Problem: allocating resources

- Congestion control
- Quality of service

Congestion Control and Resource Allocation

Hongwei Zhang
http://www.cs.wayne.edu/~hzhang

The hand that hath made you fair hath made you good.
--- William Shakespeare

Acknowledgement: this lecture is partially based on the slides of Dr. Larry Peterson
Issues in resource allocation

- Two sides of the same coin
  - control congestion if (and when) it occurs: reactive
  - pre-allocate resources so as to avoid congestion: proactive

- Two places of implementation
  - *hosts* at the edges of the network (transport protocol)
  - *routers* inside the network (queuing discipline)
Network model

- Packet switched network
  - Congestion (and thus packet drop) may occur in the network; e.g.,
Network model (contd.)

- Connectionless flows
  - sequence of packets sent between source/destination pair
  - no well-defined circuit, but a sequence of packets can be regarded a \textit{flow}, and we can maintain \textit{soft state} at routers for individual flows

Multiple flows passing through a set of routers
Network model (contd.)

- Underlying service models
  - best-effort (assume for now)
  - multiple *qualities of service* (later)
Taxonomy of resource allocation methods

- router-centric vs. host-centric
- reservation-based vs. feedback-based
- rate-based vs. window-based

In practice,
- *Best-effort* service model usually implies feedback-based method, and thus host-centric and usually window-based
- QoS-based service model usually implies reservation-based method, and thus router-centric and usually rate-based
Evaluation criteria

- Fairness among different flows
- Power (ratio of throughput to delay)
Outline

- Queuing Discipline

- Congestion control
  - Reacting to Congestion
  - Avoiding Congestion
Outline

- Queuing Discipline

- Congestion control
  - Reacting to Congestion
  - Avoiding Congestion
Queuing Discipline

- **First-In-First-Out (FIFO)**
  - does not discriminate between traffic sources

- **Fair Queuing (FQ)**
  - explicitly segregates traffic based on flows (source-destination pair)
  - ensures no flow captures more than its share of capacity

![Diagram showing round-robin service with four flows: Flow 1, Flow 2, Flow 3, and Flow 4 with a round-robin service cycle.]
Problem with basic round-robin service

- Packet length of different flows is different

  =>

  Unfair sharing of bandwidth
FQ Algorithm

- Suppose clock ticks each time a bit is transmitted.
- Let $P_i$ denote the length of packet $i$.
- Let $S_i$ denote the time when start to transmit packet $i$.
- Let $F_i$ denote the time when finish transmitting packet $i$.
  
  \[ F_i = S_i + P_i \]

- When does router start transmitting packet $i$ (single flow):
  - if packet arrived before router finished packet $i-1$ from this flow, then immediately after last bit of $i-1$ ($F_{i-1}$).
  - if no current packets for this flow, then start transmitting when arrives (call this $A_i$).

- Thus: $F_i = \text{MAX}(F_{i-1}, A_i) + P_i$.
FQ Algorithm (contd.)

- For multiple flows
  - calculate $F_i$ for each packet that arrives on each flow
  - treat all $F_i$'s as timestamps
  - next packet to transmit is one with lowest timestamp

- Example

  ![Image](image_url)

  Output

  - Shorter packets are sent first
  - Sending of longer packet, already in progress, is completed first

- Not perfect: can’t preempt current packet
Weighted fair queuing (WFQ)

- Assign a weight for each flow (queue)

- Weight logically specifies “how many bits to transmit each time the router services a flow/queue”

- FQ: each flow has a weight of 1
Outline

- Queuing Discipline
- Congestion control
  - Reacting to Congestion
  - Avoiding Congestion
TCP Congestion Control

- Introduced by Van Jacobson through his Ph.D. dissertation work in late 1980s

- Basic ideas
  - Assumes best-effort network (FIFO or FQ routers) without reservation
  - Each source determines network capacity for itself
    - Uses implicit feedback (i.e., does not require assistance from routers)
    - ACKs pace transmission (self-clocking)
      - a received ACK enables transmission of at least another packet

- Challenge
  - determining the available capacity in the first place
  - adjusting to changes in the available capacity
Additive Increase/Multiplicative Decrease (AIMD)

- Objective: estimate and adapt to (varying) network capacity

- New state variable per connection: CongestionWindow
  - limits how much data source has in transit
    \[
    \text{MaxWin} = \text{MIN} (\text{CongestionWindow}, \text{AdvertisedWindow})
    \]
    \[
    \text{EffWin} = \text{MaxWin} - (\text{LastByteSent} - \text{LastByteAcked})
    \]

- Idea:
  - increase CongestionWindow when congestion goes down
  - decrease CongestionWindow when congestion goes up
Question: how does the source determine whether or not the network is congested?

