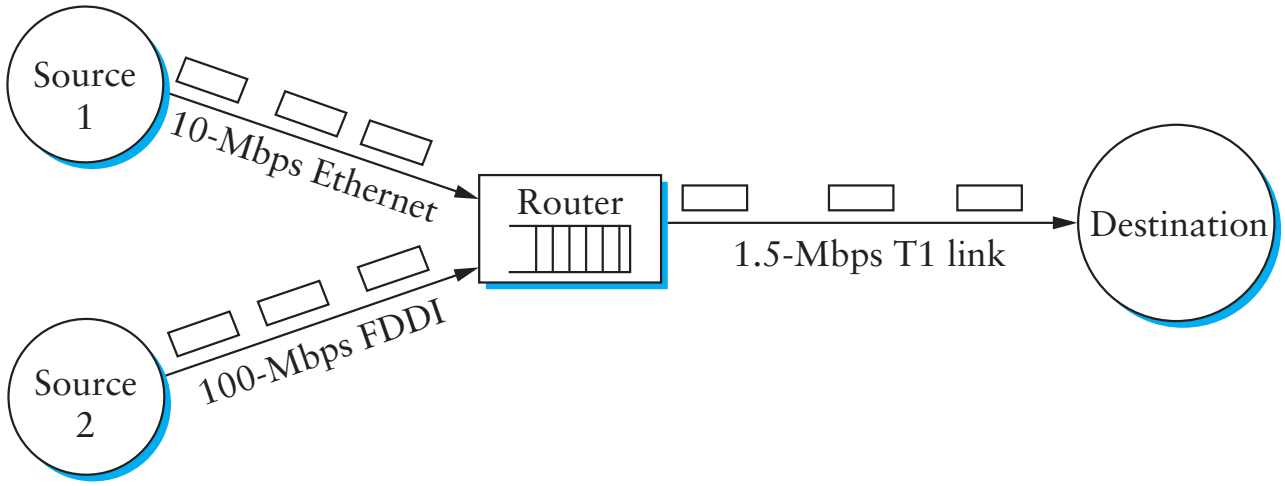
Reminders

- **Lab 3 due today**
- **Midterm exam Monday**
  - Open book, open notes, closed laptop
  - Bring textbook
  - Bring printouts of slides & any notes you may have taken
- **David Mazières moving office hours next week**
  - Monday 2:45-3:45pm (before exam, rather than next day)
- **Section Friday may be helpful for midterm review**

