Computer Networks
ICS 651

- IP fragmentation
- Path MTU discovery
- Internet checksum
- Layering and Architecture
- Network Device Design
- Device Drivers
- Multicasting Algorithms and IP Multicasting

IP fragmentation

- If a packet of size s is to be sent on an interface with MTU m,

  and if s > m

- the packet must either be dropped, or be turned into a collection of smaller packets which carry the same information
  - in IPv6 we drop the packet
  - in IPv4, we drop the packet if the Don't Fragment (DF) bit is set, and otherwise we fragment it
- each of the smaller packets has its own IP header, which is based on the IP header of the original (too large) IP packet:
- source and destination addresses, hop limit (TTL), and most other fields are the same in the original and all the fragments
- the payload length is adjusted for each fragment
- and some fields are set so that the packet can be reassembled correctly
- each packet ID is the same as in the original datagram

IP fragmentation and reassembly information

- the fragment must contain information about where the payload belongs in the original (unfragmented) datagram
  - this information is the *fragment offset*
    - the fragment offset must be a multiple of 8, so the low-order 3 bits (which are always 0) are not sent – only the top 13 bits are sent
- the fragment should also tell us how big a datagram to expect, so we know when the reassembly is complete
  - when we get the last fragment, its *More Fragments* (*MF*) bit is 0 – every other fragment has MF = 1
- unfragmented packets have fragment offset = 0, MF = 0

IP reassembly

- when receiving a fragment (MF ≠ 0 or fragment offset ≠ 0)
  - check to see if the packet ID matches an existing reassembly context
  - if not, create a new reassembly context
  - the context includes space for the reassembled packet
  - this packet is initially *empty*
- if MF ≠ 0, we now know the final length
- copy the payload into the reassembled packet at the given fragment offset
- keep track of which parts of the packet have been filled
- once the entire packet has been filled, and we know the final length, do IP processing on the entire packet

IP reassembly – summary

- the fragment offset and MF are sufficient for reconstructing the entire packet
- they must be set correctly when fragmenting
- a fragment can be fragmented again if necessary, as long as the fragment offset and MF are set correctly
- the packet ID is an arbitrary number that helps the receiver distinguish packets from the same sender
  - no two packets with the same packet ID should ever be *in flight* at the same time

IP Path MTU Discovery

- send IPv6 packets, or IPv4 packets with DF (Don’t Fragment) set
- cannot send more than local network MTU
- if a router must drop a packet that exceeds the MTU of the outgoing interface, it can send a "destination unreachable/fragmentation needed" ICMP message, or an ICMPv6 “packet too big” message
- this ICMP message carries the MTU
- if there is no ICMP message, sender can do a binary search (on common MTU sizes) to find an MTU that works
- however, the path MTU can change!
- since ICMP message may be dropped, also needs other ways to detect dropped packets
- slow, time-consuming, error-prone...

Internet Checksum

- IP header
- ICMP, TCP, and UDP header, data, and "pseudo-header"
- pseudo-header are the IP level fields which, if corrupted, cause mis-delivery: source and destination IP addresses, protocol number, packet length
- if all bytes in packet add to n (without checksum), put -n in checksum field, so all received bytes added together give 0
- 16-bit, one's complement arithmetic checksum

16-bit 1's complement arithmetic

- add unsigned 16-bit quantities as always
- "carry-out" from 16-bit addition added back in to LSB
- "carry-out" can be accumulated in high-order part of 32-bit word, and added at end
- negation is complement, zero is 0xffff or 0x0
  - in numbers obtained by addition, zero is always 0xffff
  - if you add 1 to 0xffff, you get 0 plus a carry out bit, which when added to 0, gives 1 – this is the desired result
- example: 9ABC + 8888 = 2345

Layering

- a packet on the wire has a sequence of headers followed by the payload, possibly followed by a sequence of trailers
- headers (and trailers) are properly nested: each layer will encapsulate as its own payload the header, payload, and trailer of the layer (transparent exception for fragmentation)
- if a layer can have more than one layer above it, the header must contain one or more demultiplexing fields
- layers communicate by:
  - sending frames/packets/buffers
  - receiving frames/packets/buffers
  - establishing and clearing connections

Standard Layers

- Layer 7: application layer. For example, HTTP, HTTPS (= SSL/TLS + HTTP), DNS, SMTP, NTP, etc
- Layer 6: presentation layer. For example, MIME
- Layer 5: session layer. For example, persistent logins and cookies
- Layer 4: transport layer. For example, TCP, UDP, and SCTP
- Layer 3: network layer. For example, IP
- Layer 2: data link layer. For example, Ethernet (802.3), WiFi (802.11), Zigbee (802.15.4), Bluetooth (802.15.1)
- Layer 1: physical layer. For example, CAT-5 twisted pair, 2.4GHz spread-spectrum radio, serial cable