Answer: when a packet is lost

- packets are seldom lost due to transmission error
  - Not true in wireless networks 😞
- lost packet implies congestion

A typical indication of packet loss: timeout
AIMD (contd.)

- Algorithm
  - *linear increase*: increment `CongestionWindow` by one packet each time the source successfully sends a `CongestionWindow`’s worth of packets (~1 RTT time)
  - divide `CongestionWindow` by two whenever a packet is lost (*multiplicative decrease*)

- In practice: increment a little for each ACK
  
  \[
  \text{Increment} = \text{MSS} \times \left(\text{MSS}/\text{CongestionWindow}\right)
  \]
  
  \[
  \text{CongestionWindow} += \text{Increment}
  \]

  where:
  
  - MSS is the max. segment size;
  - we assume that each ACK acks the receipt of MSS bytes
AIMD (contd.)

- Trace: sawtooth behavior
Slow Start

- **Observation:**
  - “Additive increase” works well when the source is operating close to the network capacity, but would be too slow when starting from scratch.

- **Objective:**
  - To determine the available capacity in the first place.

- **Idea:**
  - begin with `CongestionWindow = 1 packet`
  - double `CongestionWindow` each time the source successfully sends a `CongestionWindow`’s worth of packets (~1 RTT time)
    - increment by 1 packet for each ACK
  - Change to additive-increase after `CongestionWindow` exceeds certain slow-start threshold (which is initialized to 65535 bytes initially, and then the `CongestionWindow` Value that results from multiplicative-decrease).
Slow Start (contd.)

- An example
Slow Start (contd.)

- Why “slow”? exponential growth, but slower than “all at once”

- Used ...
  - when first starting a connection
  - when connection goes “silent” while waiting for timeout

- A real-world trace

- Problem: can lose up to half a CongestionWindow’s worth of data in slow-start phase (at the critical transition point in traffic load vs. capacity)
Fast Retransmit

- Problem: coarse-grain TCP timeouts lead to idle periods

- Fast retransmit:
  - A duplicate ACK is sent whenever a packet is received out-of-order
  - use “multiple duplicate ACKs (e.g., 3)” to trigger retransmission
    - Q: why not retransmit upon receiving a single duplicate ACK?
Example of fast retransmit

- Less idle time
- Fast retransmit does not eliminate coarse-grain timeout (why?)
Fast recovery

- Skip the slow start phase between “fast retransmit” and “additive increase begins”
- Go directly to half the last successful CongestionWindow

- Thus, slow start is only used ...
  - At the beginning of a connection
  - When a coarse-grain timeout occurs
Outline

- Queuing Discipline

- Congestion control
  - Reacting to Congestion
  - Avoiding Congestion
Congestion Avoidance

- TCP’s strategy
  - control congestion once it happens
  - repeatedly increase load in an effort to find the point at which congestion occurs, and then back off

- Alternative strategy
  - predict when congestion is about to happen
  - reduce rate before packets start being discarded
  - call this congestion *avoidance*, instead of congestion *control*
    - Appealing, but not yet widely adopted
Congestion avoidance (contd.)

- Two possibilities
  - router-centric: DECbit and RED (random early detection)
  - Host(source)-centric: TCP Vegas
DECbit

- Designed before TCP/IP was “standardized”
- Developed for the Digital Network Architecture (DNA), a connectionless network with connection-oriented transport protocol
Add binary congestion bit to each packet header

Router
- monitors average queue length over last busy+idle cycle

set congestion bit if average queue length > 1
  - “1” is chosen as a trade-off between significant queuing (hence higher throughput) and increased idle time (hence lower delay), i.e., to optimize the power function
End Hosts in DECbit