Congestion

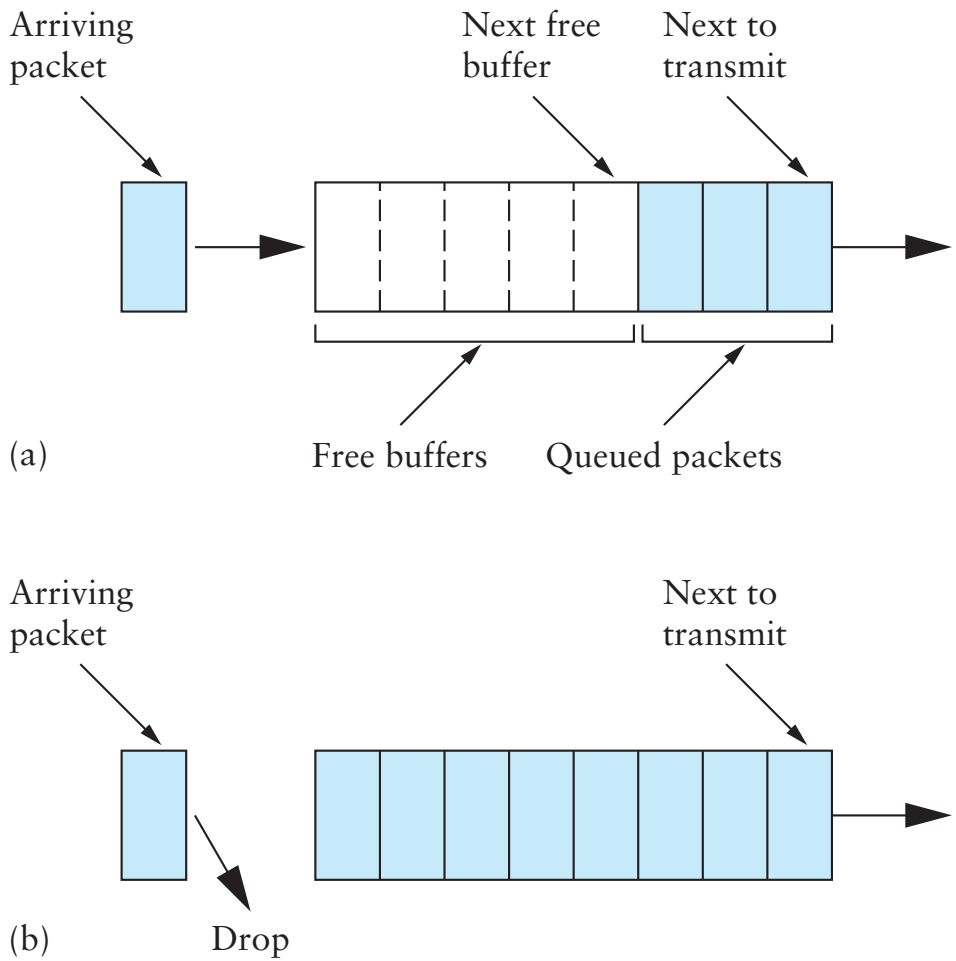


- **Can't sustain input rate $\gg$ output rate**
- **Previous lectures**
  - How should end nodes react
  - Avoid overloading the network
- **Today: What should the routers do?**
  - Prioritize who gets limited resources
  - Somehow interact well with TCP

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
  - Need to balance throughput & delay

