Queuing and Scheduling

Outline

- What is Queuing
- What is scheduling
- Why we need it

- Various Queuing Mechanisms
- Requirements of a scheduling discipline
- Fundamental choices
- Scheduling best effort connections
- Scheduling guaranteed-service connections
- Packet drop strategies
- Traffic Policing

Queuing

- A buffering or queuing point is necessary when
  - there is a rate mismatch
  - contention for resources (e.g., output port contention)
  - to control QoS by treating different classes of traffic independently
- Basic functions
  - Add a packet to the appropriate queue (as indicated by context, classification, etc.)
  - Drop a packet when queue is full by monitoring queue occupancy
  - Remove a packet for transmission when so directed by the scheduler and the scheduling discipline
    - ☞ *Scheduler controls when a packet leaves the queue*

Scheduling

- A scheduling discipline does two things:
  - decides service order
  - manages queue of service requests
- Example:
  - consider queries awaiting web server
  - scheduling discipline decides service order
  - and also if some query should be ignored

Where?

- Anywhere where contention may occur
- At every layer of protocol stack
- Usually studied at network layer, at output queues of switches

Why do we need one?

- *Because applications need it*
- We expect two types of applications
  - best-effort (adaptive, non-real time)
    - e.g. email, some types of file transfer
  - guaranteed service (non-adaptive, real time)
    - e.g. packet voice, interactive video, stock quotes

What can scheduling disciplines do?

- Give different users different qualities of service
- Example of passengers waiting to board a plane
  - early boarders spend less time waiting
  - bumped off passengers are ‘lost’!
- Scheduling disciplines can allocate
  - bandwidth
  - delay
  - loss
- They also determine how *fair* the network is

Ideal Queue and Scheduling Disciplines

- **Flexibility:** To support different services, and easily evolve to support new services
- **Scalability:** The implementation should be simple to allow scalability to large number of connections and allow for cost-effective implementation.
- **Efficiency:** In maximizing the network link utilization (i.e. maximize the throughput).
- **Guaranteed QoS:** To provide low jitter and end-to-end delay bounds for real-time traffic. Allow implementation of simple CAC functions
- **Isolation**: To reduce interference between service classes and connections.
- **Fairness**: To allow fast and fair re-distribution of bandwidth that is dynamically available. The fairness can be defined by a flexible policy.

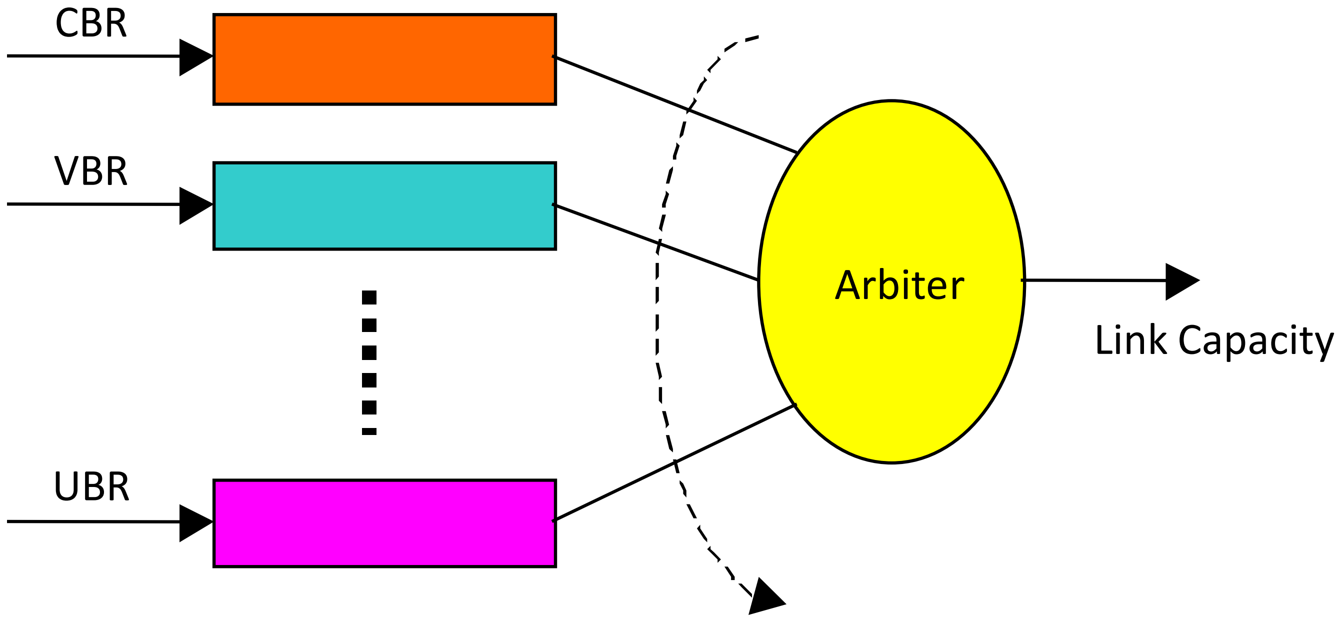
Queuing Structures for QoS Guarantees

- ***per-Group queuing***
  - the connections are categorized into *groups* as per QoS requirement
  - The traffic from each different group is queued up separately
  - Scheduling policy decides when to send out a packet of a given group as per the QoS strategy
  - Will not be able to differentiate QoS requirements for individual connections
- ***per-connection (or tunnel) queuing***
  - the traffic of each virtual connection (VC) is queued independently
  - Expensive to implement both queuing and scheduling
  - Complete control on per-connection QoS
- Combination

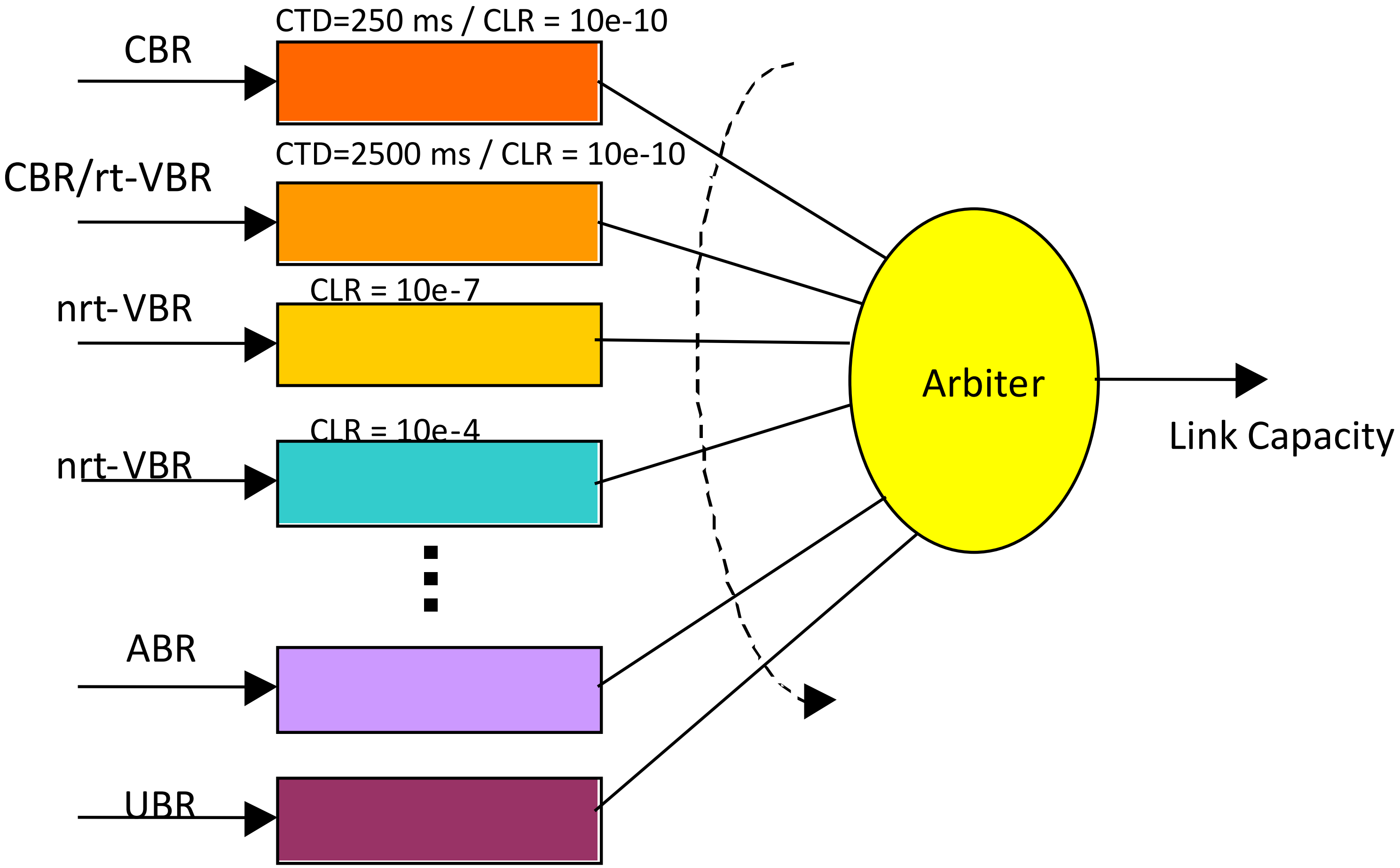
per-Group Queuing

- Typical groups consist of connections belonging to the same:
  - service category
  - service class
  - conformance definition