- Destination echoes bit back to source
- Source records how many packets resulted in set-bit
- If less than 50% of last window’s worth had bit set
  - increase congestionWindow by 1 packet (i.e., additive increase)
- If 50% or more of last window’s worth had bit set
  - decrease congestionWindow by 0.875 times (i.e., multiplicative decrease)
- Note: “50%” is chosen to maximize the power, and AIMD is to guarantee stability
Random Early Detection (RED)

- Developed for TCP/IP (by Sally Floyd & Van Jacobson)

- Notification is implicit (instead of explicit as in DECbit)
  - just drop the packet
    - TCP will timeout, or retransmit after receiving duplicate ACKs
  - could make explicit by marking the packet
    - as in Explicit Congestion Notification (ECN)

- Early random drop
  - rather than wait for queue to become full, drop each arriving packet with some *drop probability* whenever the queue length exceeds some *drop level*
RED Details

- Compute average queue length

\[
\text{AvgLen} = (1 - \text{Weight}) \times \text{AvgLen} + \text{Weight} \times \text{SampleLen}
\]

Where:
- \(0 < \text{Weight} < 1\) (usually 0.002)
- \(\text{SampleLen}\) is queue length each time a packet arrives
Two queue length thresholds

if \( \text{AvgLen} \leq \text{MinThreshold} \) then

enqueue the packet

if \( \text{MinThreshold} < \text{AvgLen} < \text{MaxThreshold} \) then

calculate probability \( P \)

drop arriving packet with probability \( P \)

if \( \text{ManThreshold} \leq \text{AvgLen} \) then

drop arriving packet
RED Details (contd.)

Computing probability $P$

\[
\text{TempP} = \text{MaxP} \times \\
(Avg\text{Len} - \text{MinThreshold})/(\text{MaxThreshold} - \text{MinThreshold}) \\
\text{P} = \text{TempP}/(1 - \text{count} \times \text{TempP})
\]

where:

\text{count} = \# \text{ of newly arriving packets queued (but not dropped) while AvgLen has been between the two thresholds}

Drop Probability Curve

![Graph showing the drop probability curve]
RED Details (contd.) --- Tuning RED

- Random dropping => probability of dropping a particular flow’s packet(s) is roughly proportional to the share of the bandwidth that flow is currently getting.

- \textbf{MaxP} is typically set to 0.02, meaning that when the average queue size is halfway between the two thresholds, the gateway drops roughly one out of 50 packets.

- If traffic is bursty, then \texttt{MinThreshold} should be sufficiently large to allow link utilization to be maintained at an acceptably high level.

- Difference between two thresholds should be greater than the typical increase in the calculated average queue length in one RTT (which is the min. source response time after detection).
  - setting \texttt{MaxThreshold} to twice \texttt{MinThreshold} is reasonable for traffic on today’s Internet.
TCP Vegas (host/source-centric)

- Idea: source watches for some sign that router’s queue is building up and congestion will happen too; e.g.,
  - RTT grows
  - sending rate flattens
Algorithm

- Let $\text{BaseRTT}$ be the minimum of all measured RTTs
  - commonly the RTT of the first packet
- If not overflowing the connection, then
  \[
  \text{ExpectRate} = \frac{\text{CongestionWindow}}{\text{BaseRTT}}
  \]
- Source calculates sending rate ($\text{ActualRate}$) once per RTT
- Source compares $\text{ActualRate}$ with $\text{ExpectRate}$
  \[
  \text{Diff} = \text{ExpectedRate} - \text{ActualRate}
  \]
  if $\text{Diff} < \alpha$ //under-utilized
    increase $\text{CongestionWindow}$ linearly
  else if $\text{Diff} > \beta$ //over-utilized
    decrease $\text{CongestionWindow}$ linearly
  else
    leave $\text{CongestionWindow}$ unchanged
Algorithm (contd.)

- **Parameters**
  - $\alpha = 1$ packet
  - $\beta = 3$ packets

![Graph showing sending rate, expected rate, and actual rate over time. The graph includes a line graph with axes labeled as Time (seconds) and Y-axis values ranging from 0 to 70 for the top graph, and from 40 to 240 for the bottom graph. The top graph shows a lower sub-graph indicating congestion window size. The bottom graph also shows two sub-graphs, one for expected rate and another for actual rate.]