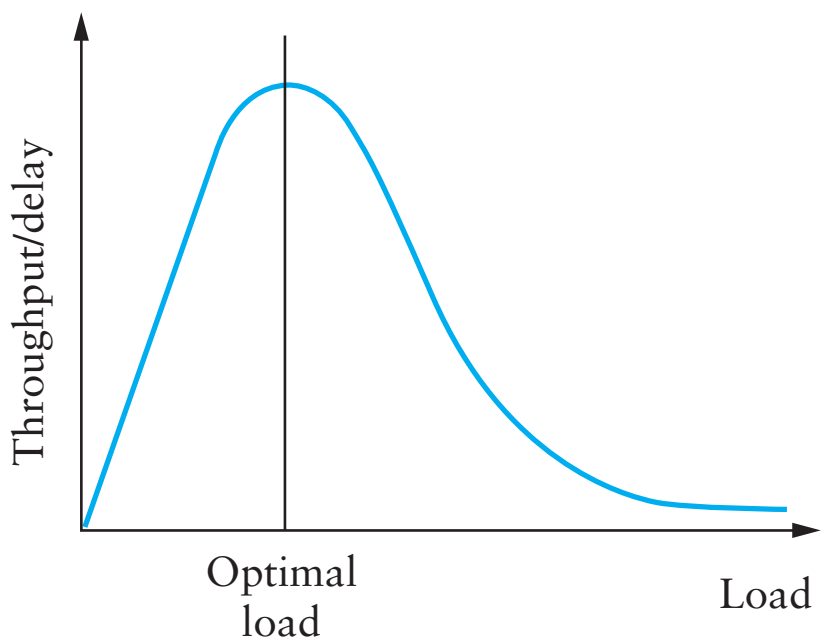
Example: FIFO tail drop



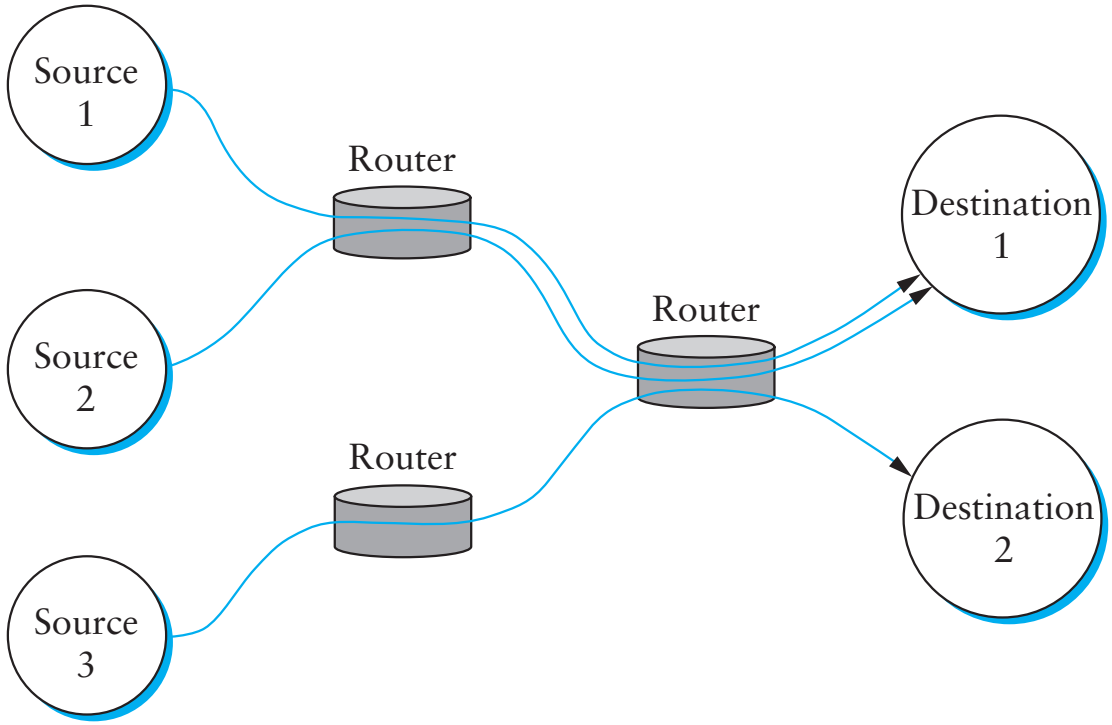
- **Differentiates packets only by when they arrive**
- **Might not provide useful feedback for sending hosts**

