Linux Networking

Nima Honarmand

(Based on slides by Don Porter and Mike Ferdman)
4- to 7-Layer Diagram

OSI and TCP/IP Stacks

Used in Read World

OSI and TCP/IP Stacks (From Understanding Linux Network Internals)
Ethernet (IEEE 802.3)

- LAN (Local Area Network) connection

- Simple packet layout:
  - Header
    - Type (e.g., IPv4)
    - source MAC address
    - destination MAC address
    - length (up to 1500 bytes)
    - ...
  - Data block (payload)
  - Checksum

- Higher-level protocols “wrapped” inside payload

- “Unreliable” – no guarantee packet will be delivered
Shared vs. Switched

**Shared Ethernet:** 1 collision domain for multiple nodes. The possibility of collisions. Non-deterministic

**Switched Full Duplex Ethernet:** 1 collision domain per node. Use of switch. No possibility of collisions. Deterministic.

Source: http://www.industrialethernetu.com/courses/401_3.htm
Ethernet Details

• Originally designed for a shared wire (e.g., coax cable)

• Each device listens to all traffic
  – Hardware filters out traffic intended for other hosts
    • i.e., different destination MAC address
  – Can be put in “promiscuous” mode
    • Accept everything, even if destination MAC is not own

• If multiple devices talk at the same time
  – Hardware automatically retries after a random delay
Switched Networks

- Modern Ethernets are point-to-point and switched

- What is a hub vs. a switch?
  - Both are boxes that link multiple computers together
  - Hubs broadcast to all plugged-in computers
    - Let NICs figure out what to pass to host
      - Promiscuous mode sees everyone’s traffic
  - Switches track who is plugged in
    - Only send to expected recipient
      - Makes sniffing harder 😞
Internet Protocol (IP)

• 2 flavors: Version 4 and 6
  – Version 4 widely used in practice
  – Version 6 should be used in practice – but isn’t
    • Public IPv4 address space is practically exhausted (see arin.net)

• Provides a network-wide unique address (IP address)
  – Along with netmask
  – Netmask determines if IP is on local LAN or not

• If destination not on local LAN
  – Packet sent to LAN’s gateway
  – At each gateway, payload sent to next hop
Address Resolution Protocol (ARP)

- IPs are logical (set in OS with `ifconfig` or `ipconfig`)

- OS needs to know where (physically) to send packet
  - And switch needs to know which port to send it to

- Each NIC has a MAC (Media Access Control) address
  - “physical” address of the NIC

- OS needs to translate IP to MAC to send
  - Broadcast “who has 10.22.17.20” on the LAN
  - Whoever responds is the physical location
    - Machines can cheat (spoof) addresses by responding
  - ARP responses cached to avoid lookup for each packet
User Datagram Protocol (UDP)

- Applications on a host are assigned a port number
  - A simple integer
  - Multiplexes many applications on one device
  - Ports below 1k reserved for privileged applications

- Simple protocol for communication
  - Send packet, receive packet
  - No association between packets in underlying protocol
    - Application is responsible for dealing with...
      - Packet ordering
      - Lost packets
      - Corruption of content
      - Flow control
      - Congestion
Transmission Control Protocol (TCP)

• Same port abstraction (1-64k)
  – But different ports
  – i.e., TCP port 22 isn’t the same port as UDP port 22

• Higher-level protocol providing end-to-end reliability
  – Transparent to applications
  – Lots of features
    • packet acks, sequence numbers, automatic retry, etc.
  – Pretty complicated
Web Request Example

Figure 13-4. Headers compiled by layers: (a...d) on Host X as we travel down the stack; (e) on Router RT1

From *Understanding Linux Network Internals*
User-level Networking APIs

• Programmers rarely create Ethernet frames
  – Or IP or TCP packets

• Most applications use the **socket** abstraction
  – Stream of messages or bytes between two applications
  – Applications specify protocol (TCP or UDP), remote IP address and port number

• `bind() / listen() / accept()`: **waits for incoming connection (Server)**
• `connect()`: **connect to remote end (client)**
• `send() / recv()`: **send and receive data**
  – All headers are added/stripped by OS
Linux Implementation

• Sockets implemented in the kernel
  – So are TCP, UDP, and IP

• Benefits:
  – Application not involved in TCP ACKs, retransmit, etc.
    • If TCP is implemented in library, app wakes up for timers
  – Kernel trusted with correct delivery of packets

• A single system call:
  – sys_socketcall(call, args)
    • Has a sub-table of calls, like bind, connect, etc.
Linux Plumbing

• Each message is put in a `sk_buff structure`
  – Passed through a stack of protocol handlers
  – Handlers update bookkeeping, wrap headers, etc.

