The Journey of a Packet Through the Linux Network Stack

... plus hints on Lab 9
Some Words

- Assume IP version 4
- Codes are from Kernel 2.6.9.EL (use in Lab 9)
- Ideas are similar
Linux High-Level Network Stack

- Interface to users
- TCP/UDP/IP etc...
- Queue for device
Receiving a Packet (Device)

- **Network card**
  - receives a frame
  - issues an *interrupt*

- **Driver**
  - handles the *interrupt*
    - Frame $\rightarrow$ RAM
    - Allocates `sk_buff` (called `skb`)
    - Frame $\rightarrow$ skb
Aside: `sk_buff` *(skbuff.h)*

- Generic buffer for all packets
- Pointers to `skb` are passed up/down
- Can be linked together
`sk_buff` (cont.)

<table>
<thead>
<tr>
<th><strong>struct sk_buff</strong></th>
<th><em>next</em></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>struct sk_buff</strong></td>
<td><em>prev</em></td>
</tr>
<tr>
<td><strong>struct sk_buff_head</strong></td>
<td><em>list</em></td>
</tr>
<tr>
<td><strong>struct sock</strong></td>
<td><em>sk</em></td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>union {tcphdr; udphdr; ...}</td>
<td>h;</td>
</tr>
<tr>
<td>union {iph; ipv6h; arph; ...}</td>
<td>nh;</td>
</tr>
<tr>
<td>union {raw}</td>
<td>mac;</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

- **Transport Header**
- **Network Header**
- **MAC Header**
- **DATA**
sk_buff (cont.)

Image from “Understanding Linux Network Internals”, Christian Benvenuti
Receiving a Packet (Device)

- Driver (cont.)
  - calls device independent
    - `core/dev.c:netif_rx(skb)`
      - puts skb into CPU queue
      - issues a “soft” interrupt

- CPU
  - calls `core/dev.c:net_rx_action()`
    - removes skb from CPU queue
    - passes to network layer e.g. ip/arp
    - In this case: IPv4 `ipv4/ip_input.c:ip_rcv()`
Receiving a Packet (IP)

- **ip_input.c:ip_rcv()**

  **checks**
  - Length $\geq$ IP Header *(20 bytes)*
  - Version == 4
  - Checksum
  - Check length again

- **calls**
  - `ip_rcv_finish()`

- **calls**
  - `route.c:ip_route_input()`
Aside: Finish/Slow suffix

- Division into two stages is common
- Usually called “slow”

<table>
<thead>
<tr>
<th>The first stage</th>
<th>cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>The second stage</td>
<td>table</td>
</tr>
</tbody>
</table>
Receiving a Packet (routing)

- `ipv4/route.c:ip_route_input()`

<table>
<thead>
<tr>
<th>Destination == me?</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td><code>ip_input.c:ip_local_deliver()</code></td>
</tr>
<tr>
<td>NO</td>
<td>Calls <code>ip_route_input_slow()</code></td>
</tr>
</tbody>
</table>

- `ipv4/route.c:ip_route_input_slow()`

<table>
<thead>
<tr>
<th>Can forward?</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Forwarding enabled?</td>
<td></td>
</tr>
<tr>
<td>• Know route?</td>
<td></td>
</tr>
<tr>
<td>NO</td>
<td>Sends ICMP</td>
</tr>
</tbody>
</table>

Forwarding a Packet

- Forwarding is per-device basis
  - Receiving device!
- Enable/Disable forwarding in Linux:
  - Kernel
  - `/proc` file system ↔ Kernel
  - read/write normally (in most cases)

```
• /proc/sys/net/ipv4/conf/<device>/forwarding
• /proc/sys/net/ipv4/conf/default/forwarding
• /proc/sys/net/ipv4/ip_forwarding
```
Forwarding a Packet (cont.)

- `ipv4/ip_forward.c:ip_forward()`

<table>
<thead>
<tr>
<th>IP TTL &gt; 1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>Decreases TTL</td>
</tr>
<tr>
<td>NO</td>
<td>Sends ICMP</td>
</tr>
</tbody>
</table>

- `core/dev.c:dev_queue_xmit()`
- Default queue: priority FIFO
  `sched/sch_generic.c:pfifo_fast_enqueue()`
- Others: FIFO, Stochastic Fair Queuing, etc.
Priority Based Output Scheduling

- `pfifo_fast_enqueue()`
- Again, per-device basis
- Queue Discipline (Qdisc: `pkt_sched.c`)
  - Not exactly a priority queue
  - Uses three queues (bands)
    - 0 “interactive”
    - 1 “best effort”
    - 2 “bulk”
- Priority is based on IP Type of Service (ToS)
  - Normal IP packet \(\rightarrow\) 1 “best effort”
Queue Discipline: Qdisc

Mapping IP ToS to Queue

- **IP ToS: PPPDTRCX**
  - **PPP → Precedence**
    - Linux = ignore!
    - Cisco = Policy-Based Routing (PBR)
  - **D → Minimizes Delay**
  - **T → Maximizes Throughput**
  - **R → Maximizes Reliability**
  - **C → Minimizes Cost**
  - **X → Reserved**
Mapping IP ToS to Queue (cont.)