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

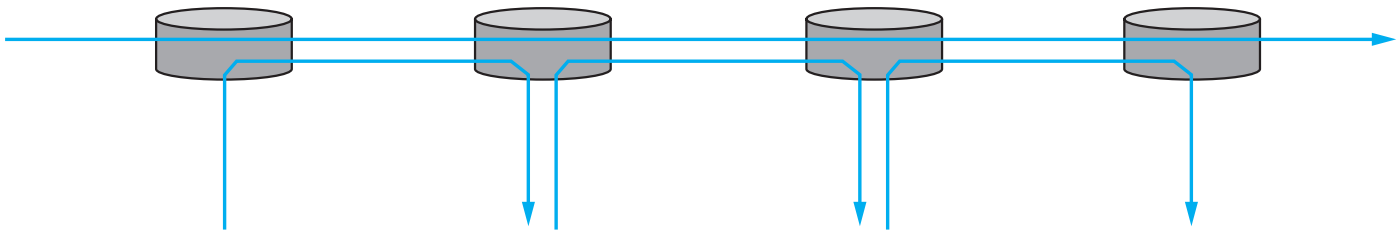


Connectionless flows



- **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



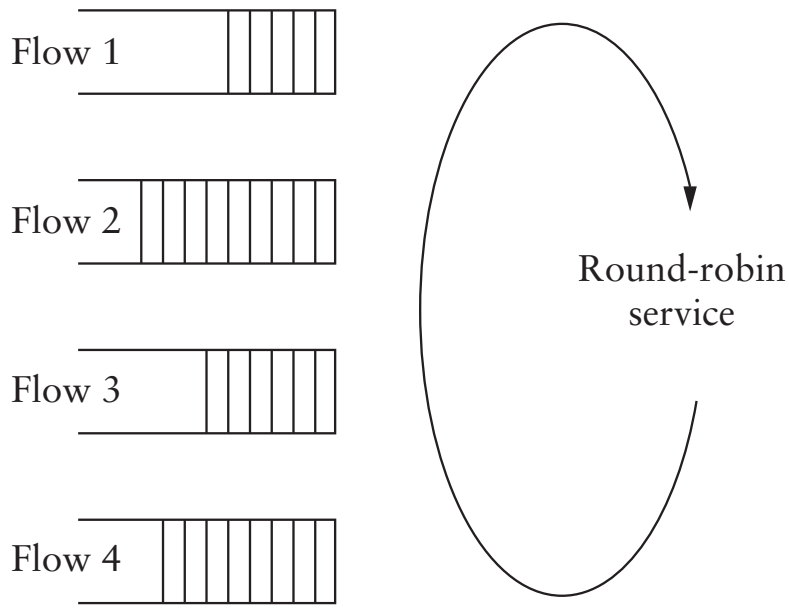
- **What is fair in this situation?**
  - Each flow gets $1/2$ link b/w? Long flow gets less?
- **Usually fair means equal**
  - For flow bandwidths $(x_1, \ldots, x_n)$, *fairness index*:

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

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

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**

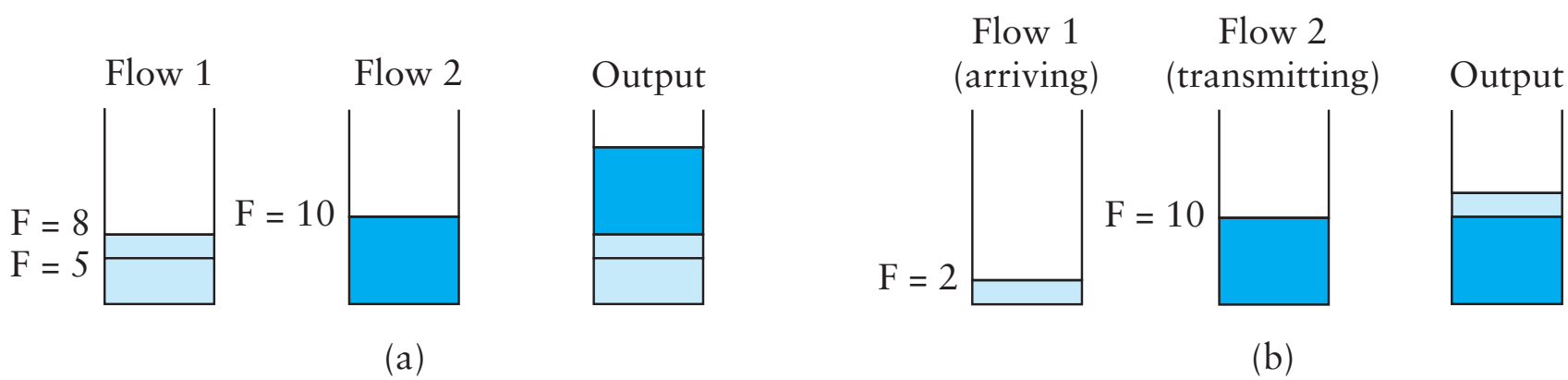


FQ Algorithm

- **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?**
  - 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:**