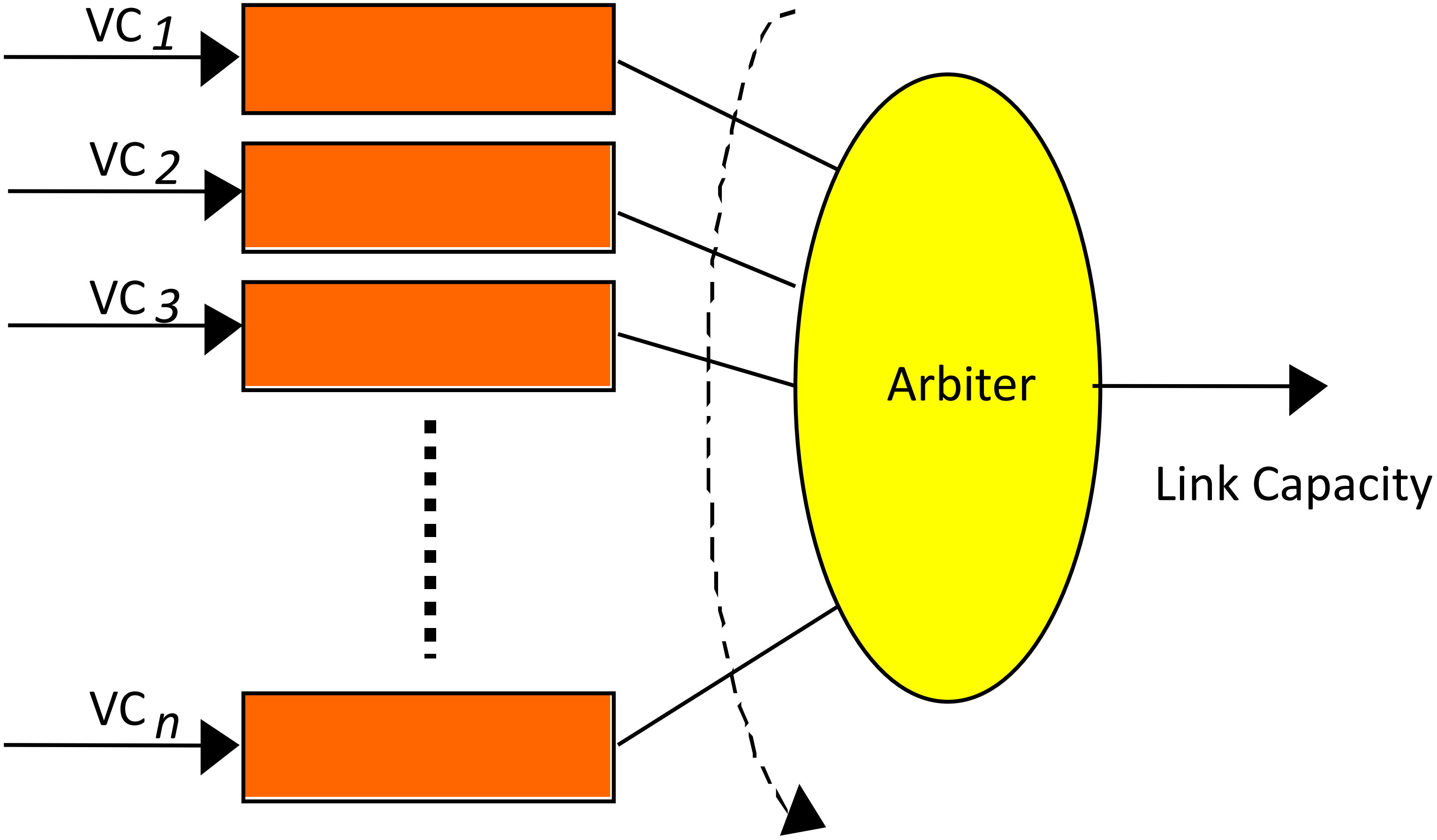
Group as Service Category



Group as Service Class



per-Virtual Connection (per-VC) Queueing



How many such Queues??

Buffer Management

- Applicable to both per-group and per-connection Queueing
- Queueing is a logical concept
- The packets are actually stored in a physical buffer (memory)
- Buffer management deals with:
    - How the physical memory is allocated to the various queues?
    - Is it completely partitioned among the queues?
    - Is it completely shared among the queues?
    - Congestion control

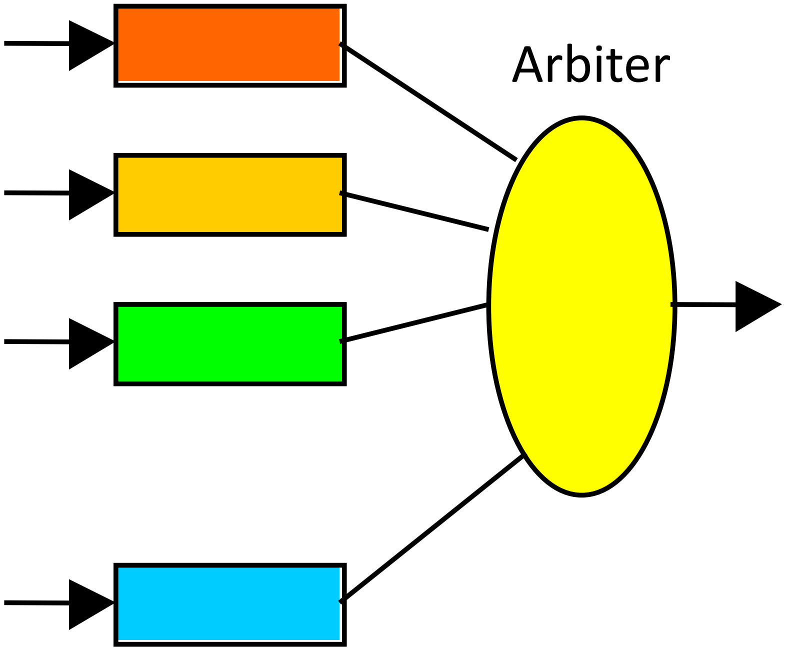
(We will return to this topic a little later!!)

Scheduling

- Sharing always results in contention
- A contention point can have a number of queues.
- An *arbitration (or scheduling*) function is therefore needed to extract packets from these queues and transmit (or serve) them appropriately to meet the QoS objectives for each connection
- A *scheduling discipline* resolves contention: *who's next?*
- An *arbiter* or *arbitration function* or *scheduler* is responsible for this scheduling and allocation of bandwidth to the queues
- An arbiter implements a *scheduling algorithm* or *scheduling mechanism*
- Key to *fairly sharing resources* and *providing performance guarantees*
- Depending upon how the queues and arbiters are organized, scheduling schemes can be divided into *flat (or single)-level scheduling* and *hierarchical scheduling*.

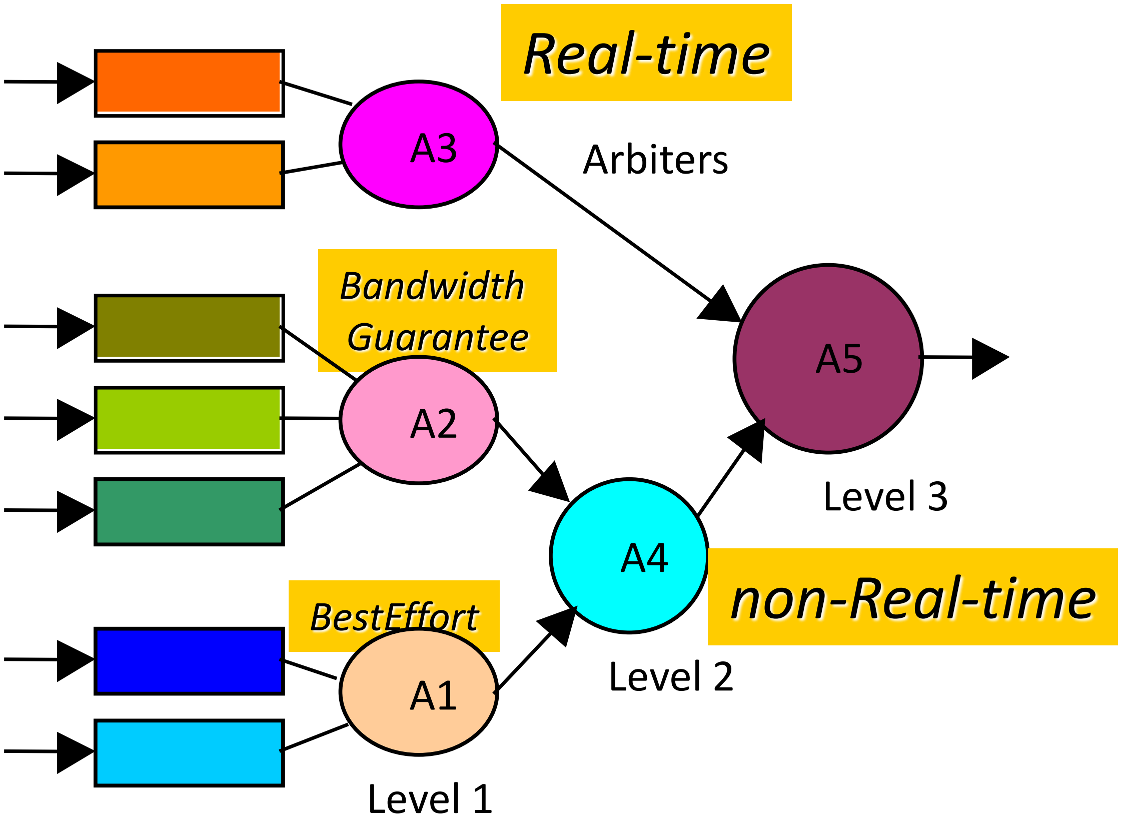
Flat Scheduling

- In a flat (or single)-level arbitration, a contention point uses a flat arbitration scheme with one single arbiter serving all the queues.



Hierarchical Scheduling

- In order to divide the bandwidth in a more flexible and accurate way, the arbitration can be performed in multiple levels or *hierarchically*



Hierarchical Scheduling (contd ..)