• At the bottom is the device itself (e.g., NIC driver)
  – Sends/receives packets on the wire
Efficient Packet Processing

- **Recv side:** Moving pointers is better than removing headers

- **Send side:** Prepending headers is more efficient than re-copy

(From *Understanding Linux Network Internals*)
Received Packet Processing

Application
- application reads from socket
- socket reads from queue

Transport
- TCP checks for errors
- packet goes in socket queue
- socket asks for payload

Internet
- IP checks for errors
- route to different host
- (See IP Forwarding)

Link
- scheduler runs "bottom half"
- net_bh pops packet queue
- net_bh matches protocol (IP)
- out to send queue
- Packet waits on queue

Source: http://www.cs.unh.edu/cnrg/people/gherrin/linux-net.html
Interrupt Handler

• “Top half” responsible to:
  – Allocate/get a buffer (sk_buff)
  – Copy received data into the buffer
  – Initialize a few fields
  – Call “bottom half” handler

• In reality:
  – Systems allocate ring of sk_buffs and give to NIC
  – Just “take” the buff from the ring
    • No need to allocate (was done before)
    • No need to copy data into it (DMA already did it)
Soft-IRQs

• A hardware IRQ is the hardware interrupt line
  – Use to trigger the “top half” handler from IDT

• Soft-IRQ is the big/complicated software handler
  – Or, “bottom half”

• Why separate top and bottom halves?
  – To minimize time in an interrupt handler with other interrupts disabled
  – Simplifies service routines (defer complicated operations to a more general processing context)
    • E.g., what if you need to wait for a lock?
  – Gives kernel more scheduling flexibility
Soft-IRQs

• How are these implemented in Linux?
  – Two canonical ways: Softirq and Tasklet
  – More general than just networking

• Kernel’s view: per-CPU work lists
  – Tuples of <function, data>

• At the right time, call function(data)
  – Right time: Return from exceptions/interrupts/syscalls
  – Each CPU also has a kernel thread ksoftirqd_CPU#
    • Processes pending requests
    • ksoftirqd is nice +19: Lowest priority – only called when nothing else to do
Softirqs

• Only one instance of softirq will run on a CPU at a time
  – Doesn’t need to be reentrant
    • If interrupted by HW interrupt, will not be called again
    • Guaranteed that invocation will be finished before start of next

• One instance can run on each CPU concurrently
  – Need to be thread-safe
    • Must use locks to avoid conflicting on data structures
Tasklets

• Especial form of softirq
  – For the faint of heart (and faint of locking prowess)

• Constrained to only run one at a time on any CPU
  – Useful for poorly synchronized device drivers
    • Those that assume a single CPU in the 90’s
  – Downside: All tasklets are serialized
    • Regardless of how many cores you have
    • Even if processing for different devices of the same type
      • e.g., multiple disks using the same driver
Back to Receive: Bottom Half

• For each pending `sk_buff`:
  – Pass a copy to any taps (sniffers)
  – Do any MAC-layer processing, like bridging
  – Pass a copy to the appropriate protocol handler (e.g., IP)
    • Recur on protocol handler until you get to a port number
      • Perform some handling transparently (filtering, ACK, retry)
    • If good, deliver to associated socket
    • If bad, drop
Socket Delivery

• Once bottom half moves payload into a socket:
  – Check to see if a task is blocked on input for this socket
    • If yes, wake it up

• Read/recv system calls copy data into application
Socket Sending

- Send/write system calls copy data into socket
  - Allocate `sk_buff` for data
  - Be sure to leave plenty of head and tail room!

- System call handles protocol in application’s timeslice
  - Receive handling not counted toward app

- Last protocol handler enqueues packet for transmit

- Interrupt usually signals completion
  - Interrupt handler just frees the `sk_buff`
Receive Livelock

• What happens when packets arrive at a very high frequency?
  – You spend all of your time handling interrupts!

• **Receive Livelock:** Condition when system never makes progress
  – Because spends all of its time starting to process new packets
  – Bottom halves never execute
    • Hard to prioritize other work over interrupts

• Better process one packet to completion than to run just the top half on a million
Receive Livelock in Practice

Fig. 2. Forwarding performance of unmodified kernel.

Source: Mogul & Ramakrishnan, ToCS, Aug 1997
Shedding Load

• If can’t process all incoming packets, must drop some

• If going to drop some packets, better do it early!
  – Stop taking packets off of the network card
    • NIC will drop packets once its buffers get full on its own
Polling Instead of Interrupts

• Under heavy load, disable NIC interrupts

• Use polling instead
  – Ask if there is more work once you’ve done the first batch

• Allows packet go through bottom half processing
  – And the application, and then get a response back out

• Ensures some progress
Why not Poll All the Time?

• If polling is so great, why bother with interrupts?

• Latency
  – If incoming traffic is rare, want high-priority
    • Latency-sensitive applications get their data ASAP
    • Example: annoying to wait at ssh prompt after hitting a key
General Insight on Polling

• If the expected input rate is low
  – Interrupts are better

• When expected input rate is above threshold
  – Polling is better

• Need way to dynamically switch between methods
Why Only Relevant to Networks?

• Why don’t disks have this problem?
  – Inherently rate limited

• If CPU is too busy processing previous disk requests
  – It can’t issue more

• External CPU can generate all sorts of network inputs
Linux NAPI (New API)

• Drivers provides `poll()` method for low-level receive
  – Passes packets received by the device to kernel

• Top half schedules `poll()` to do the receive as a softirq
  – Can disable the interrupt under heavy loads
    • And use a timer interrupt to schedule a poll
  – Bonus: Some NICs have a built-in timer
    • Can fire an interrupt periodically, only if something to say!

• Gives kernel control to throttle network input
  – Under heavy-load, device will overwrite some packets
    • Packets dropped in the device itself without involving the CPU