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*

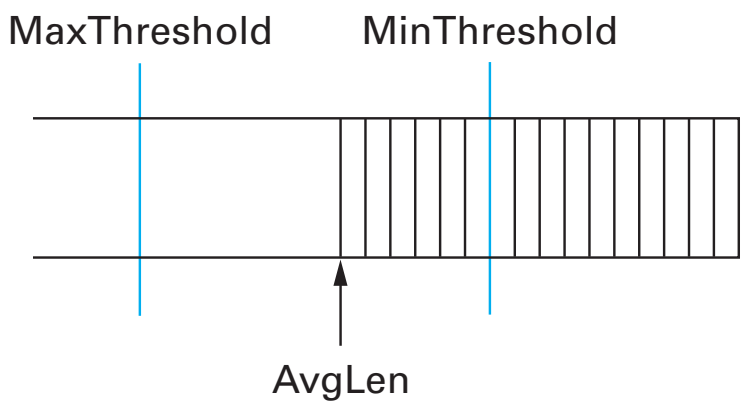
RED Details

- **Compute average queue length**

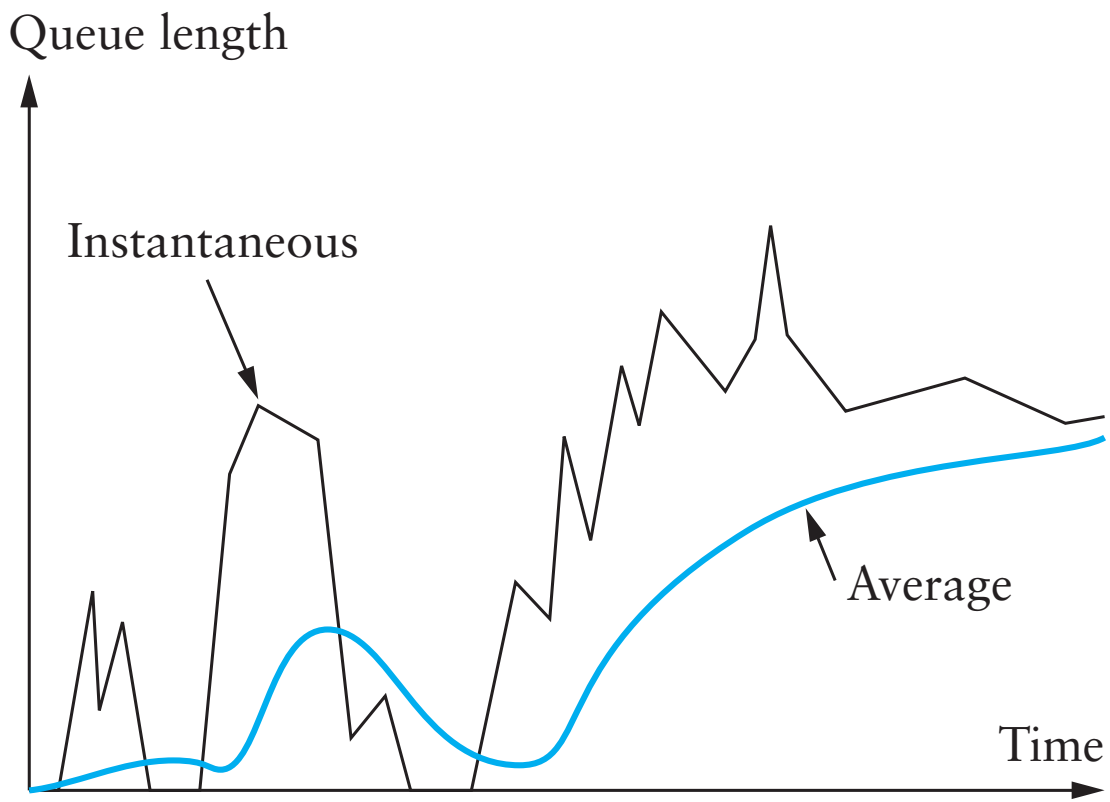
  $\text{AvgLen} = (1 - \text{Weight}) \cdot \text{AvgLen} + \text{Weight} \cdot \text{SampleLen}$

