# Computer Networks
# ICS 651

- Network Device Design
- Device Drivers
- Multicasting Algorithms and IP Multicasting
- Overview of TCP
- TCP connection management
- TCP 3-way handshake
- TCP close
- TCP reset

# Network Interface Device

- Networking Hardware, NIC (Network Interface Card)
- 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, merging data streams from multiple actual senders
- 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

# The need for TCP

- the task of IP is to transfer packets of data end-to-end
- packets may be lost, corrupted, reordered (even mis-delivered)
- applications could use IP directly, but:
  - need a way to demultiplex data at the receiver, so multiple applications can run simultaneously
  - most applications require reliable data delivery
  - applications may need to send packets larger than the largest possible IP datagram
  - a fast sender could overwhelm a slow receiver, causing loss of data

# Overview of TCP

- uses IP to gain (unreliable) end-to-end connectivity
- uses port numbers for demultiplexing to multiple applications
- uses checksum to discard corrupted data
- uses sequence numbers to detect lost and reordered packets
- uses acknowledgments and retransmission for reliable delivery
- uses windows to avoid overwhelming the receiver
- provides streams to overcome any packet size limitations

# TCP connections

- sequence numbers, windows, etc. must be remembered and applied to incoming packets
- remembering these numbers is a form of state
- since TCP has state, designers decided to have the peers explicitly manage this state (called a connection)
- the peers agree on when to establish (**open**) a connection, when to tear it down (**close**), and when the connection must be thrown away (**reset**)
- the state on each system reflects an understanding about the state on the peer

# TCP connection establishment

- when I receive a request to establish a connection, I must check:
  - that I don't already have this socket: one or more of the port numbers or IP numbers must differ from existing connections
  - that an application on my end desires to be connected
  - that I have sufficient resources to handle this connection
- the purpose of the connection establishment phase is to set up consistent connection state on the two peers

# TCP 3-way handshake

- from state CLOSED:
  - send SYN, enter state SYN SENT
  - receive SYN and ACK, send ACK, enter state ESTAB, or
  - receive SYN, send ACK, enter state SYN RCVD, then proceed as below
- from state LISTEN:
  - receive SYN, send SYN and ACK, enter state SYN RCVD
  - receive ACK, enter state ESTAB
- retransmissions in case any of these are dropped
- see page 23 of RFC 793

# TCP close

- from state ESTAB:
  - receive FIN, send ACK, enter state CLOSE WAIT
  - application closes connection, send FIN, enter state LAST ACK
  - receive ACK, enter state CLOSED
- from state ESTAB:
  - application closes connection, send FIN, enter state FINWAIT-1
  - receive FIN, send ACK, enter state CLOSING
  - receive ACK, enter state TIME WAIT

# TCP close, part 2

- from state FINWAIT-1, if we get an ACK:
  - receive ACK, enter state FINWAIT-2
  - receive FIN, send ACK, enter state TIME WAIT
- from state TIME WAIT, enter state CLOSED after 4 minutes (2 Maximum Segment Lifetimes)
- last ack issue

# TCP reset

- what should I do if I get a TCP segment for a connection that I have no record of? -- tell the sender to reset its connection
- If I am opening the connection and the segment I receive has an acknowledgement number I've never used, it might be an old segment. Again, reset the connection
- If the application program terminates, no sense in waiting for all the data to be delivered using the normal close

# ATM connection establishment

- Asynchronous Transfer Mode, Q.293b
- typical of public carrier protocols
- a connection request may elicit a response or an acknowledgement
- eventually we expect to get a response, which we acknowledge
- less focus on efficiency and light weight, more focus on informing the "application" of the current status
- signaling and connections are always point-to-point, not end-to-end (in other words, the inter/network layer is connection-oriented)
  - this makes it easier to allocate resources to connections