *host*
IP Within Ethernet

- In a router, we operate on raw Ethernet frames, with an IP or ARP packet as payload
- To forward a packet one hop, must know:
  - Destination IP address
  - Next-hop MAC address of the next-hop IP address

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dest MAC Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dest MAC Address (continued)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source MAC Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source MAC Address (continued)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ethertype</td>
<td>Payload</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Necessity of ARP

• Suppose I get a packet destined for 128.12.94.3
  – Matches default route only (first entry)

• Need to send raw Ethernet frame to the MAC address of 5.10.1.1

• But our routing table has only IP addresses, not MAC addresses

### Table

<table>
<thead>
<tr>
<th>IP Address</th>
<th>Netmask</th>
<th>Gateway</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0.0.0</td>
<td>0.0.0.0</td>
<td>5.10.1.1</td>
<td>eth0</td>
</tr>
<tr>
<td>12.1.0.0</td>
<td>255.255.0.0</td>
<td>12.1.0.1</td>
<td>eth1</td>
</tr>
<tr>
<td>12.1.1.0</td>
<td>255.255.255</td>
<td>12.1.1.1</td>
<td>eth2</td>
</tr>
<tr>
<td>12.1.1.200</td>
<td>255.255.255.255</td>
<td>12.1.1.200</td>
<td>eth3</td>
</tr>
</tbody>
</table>
 ARP Packet Format

- Corresponds to struct sr_arphdr
  - HTYPE = ar_hrd = htons(ARPHDR_ETHER)
  - PTYPE = ar_pro = htons(ETHERTYPE_IP)
  - HLEN = ar_hln = # bytes in MAC address
  - PLEN = ar_pln = # bytes in IP address
  - OPER = ar_op = htons(ARP_REQUEST) or htons(ARP_REPLY)
  - SHA/SPA = sender MAC/IP address (copy from sr_if of outgoing IF)
  - THA/TPA = target MAC/IP address

<table>
<thead>
<tr>
<th>+</th>
<th>0  - 7</th>
<th>8 - 15</th>
<th>16 - 31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Hardware type (HTYPE)</td>
<td>Protocol type (PTYPE)</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Hardware length (HLEN)</td>
<td>Protocol length (PLEN)</td>
<td>Operation (OPER)</td>
</tr>
<tr>
<td>64</td>
<td>Sender hardware address (SHA)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>?</td>
<td>Sender protocol address (SPA)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>?</td>
<td>Target hardware address (THA)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>?</td>
<td>Target protocol address (TPA)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Generating ARP Request

• Request: Who has IP 192.168.1.3? Create ARP request with fields:
  – Source HW addr: MAC_{src}
  – Source protocol addr: IP_{src}
  – Target protocol addr: 192.168.1.3

• ARP requests are sent to the Ethernet broadcast address
Handling ARP Request

• Get request: Who has 192.168.1.3
• If one of the IPs of my router is 192.168.1.3, send an ARP reply: I have IP of 192.168.1.3 with MAC address of 00-11-22-33-44-55-66 (six bytes in hex, or 48 bits)
  – Source HW addr: 00-11-22-33-44-55-66
  – Source protocol address: 192.168.1.3
  – Target HW addr: MAC_{src}
  – Target protocol addr: IP_{src}
• ARP reply is sent directly to MAC_{src}
Handling ARP Reply

• Reply: I have IP of 192.168.1.3 with MAC address of 00-11-22-33-44-55-66

• If the target IP of the ARP reply is the IP of the interface this came in on:
  – Add the IP to MAC mapping to the ARP cache
  – Send off any packets that were waiting on this ARP reply
Lab 3 ARP Functionality

• Generate ARP requests if a next hop MAC is not in the ARP cache
• Process ARP replies that target us and place them into the ARP cache
• Process ARP requests that target us and generate an appropriate ARP reply
• Go through ARP request queue every second and send off ARP requests every second (edit sr_arpcache_sweepreqs in sr_arpcache.c)
Handling ARP

• Routing table contains next-hop IPs, but you need both a next-hop IP and a next-hop MAC address

• What you have to do:
  – Generate ARP requests and parse ARP replies
  – Listen to ARP requests and send ARP replies
  – Don’t send a request for each packet; instead, use an ARP cache
  – Requests should time out after 5 tries of about 1 second each
  – ARP cache entries should time out after about 15 seconds

<table>
<thead>
<tr>
<th>ARP Cache</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IP address</td>
<td>Ethernet MAC Address</td>
</tr>
<tr>
<td>172.24.74.130</td>
<td>00:e0:81:04:08:9b</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
ARP Cache Class

• We’ve given you an ARP cache class containing
  – ARP request queue
  – ARP cache
• ARP cache entries time out automatically after 15 seconds
• The cache class is essentially two linked lists: one for the cache, and one for requests. There are functions to handle querying and inserting into these two lists.
• ARP request queue written so that it is easy to consolidate ARP requests across multiple packets, and enforce the ARP request timeout
• See pseudocode in sr_arpcache.h for more detailed info
ICMP

- Used to send control messages back to sending host
- Must process ICMP Echo Request
- Must generate
  - ICMP Echo Reply
  - ICMP Destination Host Unreachable
  - ICMP Destination Port Unreachable
  - ICMP Time Exceeded
- Check Network Sorcery pages on details of how to format these messages
Basic Lab 4 Flow Diagram

Receive Raw Ethernet Frame

Process IP Packet
- Process Packet
  - Sent to self
    - If ICMP echo req, send ICMP echo reply
    - If IP+TCP/UDP, send ICMP port unreachable
  - Not sent to self
    - Forward Packet

Forward Packet
- Do LPM on routing table
- Match
  - Check ARP cache
    - Hit
      - Send frame to next hop
    - No hit
      - Add request to ARP queue, sending if necessary
- No match
  - Send ICMP host unreachable

Process ARP Request/Reply
- See previous slides
More Lab 4 Thoughts

• Organize your code
  – Sticking everything sr_router.c will probably give you a headache
  – Make some new files (suggestions, not necessary):
    • sr_arp.c/h for handling/generating ARP packets
    • sr_icmp.c/h for handling/generating ICMP packets
    • sr_ip.c/h for to handle generating IP packets
    • Add the sources and headers to the Makefile

• Do one thing at a time
  – Need ARP to send anything at all
  – If you just do the forwarding path without ICMP, should be able to route packets to the app servers
  – Can add ICMP support last
Required Functionality

• Forwarding packets should work
• Handles and generates ARP requests/replies correctly
• You can download a file using http and ftp from one of the app servers behind the router
• You can traceroute (tracepath) to and through the router
• You can ping to and through the router
• ICMP host and port unreachable messages are generated correctly
• ICMP time exceeded messages are generated correctly
• No shortcuts taken (don’t forward a packet to every interface)
• Should not crash, even with a malformed packet
Main Functions and Structures

• In `sr_router.h`:
  – `struct sr_instance` is the context of the router

• In `sr_router.c`:
  – `sr_handlepacket` is called for every packet that goes through the router—you have to fill it out

• `sr_protocol.h` contains convenience structs for accessing fields in packets
  – Note: only the basic ICMP header is provided; `sr_protocol.h` doesn’t include structs for all the various ICMP packet types you’ll need so you’ll have to make your own

• `sr_if.h` contains methods for getting information about the router’s interfaces

• ARP Cache in `sr_arpcache.h`
Suggestions

• Think before coding—there’s a good amount of code to write
• Read through the Network Sorcery pages to understand the protocols
• Read the FAQ
• Post and check the newsgroup
• Start early