  $0 < \text{Weight} < 1$ **(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

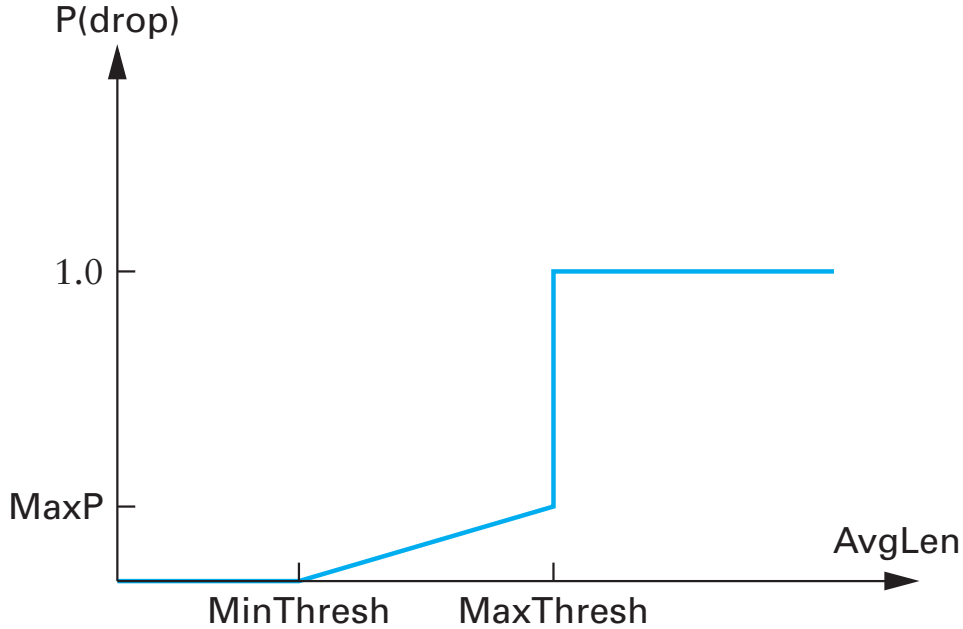
RED Details (cont)