- Hierarchical scheduling uses multiple arbitration functions.
- One arbiter is used to arbitrate between a set of queues and a few other arbitration functions are used in a hierarchical fashion to arbitrate between the arbiters.
- Each of the arbiters in the hierarchy can implement a different scheduling algorithm to achieve specific bandwidth partitioning and control
- Hierarchical scheduling is useful to divide the bandwidth on a link between different sets of queues (e.g., queues belonging to a given customer, to a given service etc.).
- With hierarchical scheduling, the bandwidth left over by a connection is redistributed first between the connections within the same set. A flat arbitration scheme cannot achieve this type of bandwidth redistribution.

Requirements

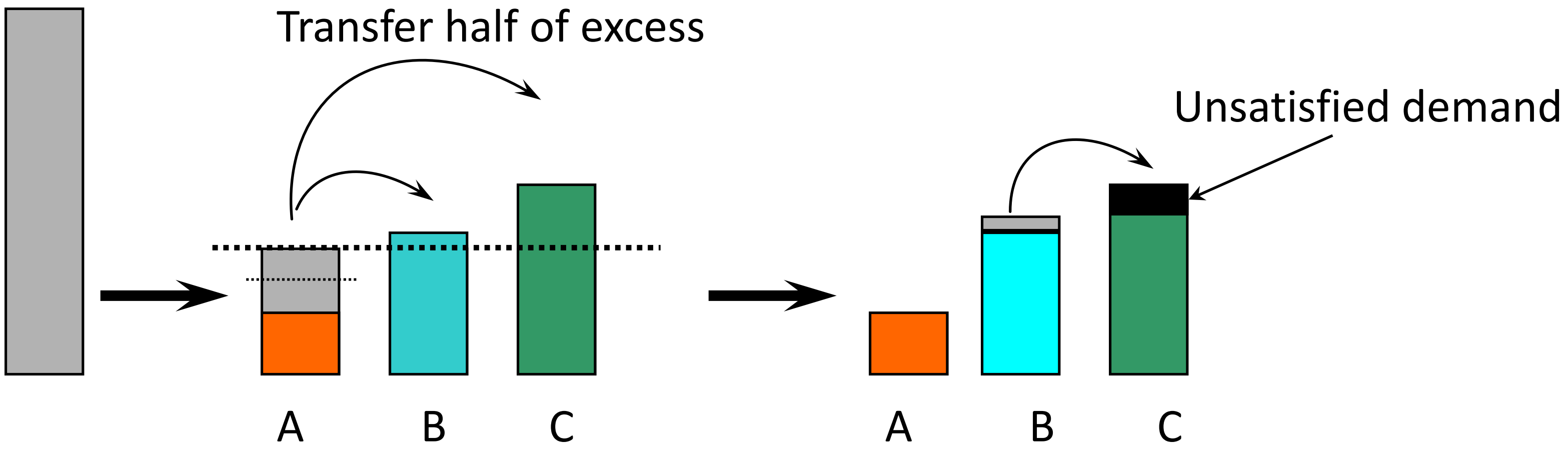
- An ideal scheduling discipline
  - is easy to implement
  - is fair
  - provides performance bounds
  - allows easy *admission control* decisions
    - as it controls how much bandwidth is finally given
    - to decide whether a new flow can be allowed

Requirements: Ease of implementation

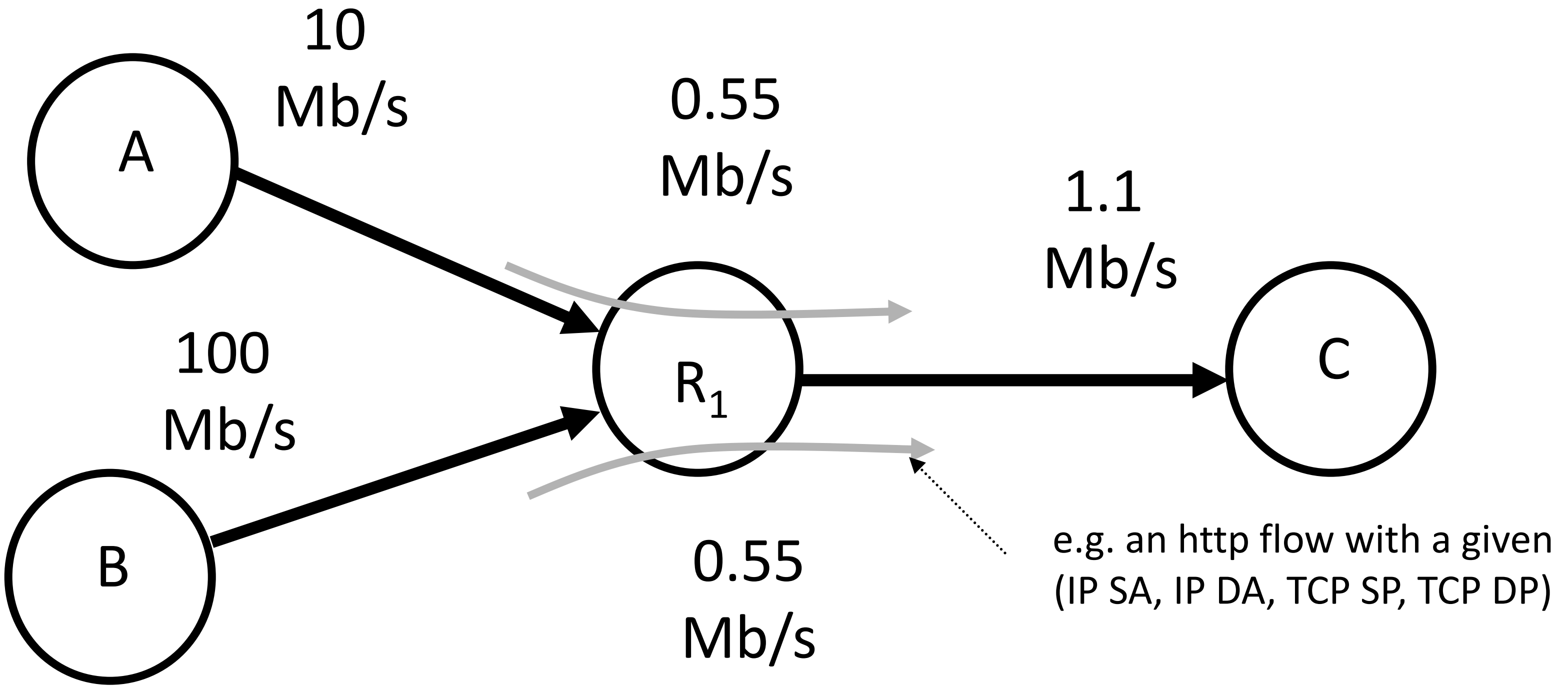
- Scheduling discipline has to make a decision once every few microseconds!
- Should be able to implement in a few instructions or hardware
  - for hardware: critical constraint is VLSI *space*
- Work per packet should scale less than linearly with number of active connections

Requirements: Fairness

- Scheduling discipline *allocates* a *resource*
- An allocation is fair if it satisfies *max-min fairness*
- Intuitively
  - each connection gets no more than what it wants
  - the excess, if any, is equally shared



Fairness



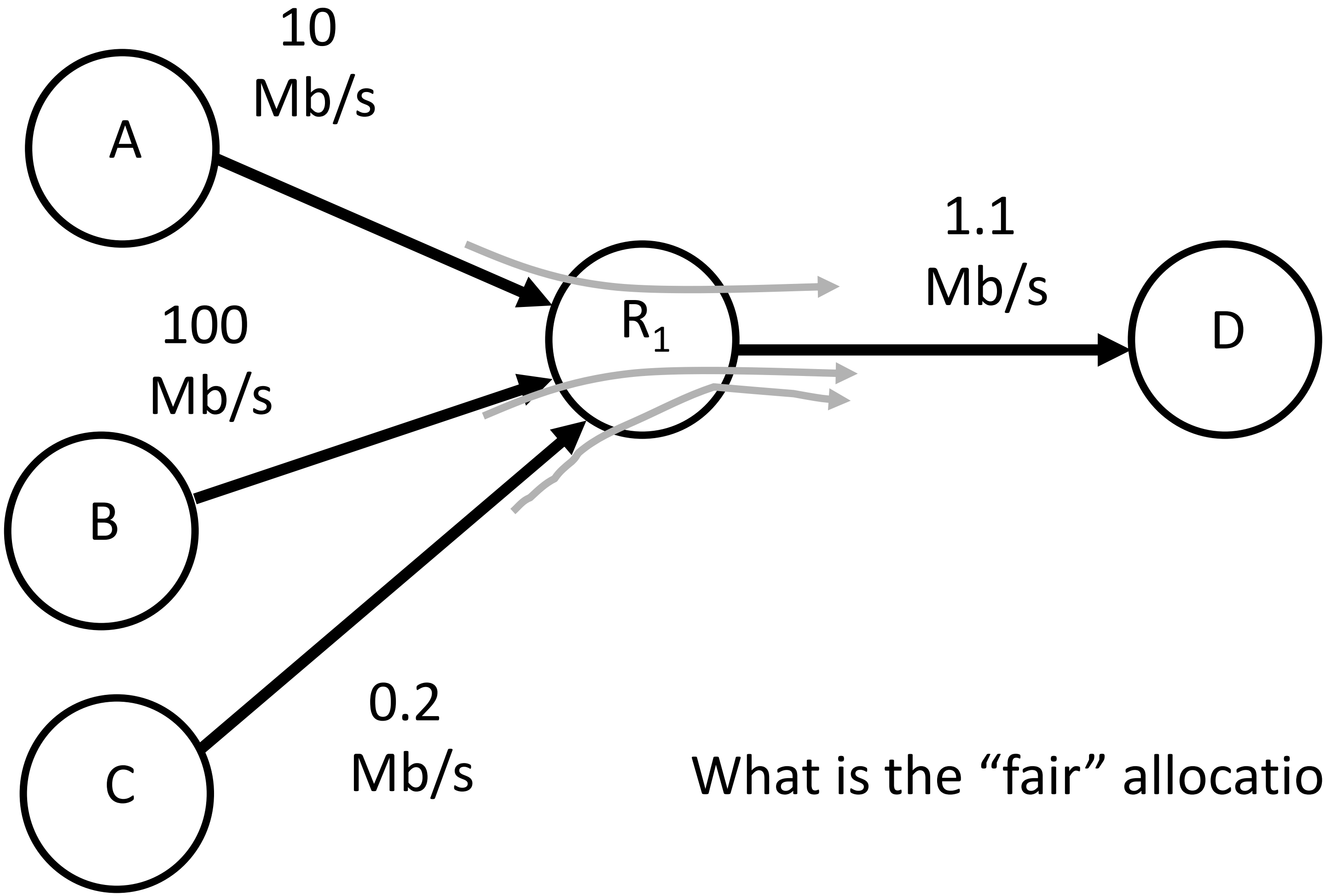
e.g. an http flow with a given
(IP SA, IP DA, TCP SP, TCP DP)

What is the “fair” allocation:
(0.55Mb/s, 0.55Mb/s) or (0.1Mb/s, 1Mb/s)?

