Administrivia

- **Canceling my office hours this week**
- **Phil and I both have to be out of town**
- **Sachin Katti will give guest lecture on Coding Thursday**

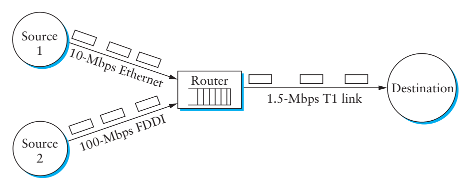
Overview

- **How routers queue affects how TCP and other protocols behave**
- **Two router questions: drop policy, scheduling policy**
- **Reducing congestion through content distribution**
  - Clients can cache
  - Services can use a CDN

Congestion Control Revisited

- **Congestion is when the input rate $\gg$ output rate**
  - In TCP, flow control window ensures sender does not exceed rate at which receiver consumes data
  - What if senders exceed a router's maximum output rate?
- **What should routers do? Make sender slow down**
- **TCP sending rate =** $\mathrm{window\text{-}size}/\mathrm{RTT}$**, so 2 options:**
  1. Increase RTT – buffer more packets $\Longrightarrow$ more queuing delay
  2. Reduce window size – happens if router drops packets
- **Recall TCP reacts to packet loss by shrinking congestion window**
  - Triple duplicate ack: halve window, enter CA state
  - Timeout: set window to 1, enter SS state

Congestion at Router

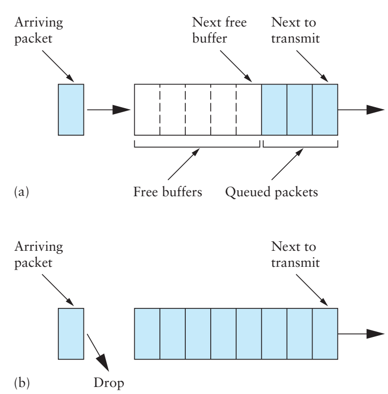


- **Router goals**
  - Prioritize who gets limited resources
  - Somehow interact well with TCP

Router design issues

- ***Scheduling discipline***
  - Which of multiple packets should you send next?
  - May want to achieve some notion of fairness
  - May want some packets to have priority
- ***Drop policy***
  - When should you discard a packet?
  - Which packet to discard?
  - Some packets more important (perhaps BGP)
  - Some packets useless w/o others (IP fragments)
- **Need to balance throughput & delay**
  - Could minimize/eliminate drops with enormous buffers
  - But queuing delay highly frowned upon (interactive apps)

Example: FIFO tail drop



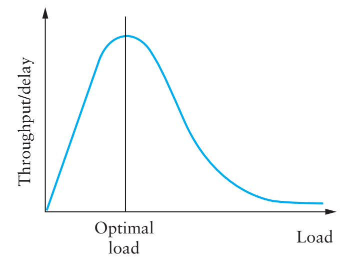
- **Differentiates packets only by when they arrive**
  - Packet dropped if queue full when it arrives

Tail drop issues

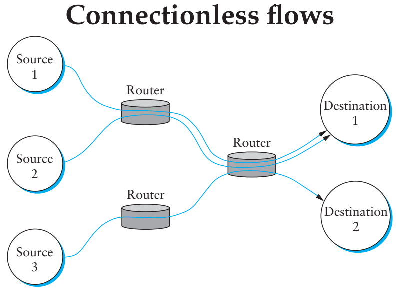
- **When stable, queue will always be nearly full**
  - Guarantees high latency for all traffic
- **Possibly unfair for flows with small windows**
  - E.g., small flow (< 4 packages) may be stuck in backoff, while larger flows can use fast retransmit to recover
- **Window synchronization**
  - Consider many flows in a stable configuration
  - New flow comes in, causes a bunch of packet losses
  - Existing flows all cut their windows together (underutilizing link)
  - Flows all grow their windows together until link again overloaded and many packets lost. Repeat…

What to optimize for?

- ***Fairness*** **(in two slides)**
- ***High throughput*** **– queue should never be empty**
- ***Low delay*** **– so want short queues**
- **Crude combination: *power* = Throughput/Delay**
  - Want to convince hosts to offer optimal load

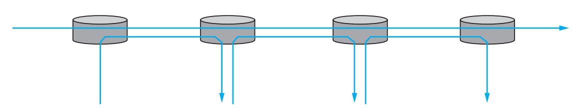


Connectionless flows



- **Even in Internet, routers can have a notion of flows**
  - E.g., base on IP addresses & TCP ports (or hash of those)
  - *Soft state*—doesn't have to be correct
  - But if often correct, can use to form router policies