- **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 ManThreshold <= AvgLen then
    drop arriving packet
```

RED Details (cont)

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



- **Actual probability depends on how recently dropped**
  - $\text{count} = \#$ pkts since drop or $\text{MinThresh} < \text{Avglen} < \text{MaxThresh}$
  - $P = \text{TempP}/(1 - \text{count} \cdot \text{TempP})$
  - Otherwise, drops not well distributed, and since senders bursty will overly penalize one sender

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, 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 **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

FPQ

- **Problem: Tuning RED can be slightly tricky**
- **Observations:**
  - TCP performs badly with window size under 4 packets: Need 4 packets for 3 duplicate ACKs and fast retransmit
  - Can supply feedback through delay as well as through drops
- **Solution: Make buffer size proportional to #flows**
  - Few flows $\Longrightarrow$ low delay; Many flows $\Longrightarrow$ low loss rate
  - Window size is a function of loss rate (recall $W \approx \sqrt{\frac{8}{3 \cdot p}}$)
  - RTT $\approx$ Qlen
  - Transmit rate = Window size / RTT
  - Router automatically adjusts Qlen to slow rate keeping $W \geq 4$
- **Clever algorithm estimates number of flows**
  - Hash flow info, set bits, decay – requires reasonable storage

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

One approach: TTLs

- **Data is accompanied by "time-to-live" (TTL)**
- **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

Callback limitations

- **Callbacks problematic if node or network down**
  - Must deliver invalidation notices to perform updates
  - With write caching, must flush if read from elsewhere
- **Callbacks also have scalability issues**
  - E.g., server may have to track large number of caches
  - Store list on disk? Slow, lots of disk accesses
  - Store in memory? What happens after crash/reboot
- **Clients must clear callbacks – extra network traffic**
  - When evicting file from the cache
  - When shutting down (if polite)

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
- **Variation: One lease covers all callbacks for same client**

Web caching

- **Caching can occur at browser and/or proxies**
- **HTTP 1.1 Defines `Cache-Control:` header**
  - Allows for mix of TTL and polling strategies
- **In requests: Cache-control can contain:**
  - `no-cache` – Disables any caching
  - `no-store` – No persistent storing (if request sensitive)
  - `max-age=`*seconds* – Client wants cached object validated no more than *seconds* seconds ago
  - `max-stale`*[=seconds]* – Client is willing to accept response that expired *seconds* seconds ago
  - `min-fresh=`*seconds* – Requests object w. TTL $\geq$ *seconds*
  - `only-if-cached` – Don't forward request to server

Web caching (continued)

- **In responses, Cache-control can contain:**
  - `public` – Cached object okay for multiple users
  - `private` – Object only valid for the particular user
  - `no-cache` – Clients/proxies should not cache
  - `no-store` – Clients should not store to disk
  - `no-transform` – E.g., don't compress images
  - `max-age=`*seconds* – TTL for object in cache
  - `must-revalidate` – *Really* obey `max-age` TTL (Some proxies not strict—might return cached object if network down)
  - `proxy-revalidate` – Like `must-revalidate`, but only for proxies, not browser cache

Web cache hierarchies

- **Bigger caches mean bigger hit rates**
  - More users $\Longrightarrow$ more likely to access same object
  - More capacity $\Longrightarrow$ more likely to hit in cache
- **Also raises scalability issues**
- **Solution: Cache hierarchies**
  - Proxies can funnel all requests through *parent* cache
    child reduces latency, reduces load (not storage) of parent
  - Proxies can check nearby *sibling* caches, get data from
    sibling if available, else go to server
- **Internet Cache Protocol over UDP**
  - Allows proxies to query each other in light-weight manner

ICP packet format

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Opcode    |    Version    |         Message Length        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Request Number                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             Options                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Option Data                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Sender Host Address (not really used)   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|                             Payload                           |
/                                                               /
/                                                               /
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

ICP protocol (continued)

- **Some opcodes**
  - ICP_OP_QUERY – asks if proxy has URL specified in payload
  - ICP_OP_HIT – yes, proxy has the URL
  - ICP_OP_MISS – no, proxy doesn't have the URL, but would fetch it recursively
  - ICP_OP_MISS_NOFETCH – proxy doesn't have & won't fetch
  - ICP_OP_DENIED – permission denied
  - ICP_OP_HIT_OBJ – for small objects, contents is in reply UDP packet (bad idea, because of fragmentation)
- **Options**
  - ICP_FLAG_HIT_OBJ – Requests ICP_OP_HIT_OBJ reply
  - ICP_FLAG_SRC_RTT – Requests network RTT to server be included in reply (16-bit # msec)

Cache Digests

- **Problem: ICP adds latency, network traffic**
- **Idea: Download list of everything in other caches**
  - Just need a way to compress it efficiently
- **Use *cache digests* (Bloom filters)**
  - Size is 5 bits per object in cache, computed as follows
  - Zero out bit array of size 5 bits / obj in cache
  - Hash each URL down into a 16-byte value with MD5
  - Break MD5 value into 4 32-bit integers, $k[1], k[2], k[3], k[4]$
  - Set bits $k[1]$ % *cache-size*, $k[2]$ % *cache-size*, ...
  - Can check and learn that object is not in cache, or that object is probably in cache
- **Used by squid proxy cache**

Reverse proxies

- **Proxies can also be configured *server-side***
  - For servers with insufficient network connectivity
  - Redirect clients into large proxy networks to absorb load
  - Technique used by commercial CDNs (e.g., Akamai)
- **Choice of proxy determined by server in URL, not client making request**
- **Could redirect all traffic to one proxy per server**
  - Very good for hit rate, not so good for scalability
- **Or distribute requests for same server to many proxies**
  - Will have low hit rate for less popular URLs
- **Also don't assume proxies 100% reliable**

Straw man: Modulo hashing

- **Say you have N proxy servers (e.g., 100)**
- **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

- **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
- **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:
    $h_1 \leftarrow \mathrm{HASH}(p_1, u), h_2 \leftarrow \mathrm{HASH}(p_2, u), \ldots$
  - 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

Disconnected operation

- **Until now, have considered caching mostly for performance**
  - Could go to server, but slow
- **Also allows additional functionality with *disconnected operation***
  - Example: PDA cannot always sync with desktop
  - Now we *can't* guarantee desired consistency
- **Approach:**
  - Merge multiple unrelated updates
  - Detect if two updates conflict
  - Somehow resolve conflicts (e.g., involve user)