Weighted Fairness

Courtesy Nick McKeown@Stanford

Fairness



What is the “fair” allocation?

Courtesy Nick McKeown@Stanford

Max-Min Fairness

A common way to allocate flows

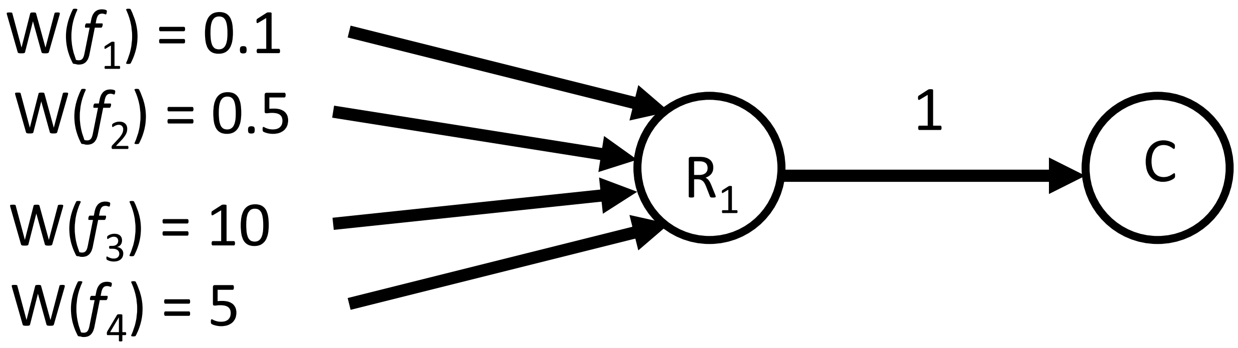
N flows share a link of rate *C*. Flow *f* wishes to send at rate W(*f*), and is allocated rate R(*f*).

1. Pick the flow, *f*, with the smallest requested rate.
2. If W(*f*) < *C/N*, then set R(*f*) = W(*f*).
3. If W(*f*) > *C/N*, then set R(*f*) = *C/N*.
4. Set *N = N – 1*. *C = C* – R(*f*).
5. If *N* > 0 go to 1.

Courtesy Nick McKeown@Stanford

Max-Min Fairness

An example



$W(f_1) = 0.1$

$W(f_2) = 0.5$

$W(f_3) = 10$

$W(f_4) = 5$

R_1

1

C

Round 1: Set $R(f_1) = 0.1$

Round 2: Set $R(f_2) = 0.9/3 = 0.3$

Round 3: Set $R(f_4) = 0.6/2 = 0.3$

Round 4: Set $R(f_3) = 0.3/1 = 0.3$

Courtesy Nick McKeown@Stanford

Max-Min Fairness

- How can an Internet router “allocate” different rates to different flows?
- First, we will see how a router can allocate the “same” rate to different flows…
- And then extend it
  - per-flow Queueing
  - Weighted Fair Queueing

Fairness (contd..)

- Fairness is *intuitively* a good idea
- But it also provides *protection*
  - traffic hogs cannot overrun others
  - automatically builds *firewalls* around heavy users
- Fairness is a *global* objective, but scheduling is local
  - *How to achieve global max-min fairness??*
- Each endpoint must restrict its flow to the smallest fair allocation
- Dynamics + delay => global fairness may never be achieved

- We will revisit Global Fairness again a little later !!

Requirements: Performance bounds

- What is it?
  - A way to obtain a desired level of service
- Can be *deterministic* or *statistical*
- Common parameters are
  - bandwidth
  - delay
  - delay-jitter
  - loss

Bandwidth

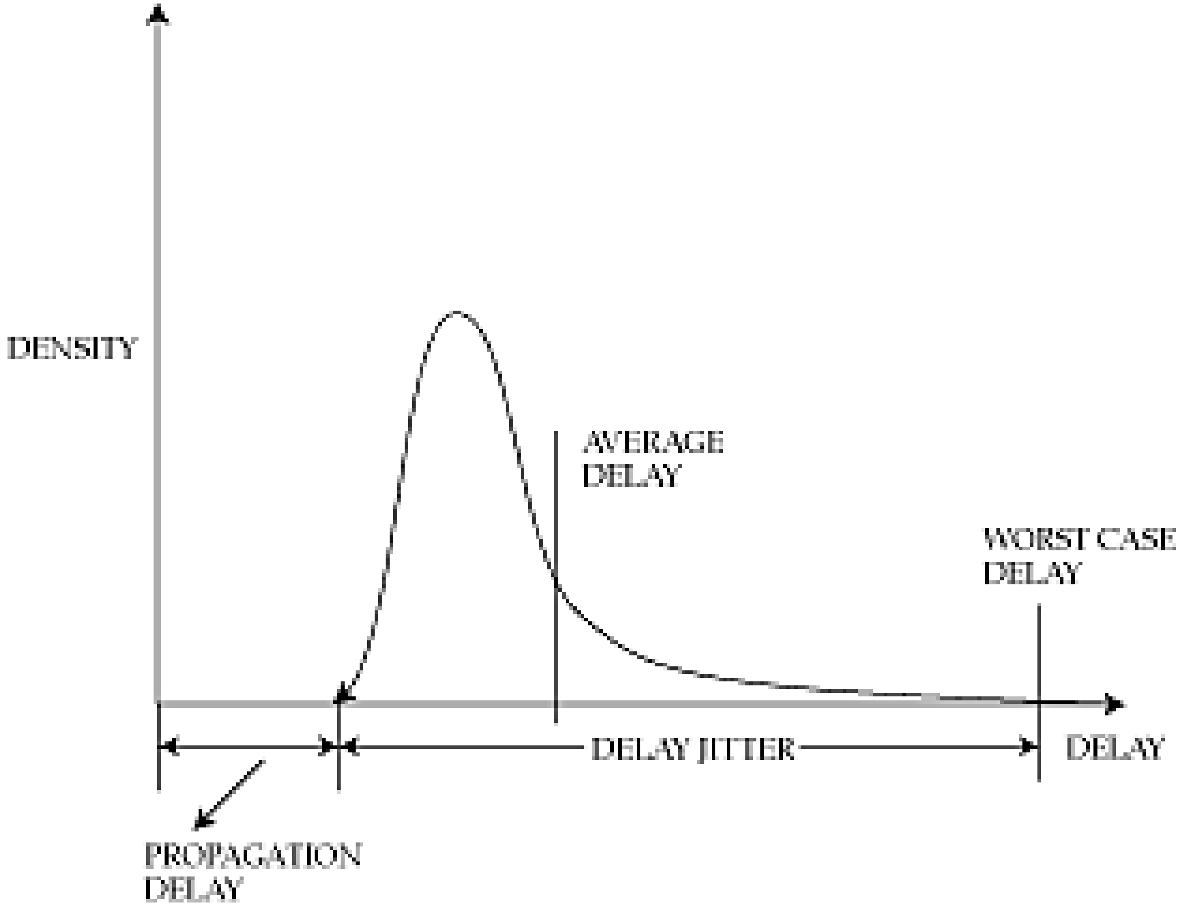
- Specified as minimum bandwidth measured over a pre-specified interval
- E.g. > 5Mbps over intervals of > 1 sec
- Meaningless without an interval!
- Can be a bound on average (sustained) rate or peak rate
- Peak is measured over a ‘small’ interval
- Average is asymptote as intervals increase without bound

Req'ments: Ease of admission control

- Admission control needed to provide QoS
- Overloaded resource cannot guarantee performance
- Choice of scheduling discipline affects ease of admission control algorithm
  - E.g., Minimum rate guarantees cannot be provided for low priority traffic in a priority scheduling discipline

Delay and delay-jitter

- Bound on some parameter of the delay distribution curve



Outline

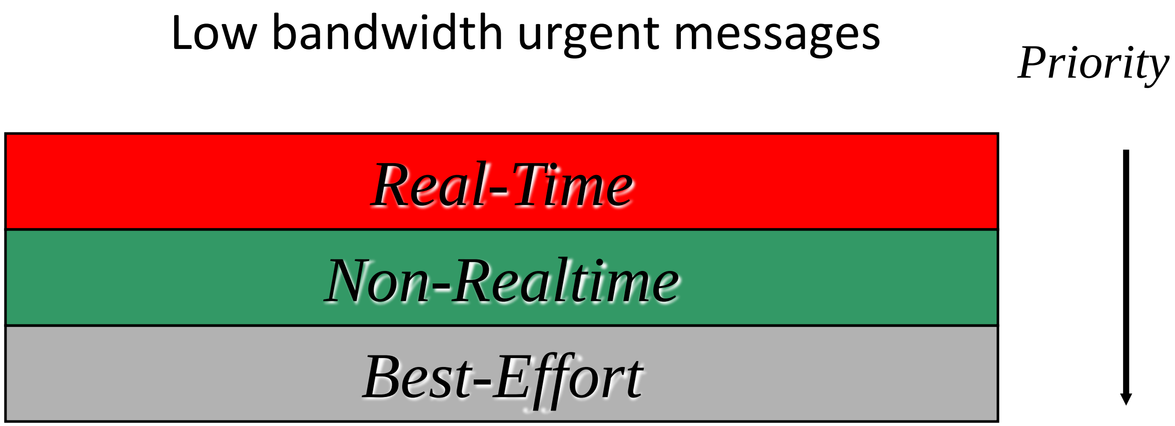
- What is scheduling
- Why we need it
- Requirements of a scheduling discipline
- Fundamental choices
- Scheduling best effort connections
- Scheduling guaranteed-service connections
- Packet drop strategies