<table>
<thead>
<tr>
<th>IP ToS</th>
<th>Band</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>1</td>
</tr>
<tr>
<td>0x2</td>
<td>2</td>
</tr>
<tr>
<td>0x4</td>
<td>2</td>
</tr>
<tr>
<td>0x6</td>
<td>2</td>
</tr>
<tr>
<td>0x8</td>
<td>1</td>
</tr>
<tr>
<td>0xA</td>
<td>2</td>
</tr>
<tr>
<td>0xC</td>
<td>0</td>
</tr>
<tr>
<td>0xE</td>
<td>0</td>
</tr>
<tr>
<td>0x10</td>
<td>1</td>
</tr>
<tr>
<td>0x12</td>
<td>1</td>
</tr>
<tr>
<td>0x14</td>
<td>1</td>
</tr>
<tr>
<td>0x16</td>
<td>1</td>
</tr>
<tr>
<td>0x18</td>
<td>1</td>
</tr>
<tr>
<td>0xA</td>
<td>1</td>
</tr>
<tr>
<td>0x1C</td>
<td>1</td>
</tr>
<tr>
<td>0x1E</td>
<td>1</td>
</tr>
</tbody>
</table>

- `pfifo_fast_enqueue()` maps IP ToS to one of three queues
- IP ToS: PPPDTRCX
- Mapping array: prior2band
Queue Selection

`sch_generic.c`  Mapping array

```c
71 static const u8 prio2band[TC_PRIO_MAX+1] =
72 { 1, 2, 2, 2, 1, 2, 0, 0 , 1, 1, 1, 1, 1, 1, 1, 1 };
73
74 /* 3-band FIFO queue: old style, but should be a bit faster than
75   generic prio+fifo combination.
76 */
77
79 static int
80 pfifo.fast_enqueue(struct sk_buff *skb, struct Qdisc* qdisc)
81 {
82   struct sk_buff_head *list;
83
84   list = ((struct sk_buff_head*)qdisc->data) +
85   prio2band[skb->priority&TC_PRIO_MAX];
86```

Band “0” (first in Qdisc)  Change band
Queue Selection (cont.)

- Kernel 2.6.9.EL

```
Qdisc

<table>
<thead>
<tr>
<th>sk_buff_head</th>
<th>band 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sk_buff_head</td>
<td>band 1</td>
</tr>
<tr>
<td>sk_buff_head</td>
<td>band 2</td>
</tr>
</tbody>
</table>

list = ((struct sk_buff_head*)qdisc->data + prior2band[skb->priority&TC_PRIOR_MAX]
```

...
Sending Out a Packet

- `pfifo_fast_dequeue()`
  - Removes the oldest packet from the highest priority band
  - The packet that was just enqueued!
  - Passes it to the device driver
Lab 9

Scenarios, hints, etc.
Lab 9 Part 1&2

Scenario

- Destination
- Bottleneck link: 10Mbps
- Linux Router (Your HDD)
- Virtual 1
- Virtual 2
Lab 9 Part 2

- Default: no IP forwarding
  - Enable it! /proc/...
- Only one router
- Default route on “destination”
Lab 9 Part 2

Destination

Route???

Bottleneck link: 10Mbps

Linux Router (Your Linux)

ping echo

Virtual 1

Virtual 2
Lab 9 Part 3

- Scenario

Diagram:
- Destination
  - Linux Router (Your Linux)
    - TCP
    - UDP
      - 10Mbps
  - Virtual 1
  - Virtual 2
Problem with TCP v.s. UDP?
TCP is too “nice”
Proposed solution:
Modify kernel TCP → higher priority
Lab 9 Part 4

- **Goal:** compile the modified kernel
- Print out TCP/UDP when sending or forwarding a packet
- `/proc/sys/kernel/printk`
- Start up with the new kernel!
  - Press any key on boot → OS list
  - Select 2.6.9
Lab 9 Part 5

- Goal: change the kernel scheduling
- Idea: place TCP in the higher priority band

`pfifo_fast_enqueue()`
- Default → IP ToS
- Change it to TCP v.s. UDP (+others)
- Options: UDP++ or TCP--
- Do NOT change IP ToS!
Lab 9 Part 5 (cont.)

TCP

UDP

FIFO 0

FIFO 1

FIFO 2
Lab 9 Part 5 (cont.)

```c
71 static const u8 prio2band[TC_PRIO_MAX+1] =
72 { 1, 2, 2, 2, 1, 2, 0, 0 , 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 };    
74 /* 3-band FIFO queue: old style, but should be a bit faster than 
75     generic prio+fifo combination. */
77 static int
78 pfifo_fast_enqueue(struct sk_buff *skb, struct Qdisc* qdisc)
79 {
80     struct sk_buff_head *list;
81     list = ((struct sk_buff_head*)qdisc->data) +
82             prio2band[skb->priority&TC_PRIO_MAX];
83 ```
Lab 9 Part 5 (cont.)

- Remember: take `printk()` out!
  - boot into 2.6.9
  - enable forwarding

- What should happen?
- Different from Part 2?
Interesting Links

Linux Networking
- http://www.linuxfoundation.org/collaborate/workgroups/networking/kernelflow
- Understanding Linux Network Internals, Christian Benvenuti
- http://lartc.org/lartc.html

Queue Discipline

/proc file system