Fairness



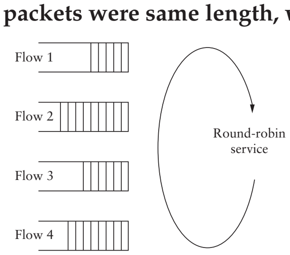
- **What is fair in this situation?**
  - Each flow gets $1/2$ link b/w? Long flow gets less?
- **Usually fair means equal**
  - For flow bandwidths $(x_1, \ldots, x_n)$, *fairness index*:

$$f(x_1, \ldots, x_n) = \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n \sum_{i=1}^{n} x_i^2}$$

  - If all x_is are equal, fairness is one
  - Weighted fairness is a simple extension
- **So what policy should routers follow?**

Scheduling Policy: Fair Queuing (FQ)

- **Explicitly segregates traffic based on flows**
- **Ensures no flow consumes more than its share**
  - Variation: weighted fair queuing (WFQ)
- **Note: if all packets were same length, would be easy**



Fair Queueing Basics

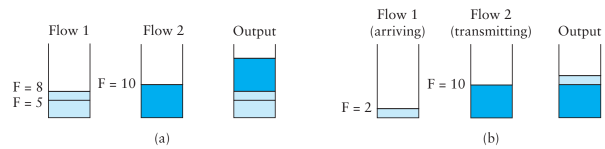
- **Keep track of how much time each flow has used link**
- **Compute how long a flow will have used link if it transmits next packet**
- **Send packet from flow which will have lowest use if it transmits**
  - Why not flow with smallest use so far?
  - Because next packet may be huge (examples coming)

FQ Algorithm

- **Suppose clock ticks each time a bit is transmitted**
- **P_i: length of packet i**
- **S_i: time when packet i started transmission**
- **F_i: time when packet i finished transmission**
- $F_i = S_i + P_i$
- **When does router start transmitting packet i?**
  - If 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:** $F_i = \max(F_{i-1}, A_i) + P_i$

FQ Algorithm (cont)

- **For multiple flows**
  - Calculate F_i for each packet that arrives on each flow
  - Treat all F_is as timestamps
  - Next packet to transmit is one with lowest timestamp
- **Not perfect: can't preempt current packet**
- **Example:**



FQ Algorithm (cont)

- **One complication: inactive flows are penalized ($A_i > F_{i-1}$)**
- **Over what interval do you consider fairness?**
  - Standard algorithm considers no history
  - Each flow gets fair share while packets queued
- **Solution:** $B_i = P_i + \max(F_{i-1}, A_i - \delta)$
- δ **= interval of history to consider**

Fair Queueing Importance

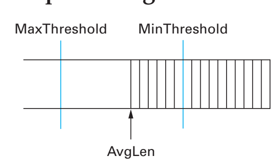
- **"Our packet-by-packet transmission algorithm is simply defined by the rule that, whenever a packet finishes transmission, the next packet is the one with the smallest F_i^α."**
- **But, fair queueing not used in core routers: finding min F in hundreds of thousands of flows is expensive. Can be used on edge routers and low speed links.**

Drop Policy: Random Early Detection (RED)

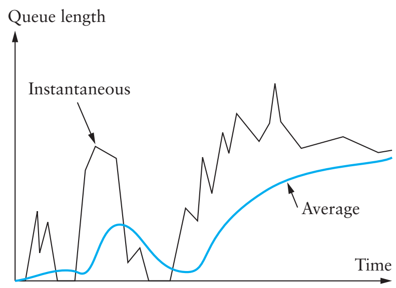
- **Notification of congestion is implicit in Internet**
  - Just drop the packet (TCP will timeout)
  - Could make explicit by marking the packet (ECN extension to IP allows routers to mark packets)
- **Early random drop**
  - Don't wait for full queue to drop packet
  - Instead, drop packets with some *drop probability* whenever the queue length exceeds some *drop level*
  - Prevents global window synchronization: many TCP flows speed up, all have packets dropped, all slow down, etc.