Fundamental choices (Types of Scheduling)

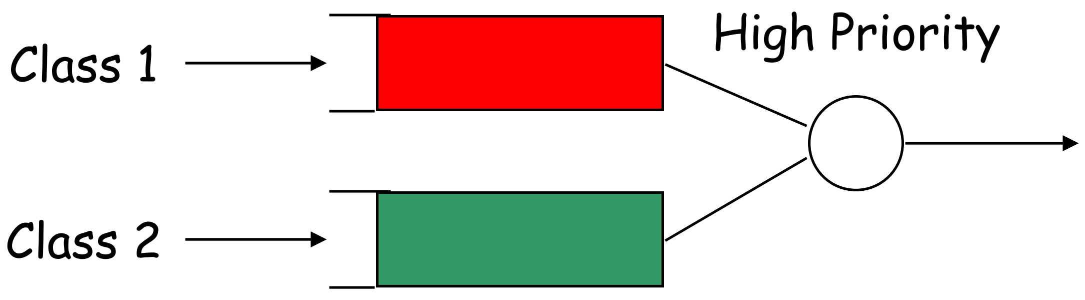
- Number of priority levels (Multiple-priority scheduling)
  - Degree of aggregation
- Work-conserving vs. non-work-conserving
- Service order within a level
  - Fair Queueing
- Traffic shaping

Non-Preemptive Priority Scheduling

- Packet is served from a given priority level only if no packets exist at higher levels (*multilevel priority with exhaustive service*)
- However cannot bump a low priority packet that is already in service
- Highest level gets lowest delay
- Watch out for starvation!
- Usually map priority levels to delay classes



Priority Scheduling (contd ..)

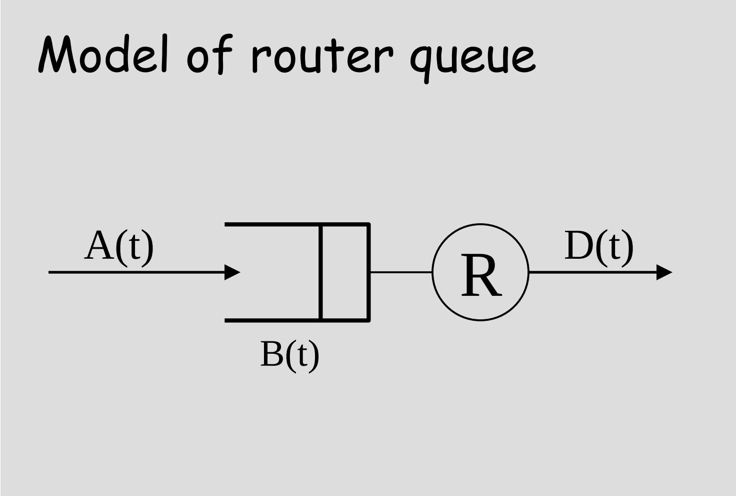


- Service for low priority depends on high priority traffic
- Class 1 gets best QOS (better than needed!)
- Difficult to develop multiple queues and services with guaranteed QOS on this platform
- Difficult to develop effective and efficient CAC for low priority services
  - Service Variability

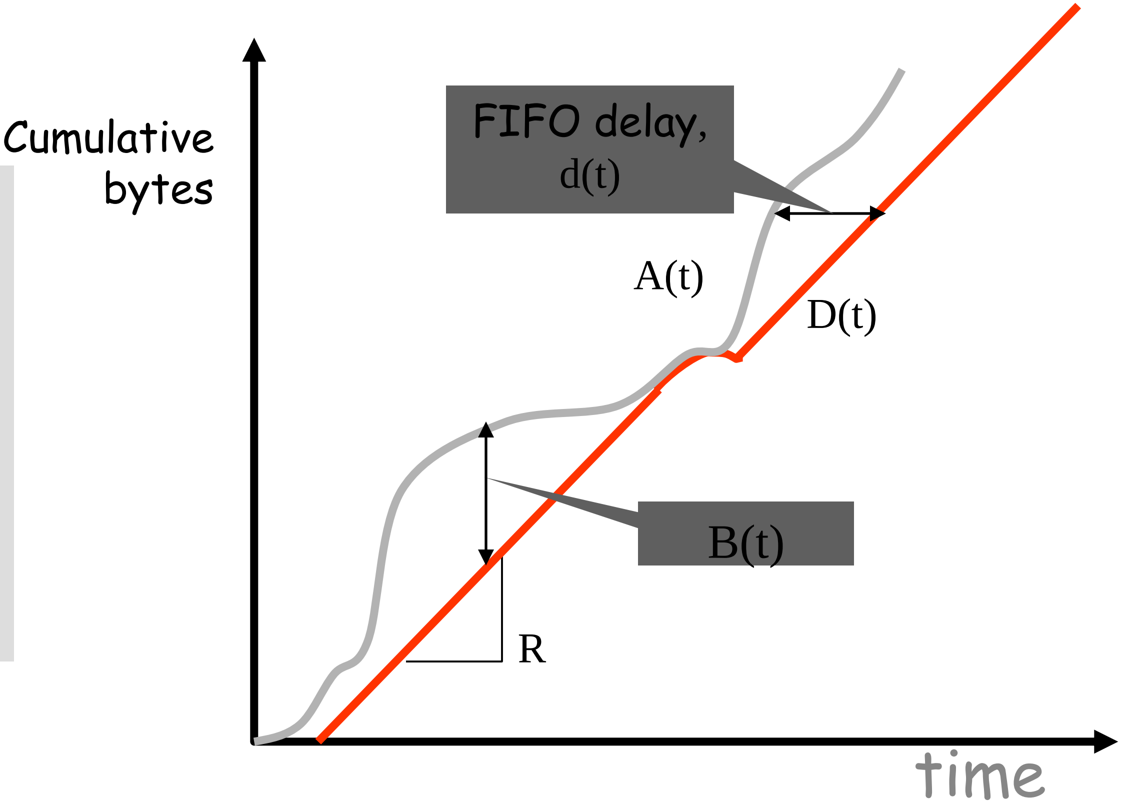
Priority Scheduling (contd ..)

- The priority scheduling can be applied to both per-Group queuing as well as per-virtual connection (VC) queuing.
- In case of per-VC queuing, all the VC queues belonging to a particular service category can be viewed as at one priority level
- Multi-priority scheduling is very simple and efficient to arbitrate between a small number of queues
- It can provide for example, a class of real-time traffic and a class of non-real-time traffic with guaranteed loss rate and a “best effort” class
- Increasing the number of classes beyond this may be insufficient to provide fine-grained QoS commitments unless some level of fair access to the bandwidth is provided such as *Weighted Fair Queueing (WFQ)*

Deterministic analysis of a router queue



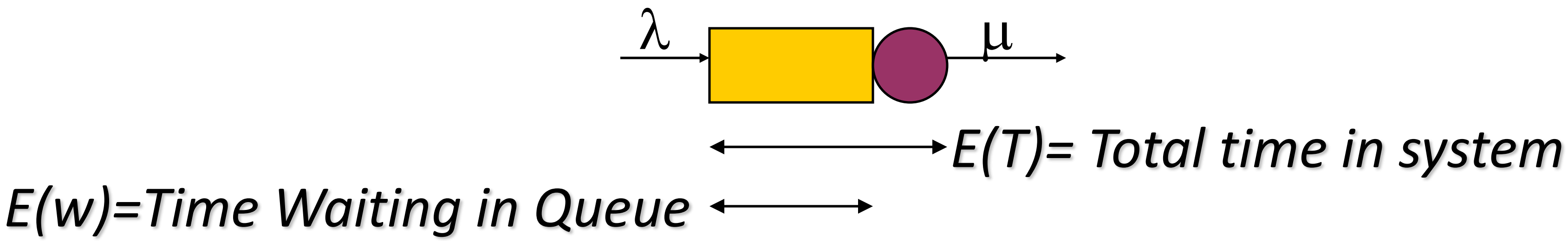
Model of router queue



Courtesy Nick McKeown@Stanford

A Simple Queueing Analysis

- Let us consider a single class *M/M/1* queue with Poisson arrivals and exponential service distribution
  - λ is the packet arrival rate
  - μ is the packet departure rate ($1/\mu$ is the packet service time)
  - ρ is the queue utilization = λ/μ
  - Expected queue length =E(n) = $\rho/(1-\rho)$
  - Expected packet delay =E(T) = E(n)/ λ= $1/\mu(1-\rho)$
  - Expected packet delay E(T) = Expected Waiting Time in the queue E(w) + Expected Service Time ($1/\mu$)



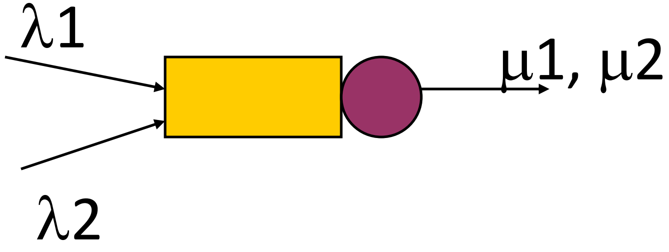
E(T)= Total time in system

E(w)=Time Waiting in Queue

A Simple Queueing Analysis (contd ..)