Upcalls and Queues

- in sending, the data can be transferred from one layer to another as parameter of a call
- in receiving, the data can be transferred from one layer to another as:
  - parameter of an upcall
  - return value of a downcall
- a receiving downcall requires synchronization to handle:
  - the downcall being ready before the data (blocking, context switch)
  - the data being ready before the downcall (queueing)
- an upcall may also require synchronization
  - e.g. may have to acquire a lock before accessing a shared data structure

Receive: Upcalls and Downcalls

- upcalls are simpler to implement and generally more efficient (less synchronization for each packet)
- upcalls are harder to fit into an overall program (multiple threads of control, requiring possibly pervasive synchronization)
- Linux (version 2.0) used 2 queues:
  - the interrupt hander is an upcall. It does device handling (described later), and puts the received packet in a device-specific queue
  - the network code reads the queue, and when packets are received, upcalls through the entire network stack, then puts the packet(s) in a socket-specific queue
- the system call reads the queue, may block

Network Interface Device

- Networking Hardware, NIC
- UART/USART, Universal [Synchronous /] Asynchronous Receiver Transmitter
- UART is a single-chip device that accepts or returns data depending on the address on the computer's bus
- the hardware device has bits which control its operation (e.g. a bit to decide whether to interrupt the host). These are grouped into one or more **control registers**
- the hardware device has bits which record events (e.g. a bit to record whether a character was received, but not given to the host). These are grouped into one or more **status registers**

NIC for packet networks

- OS configures device at initialization time
  - often by building a descriptor in memory, and writing the address of this descriptor to a control register
- device can directly access (read and write) the computer's memory -- Direct Memory Access, DMA
- when a packet is received, the device copies the contents to a buffer pre-allocated by the OS, then interrupts
- interrupt handler processes the packet, checks device status, allocates new buffers to the device
- to send, device driver writes to a control register the address of a linked list of buffers to send
  - one buffer per packet, or sometimes
  - multiple buffers for a single packet: one buffer for the headers, one buffer for user data (**scattered** representation)
- device interrupts once the packets have been sent

Device Drivers

- The device-specific part of an Operating System is called a **device driver**
- device drivers are often loaded on demand as the OS discovers new hardware
  - e.g. Linux modules
- a device driver for a network device usually includes:
  - initialization code
  - an interrupt handler: this is the “bottom half” of the driver, called by the hardware
    - the system interrupt handler calls the device-specific interrupt handler
  - code to send data to the network: this is the “top half” of the driver, (ultimately) called by user programs

Multicasting: Ideas and Reality

- Audio and video conferencing: (usually) one sender at a time, potentially many recipients
- Reality: the sender has a connection to/from each participant, sends a customized data stream to each
  - from a central server with high bandwidth
  - the originator of the data sends to this server
- Idea: intermediate routers can duplicate data streams “for free” (just by adding the same packet to multiple queues). Each sender would then be able to send a single stream of data, and reach all the recipients
  - decentralized, each participant needs the same bandwidth as every other participant
  - the automatic distribution of a single packet to multiple destinations is what network people mean by multicasting

Multicasting on a broadcast-based Local Area Network (LAN)

- multicasting requires that the hardware device of the intended recipients process the packet
  - all other systems on the network discard the packet, either in the device hardware (most efficient) or in software (less efficient)
- modern LAN hardware is designed to accept packets for its own unchangeable MAC address, for the broadcast address `ff:ff:ff:ff:ff:ff`, and also for a finite number of addresses configured at runtime: this makes LAN multicast very efficient as long as senders know which special MAC address to use
- IPv6 multicast packets sent to an IPv6 address ending in the four bytes aabb:ccdd are sent to the MAC address `33:33:aa:bb:cc:dd`
  - RFC 2464
  - so for example the routing packets in Project 1 sent to ff02::1, if they were sent on a LAN, would be sent to the MAC address `33:33:00:00:00:01`

Ideal Multicast across Routers

- Routers must know where to forward multicast packets
  - leaf-initiated join: request packet from the host takes the reverse route towards the sender
    - when the request packet reaches a router that is already carrying the multicast stream, the router starts forwarding the stream over the interface on which it received the request
    - sound familiar?
  - sender-managed multicast: sender must configure routers to forward multicast packets to all the correct destinations
  - a rendez-vous point (RP) is a central server that can act as the “sender” here
- Either way, only works if there are routers supporting multicast
  - easier to set up within an autonomous system
- Protocols that support IP multicast include:
  - Protocol-Independent Multicast (PIM), which has several variants, and
  - Multicast Source Discovery Protocol (MDSP), which can be used across domains