RED Details

- **Compute average queue length**
  - $\text{AvgLen} = (1 - \text{Weight}) \cdot \text{AvgLen} + \text{Weight} \cdot \text{SampleLen}$
  - $0 < \text{Weight} < 1$ (usually 0.002)
- **SampleLen is queue length each time a packet arrives**



AvgLen



- **Smooths out AvgLen over time**
  - Don't want to react to instantaneous fluctuations

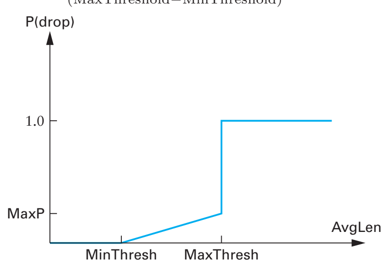
RED Details (cont)

- **Two queue length thresholds:**

```
if AvgLen <= MinThreshold then
    enqueue the packet
if MinThreshold < AvgLen < MaxThreshold then
    calculate probability P
    drop arriving packet with probability P
if MaxThreshold <= AvgLen then
    drop arriving packet
```

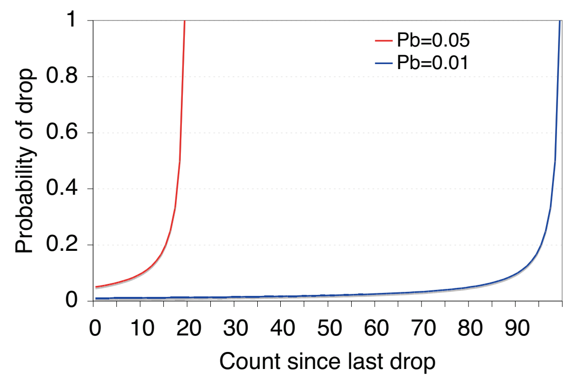
RED Details (cont)

- **Computing probability P**
  - $P_b = \text{MaxP} \cdot \frac{(\text{AvgLen}-\text{MinThreshold})}{(\text{MaxThreshold}-\text{MinThreshold})}$



- **Actual drop probability based on time since last drop**
  - $\text{count} =$ # pkts since drop or $\text{MinThresh} < \text{Avglen} < \text{MaxThresh}$
  - $P = P_b/(1 - \text{count} \cdot P_b)$
  - Space out drops, separate when to drop from which to drop

What P looks like



Tuning RED

- Probability of dropping a particular flow's packet(s) is roughly proportional to the share of the bandwidth that flow is currently getting
- **MaxP** is typically set to 0.02
- If traffic is bursty, then **MinThreshold** should be sufficiently large to allow link utilization to be maintained at an acceptably high level
- Difference between two thresholds should be larger than the typical increase in the calculated average queue length in one RTT; setting **MaxThreshold** to twice **MinThreshold** is reasonable for traffic on today's Internet

Queueing Today

- **Cisco IOS**
  - Scheduling: FIFO, FQ, WFQ, Custom queueing (patterns)
  - Drop policy: Tail drop, weighted random early detection

2-minute stretch

Content distribution

- **How can end nodes reduce load on bottleneck links?**
  - Congestion makes net slower – nobody wants this
- **Client side**
  - Many people from Stanford might access same web page
  - Redundant downloads a bad use of Stanford's net connection
  - Save resources by caching a copy locally
- **Server side**
  - Not all clients use caches
  - Can't upload unlimited copies of same data from same server
  - Push data out to content distribution network

Caching

- **Many network apps. involve transferring data**
- **Goal of caching: Avoid transferring data**
  - Store copies of remotely fetched data in *caches*
  - Avoid re-receiving data you already have
- **Caching concerns keeping copies of *data***

Examples

- **Web browser caches recently accessed objects**
  - E.g., allows "back" button to operate more efficiently
- **Web proxies cache recently accessed URLs**
  - Save bandwidth/time when multiple people locally access same remote URL
- **DNS resolvers cache resource records**
- **Network file systems cache read/written data**
- **PDA caches calendar stored in Desktop machine**

Cache consistency

- **Problem: What happens when objects change?**
- **Is cached copy of data is up to date?**
- ***Stale* data can cause problems**
  - E.g., don't see edits over a network file system
  - Get wrong address for DNS hostname
  - Shopping cart doesn't contain new items on web store
- **Can have various degrees of consistency**

One approach: TTLs

- **Eventual consistency**
- **Source controls how long data can be cached**
  - Can adjust trade-off: Performance vs. Consistency
- **Example: TTLs in DNS records**
  - When looking up `vine.best.stanford.edu`
  - CNAME record for `vine.best.stanford.edu` has very short TTL—value frequently updated to reflect load averages & availability
  - NS records for `best.stanford.edu` has long TTL (can't change quickly, and `stanford.edu` name servers want low load)
- **Example: HTTP reply can include `Expires:` field**

Polling

- **Check with server before using a cached copy**
  - Check requires far less bandwidth than downloading object
- **How to know if cache is up to date?**
  - Objects can include version numbers
  - Or compare time-last-modified of server & cached copies
- **Example: HTTP `If-Modified-Since:` request**
- **Sun network file system (NFS)**
  - Caches file data and attributes
  - To validate data, fetch attributes & compare to cached