- What if two different streams (e.g., Class 1 and Class 2) combined in the same FIFO? (M/G/1 Analysis)
  - $\lambda 1$, $\lambda 2$ is the packet arrival rate for each of the classes
  - $\mu 1$, $\mu 2$ is the packet departure rate ($1/\mu_i$ is the packet service time) for each class separately
  - ρ is the queue utilization = $\rho 1 + \rho 2 = (\lambda 1/\mu 1) + (\lambda 2/\mu 2)$
  - μ = average service rate = $(\rho 1.\ \mu 1 + \rho 2.\ \mu 2)/(\rho 1 + \rho 2)$
  - Expected queue length =E(n) = $\rho/(1\text{-}\rho)[1\text{-}\ \rho(1\text{-}\ \mu^2\sigma^2)/2]$
  - Expected packet delay =E(T) = E(n)/ $(\lambda 1 + \lambda 2)$
  - Expected packet delay E(T) = Expected Waiting Time in the queue E(w) + Expected Service Time (1/ μ)

Variance of Service

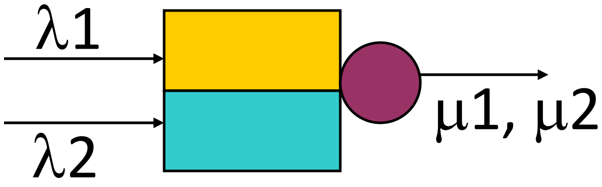


Both classes get same average delay!!

A Simple Queueing Analysis (contd ..)

- Extend the same system with two priorities (non-preemptive)
  - λ_1, λ_2 is each class packet arrival rate
  - μ_1, μ_2 is each class packet departure rate (or) 1/ μ_1 and 1/ μ_2 are the mean packet service time. Assume S_k^2 is the second moment of the service for class i (i=1,2)
  - ρ is the queue utilization = $\rho_1 + \rho_2$
  - Expected waiting time for class 1 =$E(w_1)=E(R)/(1-\rho_1)$
  - Expected waiting time for Class 2=$E(w_2)=E(R)/(1- \rho_1)(1- \rho_1- \rho_2)$
  - E(R)=mean residual service time

    $=[\lambda_1 \overline{S_1^2} + \lambda_2 \overline{S_2^2}]/2$



  - In general for k priority queue
  - $E(w_k)=E(R)/(1- \rho_1 -..- \rho_{k-1})(1- \rho_1 -..- \rho_k)$

Kleinrock’s Conservation Theorem

- Kleinrock’s conservation theorem states that

$$\boxed{\sum_{k=1}^{K} \rho_k \overline{W}_k = \frac{\rho \overline{R}}{1-\rho}}$$

where $\rho = \rho_1 + \cdots + \rho_K$ is the total load of the system.

- The weighted sum of the waiting times can never change no matter how sophisticated the queueing discipline may be
- Any attempt to modify the queueing discipline so as to reduce one of the average waiting times will force an increase in some of the other waiting time

Degree of aggregation

- More aggregation
  - less state
  - cheaper
    - smaller VLSI
    - less to advertise
  - BUT: less individualization
- Solution
  - aggregate to a *class,* members of class have same performance requirement
  - no protection within class

Service within a priority level

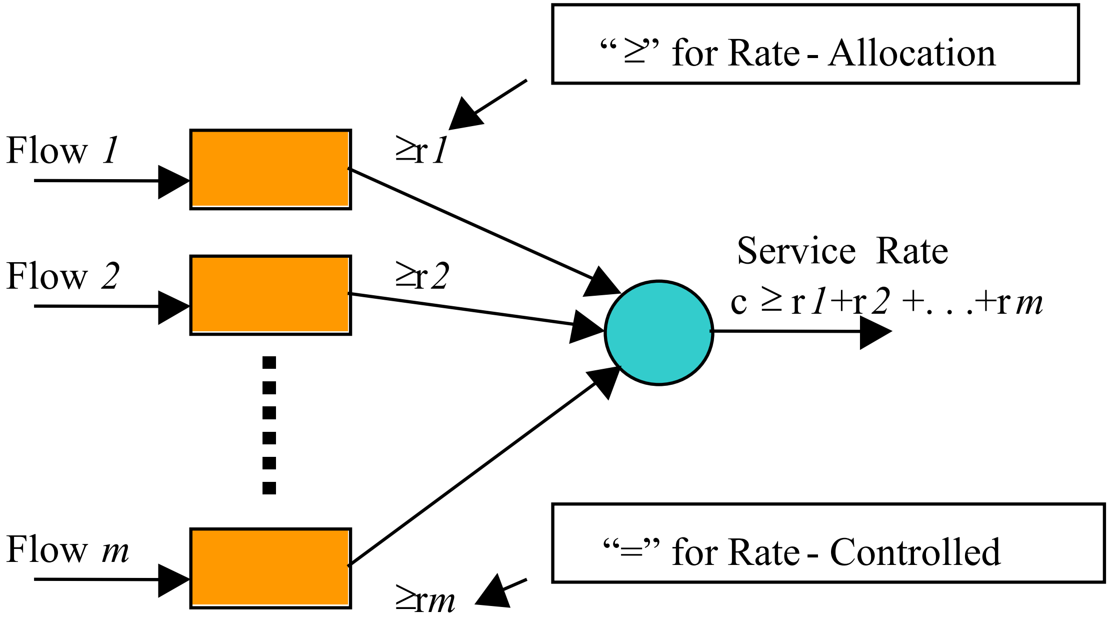
- In order of arrival (First Come First Served FCFS) or in order of a service tag
- Service tags => can arbitrarily reorder queue
  - Need to sort queue, which can be expensive
- FCFS
  - bandwidth hogs win (no protection)
  - no guarantee on delays
- Service tags
  - with appropriate choice, both protection and delay bounds possible
    - Fair Queueing Techniques

Fair-Share Scheduling

- Alternative to Priority Scheduling
- Each queue is guaranteed to get its share of link bandwidth according to its needs
- Different queues may have different bandwidth requirements and thus a weight can also be associated with a queue
  - The scheduler divides the bandwidth amongst each queue based on the *weighted fair-share*
- Fair-share scheduling is a concept that introduces firewalls (or flow isolation) between various traffic flows in a queuing system and thus the interaction between the flows is minimized
- Fair-share scheduling basically deals with how the leftover (or idle) bandwidth is distributed among the connections
- Fair-share scheduling attempts to distribute the bandwidth among competing connections in a fair manner

Fair-Share Scheduling (contd ..)

- The fair-share scheduling algorithms guarantee certain minimum rate between the flows
- These “rate-based” mechanisms can be classified into two categories
  - *rate-allocation service* discipline
  - *rate-controlled* service discipline



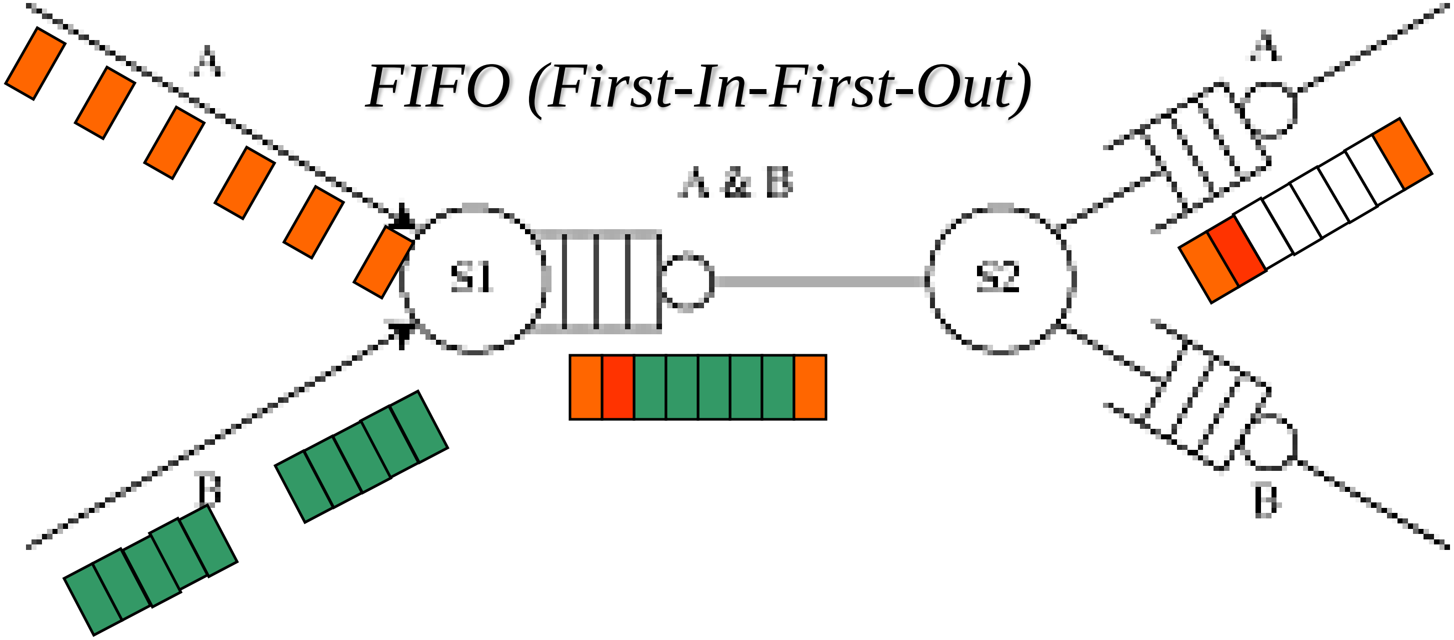
“≥” for Rate - Allocation
Flow *1*
≥r*1*
Flow *2*
≥r*2*
Service Rate
c ≥ r*1*+r*2* +. . .+r*m*
Flow *m*
≥r*m*
“=” for Rate - Controlled

Fair-Share Scheduling (contd ..)

- In the rate-allocation service discipline, a queue may be served at a higher rate than the minimum service rate as long as the guarantees made to other services do not suffer
- In the rate-controlled service discipline, the mechanism does not serve any queue at a rate higher than its assigned service rate under any circumstance
- These rate-based schemes are also classified as *work conserving* or *non-work conserving*
- A work conserving scheduler is never idle when there are cells to send in the queues
- Since rate-controlled schedulers do not serve a queue with more than its allocated rate, these are classified as non-work conserving while the rate-allocation schedulers are work conserving

Work conserving vs. non-work-conserving

- Work conserving discipline is never idle when packets await service
- Why bother with non-work conserving?



FIFO (First-In-First-Out)

Evenly spaced cells clump and get bursty
Needing larger buffers

Non-work-conserving disciplines

- Key conceptual idea: delay packet till *eligible*
- Reduces delay-jitter => fewer buffers in network
- How to choose eligibility time?
  - rate-jitter regulator
    - bounds maximum outgoing rate
  - delay-jitter regulator
    - compensates for variable delay at previous hop

Do we need non-work-conservation?

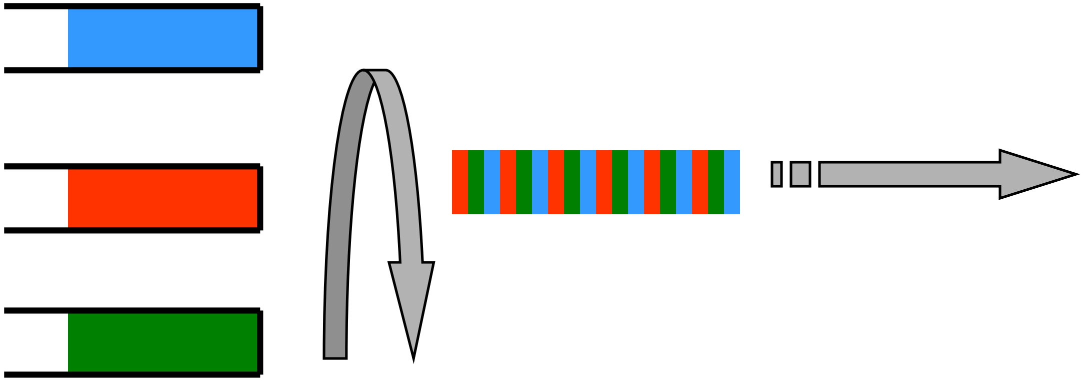
- Can remove delay-jitter at an endpoint instead
  - but also reduces size of switch buffers…
- Increases mean delay
  - not a problem for *playback* applications
- Wastes bandwidth
  - can serve best-effort packets instead
- Always punishes a misbehaving source
  - can’t have it both ways
- Bottom line: not too bad, implementation cost may be the biggest problem

Fair-Share Scheduling (contd ..)

- An ideal fair-share scheduler is the one, which employs **Processor Sharing (PS)** among the traffic flows.
  - The processor divides the link capacity equally among the contending connections *by serving the connections infinitesimally*
- When connections are given different weights and the link capacity is divided not as equal share, but in proportion to the weight assigned, then the processor sharing is referred to as **Generalized Processor Sharing (GPS).**
- It is very difficult to implement the PS or GPS, as it assumes that the traffic is infinitely divisible and all connections with non-empty queues can be served simultaneously
- In reality, a packet can only be served in its entirety (i.e., cannot be split). Therefore, **fair queuing** or the **weighted fair queuing** can be thought of as a practical implementation (or approximation) of the PS or GPS

Generalized Processor Sharing (GPS)

- Intuitively GPS serves packets as if they are in separate logical queues, visiting each non-empty queue in turn and serving an infinitesimally small amount of data from each queue
- In finite time interval, GPS visits every logical queue at least once
- Connections can be associated with service weights
- If there is no data in a queue, the scheduler skips to next non-empty queue
  - *Results in max-min fair allocation*



Precise definition of GPS

- A connection is termed "backlogged" if it has data in its queue
- Let *N* connections are served by a GPS server
- Let each of the connections are given weights $\phi(1), \phi(2), \ldots, \phi(N)$
- Let the amount of data the server serves for a given connection *i* in the time interval $[\tau, t]$ be $S(i,\tau,t)$
- Then for any connection *i* back logged in $[\tau, t]$ and for any other connection *j* (backlogged or not) we have

$$\frac{S(i,\tau,t)}{S(j,\tau,t)} \geq \frac{\phi(i)}{\phi(j)}$$

- Intuitively a *non backlogged connection* is already getting as much service as it could possibly use
- A GPS server ensures that *backlogged connections* share the remaining bandwidth in proportion to their weight, achieving max-min fair allocation

Implementing WFQ (Method 1)

- Assign a *service-deadline* (or *time-stamp*, or *virtual finishing times,* or *finish number* or *service tag*) for each flow and serve them in the ***increased order of the deadline (sorting)***. If there is a tie between the flows, the flows are re-ordered randomly. Assuming a unit link capacity, the service-deadline F_i^j assigned to the packets of each flow *j* at instant *i* has the following form:

$$F_i^j = \max\left\{F_{i-1}^j, v_i^j\right\} + \left(1/r^j\right)$$

$$F_0^j = 0$$

- Here, r^j is the minimum guaranteed bandwidth for connection *j* and v_i^j is the virtual time (or the round number) for connection *j* at instant *i*. The main difference between various algorithms is the way the virtual time is computed.

Implementing WFQ (Method 2)

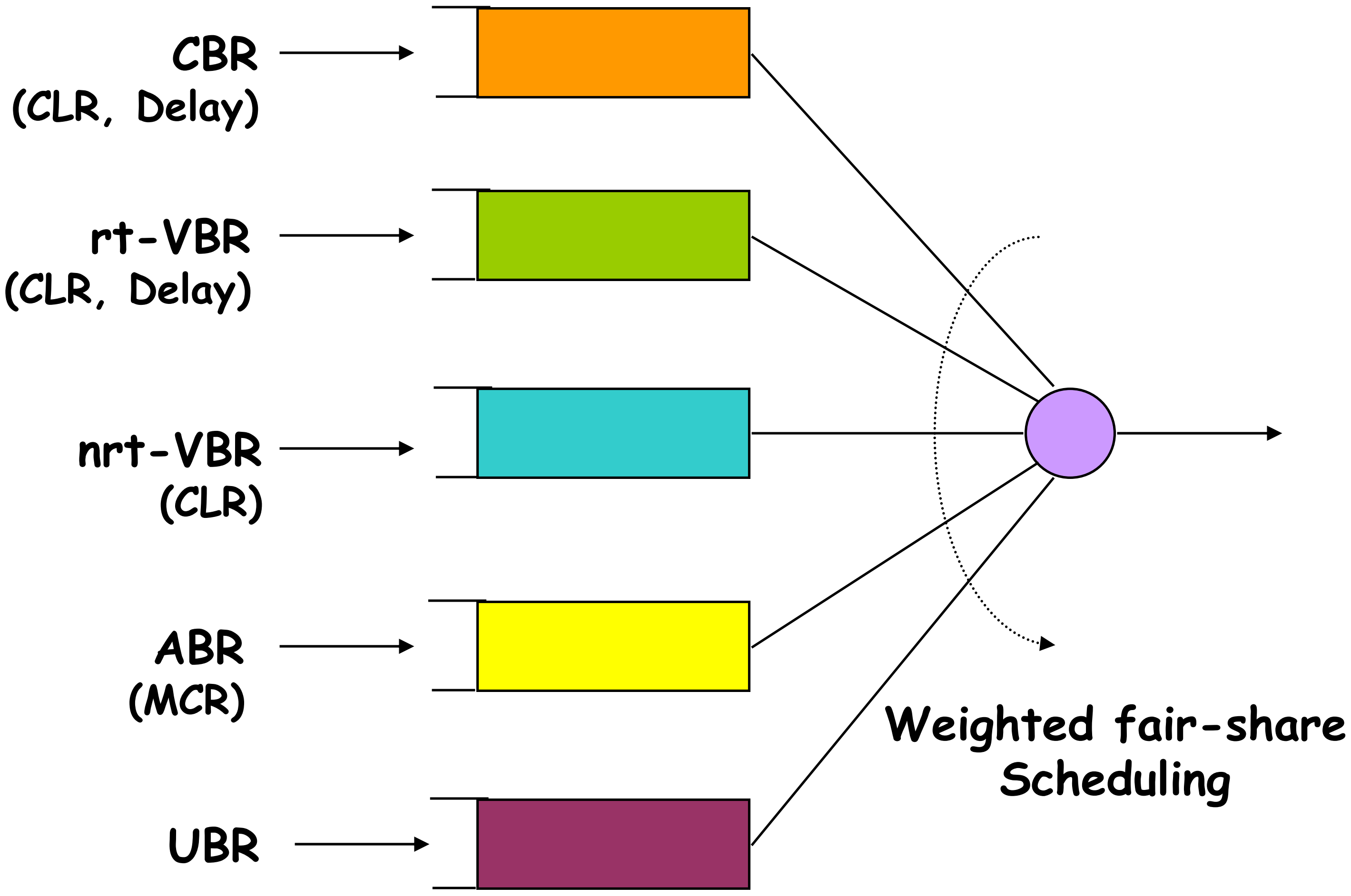
- Serve the queues in a round-robin fashion using *frames* or *cycles*.
- In each cycle, all the queues are given a transmission opportunity to transmit a cell or packet.
- Queues with larger weights can be given multiple transmission opportunities in each cycle.
- If all the queues have the same weight, then it is a round robin (RR) system otherwise it is weighted round-robin (WRR).
- This behavior is similar to a TDM system, except that if a queue is given a transmission opportunity and if the queue does not have any cell to transmit, the scheme does not waste the bandwidth.
- Instead, the scheduler examines rest of the queues for any possible cell transmission.
- With *N* backlogged flows, a round robin discipline assigns a bandwidth of *Link Capacity/N* to each flow, while for each flow with weights w_i, WRR scheme assigns a bandwidth of:

$$w_i.Link\,Capacity \Big/ \left(\sum_{1\le i\le N} w_i\right)$$

Major aspects

- Bandwidth guarantees for the traffic flows and Fairness
- Bandwidth granularity
- Assignment of weights
- Computation of cycles