Callbacks

- **Polling may cause scalability bottleneck**
  - Server must respond to many unnecessary poll requests
- **Example: AFS file system stores software packages**
  - Many workstations at university access software on AFS
  - Large, on-disk client caches store copies of software
  - Binary files rarely change
  - Early versions of AFS overloaded server with polling
- **Solution: Server tracks which clients cache which files**
  - Sends *callback* message to each client when data changes

Leases

- ***Leases* – promise of callback w. expiration time**
  - E.g., Download cached copy of file
  - Server says, "For 2 minutes, I'll let you know if file changes"
  - Or, "You can write file for 2 minutes, I'll tell you if someone reads"
  - Client can renew lease as necessary
- **What happens if client crashes or network down?**
  - Server might need to invalidate client's cache for update
  - Or might need to tell client to flush dirty file for read
  - Worst case scenario – only need to wait 2 minutes to repair
- **What happens if server crashes?**
  - No need to write leases to disk, if rebooting takes 2 minutes
- **Used by Google's internal naming/lock service (Chubby)**
- **Gray, Cheriton won test of time award for leases work done here at Stanford**

Content Distribution Network (CDN)

- **Network of computers that replicate content across the Internet**
- **Bringing content closer to requests can improve performance**
- **All users communicate with Redmond to download Microsoft SP**
  - Bottleneck: pipes to Redmond
- **Microsoft pushes SP to many hosts around the country**
  - Uses only local (not shared) capacity
- ***Actively* pushes data into the network**

Why CDNs succeed more (compared to web caches)

- **Incentives**
- **Content provider (e.g., Microsoft) uses/deploys CDN: wants to improve performance and reduce costs**
- **End user (e.g., network administrator) uses/deploys cache: wants to reduce external traffic**

Akamai

- **Challenge: static host name needs to point to different servers based on location**
- **Akamai servers cache content (images, videos, etc.)**
- **Uses DNS to direct clients to "close" servers**
- **Specifically, points clients to close NS servers**
- **Different NS servers provide different host lookups**

Caches and load balancing

- **Let's say you are Akamai**
  - Clusters of server machines running web caches
  - Caching data from many customers
  - Proxy fetches data from customer's *origin server* first time it gets request for a URL
- **Chose cluster based on client network location**
- **How to choose server within a cluster?**
- **Don't want to chose based on client...low hit rate**
  - N servers in cluster means N cache misses per URL
- **Also don't assume proxy servers 100% reliable**

Straw man: Modulo hashing

- **Say you have N proxy servers**
- **Map requests to proxies as follows:**
  - Number servers from 1 to N
  - For URL `http://www.server.com/web_page.html`, compute $h \leftarrow \text{HASH}(\text{"}\texttt{www.server.com}\text{"})$
  - Redirect clients to proxy # $p = h \bmod N$
- **Keep track of load on each proxy**
  - If load on proxy # p is too high, with some probability try again with different hash function
- **Problem: Most caches will be useless if you add/remove proxies, change value of N**

Consistent hashing [Karger]

- **Use circular ID space based on circle**
  - Consider numbers from 0 to $2^{160} - 1$ to be points on a circle
- **Use circle to map URLs to proxies:**
  - Map each proxy to several randomly-chosen points
  - Map each URL to a point on circle (hash to 160-bit value)
  - To map URL to proxy, just find successor proxy along circle
- **Handles addition/removal of servers much better**
  - E.g., for 100 proxies, adding/removing proxy only invalidates $\sim 1\%$ of cached objects
  - But when proxy overloaded, load spills to successors
  - When proxy leaves, extra misses disproportionately affect successors, but will be split among multiple successors
- **Can also handle servers with different capacities**
  - Give bigger proxies more random points on circle

Cache Array Routing Protocol (CARP)

- **Different URL $\rightarrow$ proxy mapping strategy**
  - Let list of proxy addresses be $p_1, p_2, \ldots p_n$
  - For URL u, compute: $h_1 \leftarrow \text{HASH}(p_1, u), h_2 \leftarrow \text{HASH}(p_2, u), \ldots$
  - Sort $h_1, \ldots h_n$. If h_i is minimum, route request to p_i.
  - If h_i overloaded, spill over to proxy w. next smallest h
- **Advantages over consistent hashing**
  - Spreads load more evenly when server is overloaded, if overload is just unfortunate coincidence
  - Spreads additional load more evenly when a proxy dies

Overview

- **How routers handle overload affects how TCP (and other protocols) behaves**
- **Two router questions: drop policy, scheduling policy**
- **Can reduce congestion through content distribution**
  - Clients can cache, need techniques for consistency
  - Services can use a CDN, load-balancing becomes important