Multi-Service ATM Network

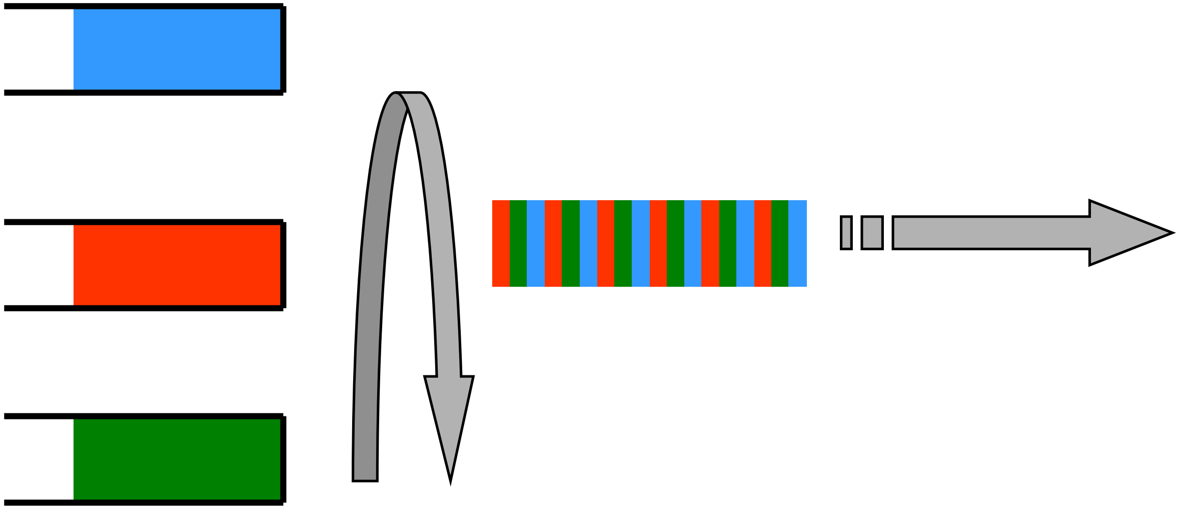


Outline

- What is scheduling
- Why we need it
- Requirements of a scheduling discipline
- Fundamental choices
- Scheduling best effort connections
- Scheduling guaranteed-service connections
- Packet drop strategies

Scheduling best-effort connections

- Main requirement is *fairness*
- Achievable using *Generalized processor sharing (GPS)*
  - Visit each non-empty queue in turn
  - Serve infinitesimal from each
  - Why is this fair?
  - How can we give weights to connections?



More on GPS

- GPS is unimplementable!
  - we cannot serve infinitesimals, only packets
- No *packet discipline* can be as fair as GPS
  - while a packet is being served, we are unfair to others
- Degree of unfairness can be bounded
- **Define**: *work(i,a,b)* = # bits transmitted for connection *i* in time [a,b]
- *Absolute* fairness bound for discipline S
  - Max (work_GPS(i,a,b) - work_S(i, a,b))
- *Relative* fairness bound for discipline S
  - Max (work_S(i,a,b) - work_S(j,a,b))

What next?

- We can’t implement GPS
- So, lets see how to emulate it
- We want to be as fair as possible
- But also have an efficient implementation
- Two Methods
  - Frames or Cycles based (RR, WRR, DRR)
  - Time Stamp based (WFQ)

Weighted round robin (WRR)

- Serve a packet from each non-empty queue in turn instead of an infinitesimal amount
- Round Robin (RR) approximates GPS reasonably well when connections have equal weights and have the same packet size
- Unfair if packets are of different length or weights are not equal
- ***Different weights, fixed packet size***
  - serve more than one packet per visit, after normalizing to obtain ***integer weights***
- ***Different weights, variable size packets***
  - normalize weights by mean packet size
    - e.g. weights {0.5, 0.75, 1.0}, mean packet sizes {50, 500, 1500}
    - normalize weights: {0.5/50, 0.75/500, 1.0/1500} = { 0.01, 0.0015, 0.000666}, normalize again {60, 9, 4} = serves 3000 bytes from A, 4500 bytes from B, 6000 bytes from C

Problems with Weighted Round Robin

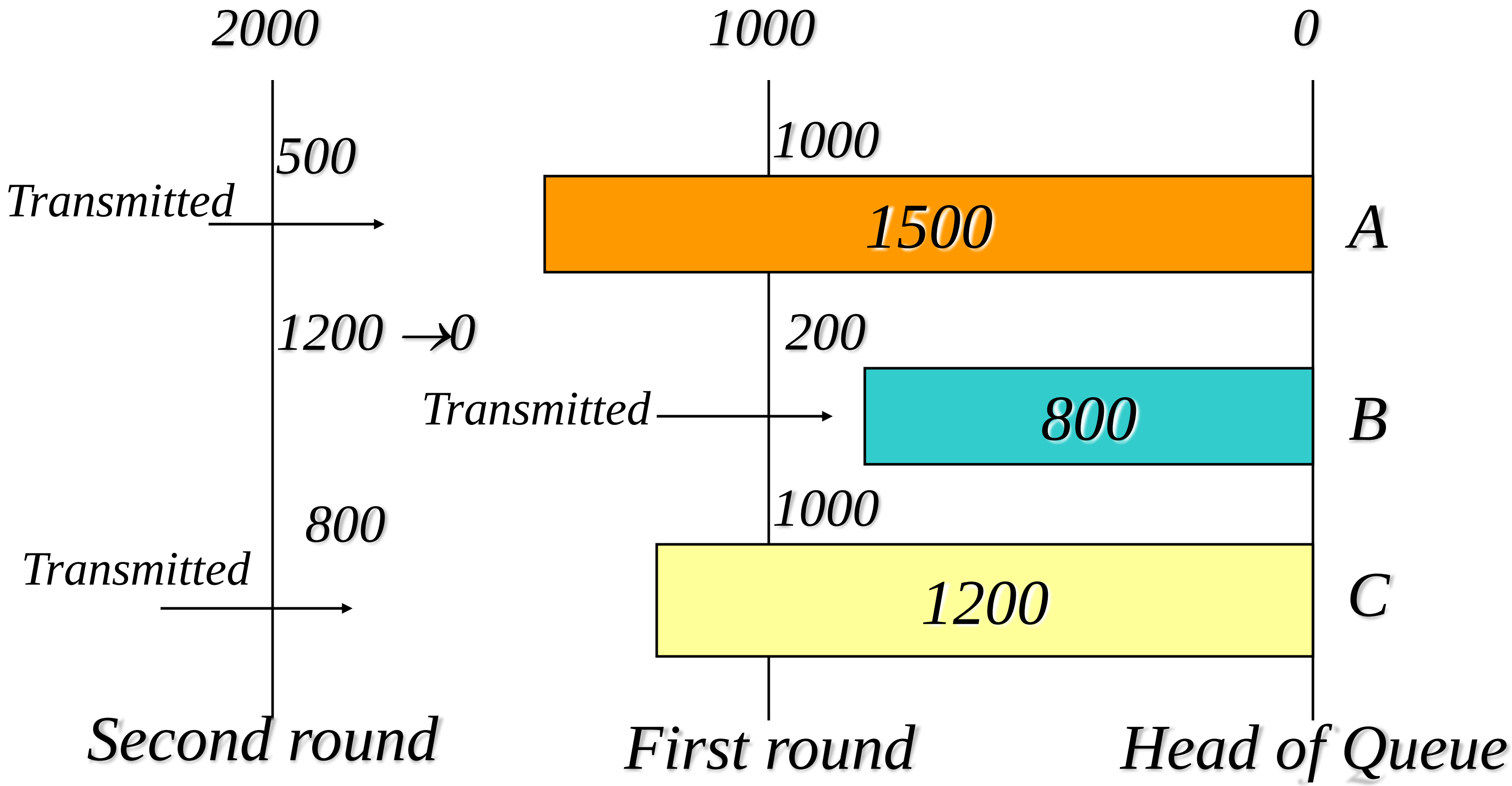
- With variable size packets and different weights, need to know mean packet size in advance
- Can be fair only over time scales longer than a round time (is the time to go through a cycle of polling)
- For shorter time scale, some connections may get more service than others
- If a connection has a small weight, or the number of connections is large, this may lead to long periods of unfairness
- E.g.:T3 trunk with 500 connections, each connection has mean packet length 500 bytes, 250 with weight 1, 250 with weight 10
  - Each packet takes 500 * 8/45 Mbps = 88.8 microseconds
  - Round time =2750 * 88.8 = 244.2 ms (i.e., 250*10+250*1)*88.8
  - Over a time smaller than this, some connections get more service than others (unfair)

Deficit Round Robin (DRR)

- Modifies weighted round robin scheduling to handle variable packet sizes
- This scheme maintains a “deficit counter” (*DCi*) for each traffic flow. The deficit counter is reset to zero whenever the queue falls empty.
- The deficit counter is initialized to the weight of connection (*Qi*) when the queue becomes backlogged.
- The weight is the number of bits a flow is allowed to send in a given round.
- Since each packet is of different length, a deficit can be built up for each traffic flow. The current deficit is set equal to the previous deficit if any less the number of bits sent in the current packet, i.e., *DCi = DCi +(Qi-bytes_senti)*. The flow can take advantage of the available deficit in its next round.

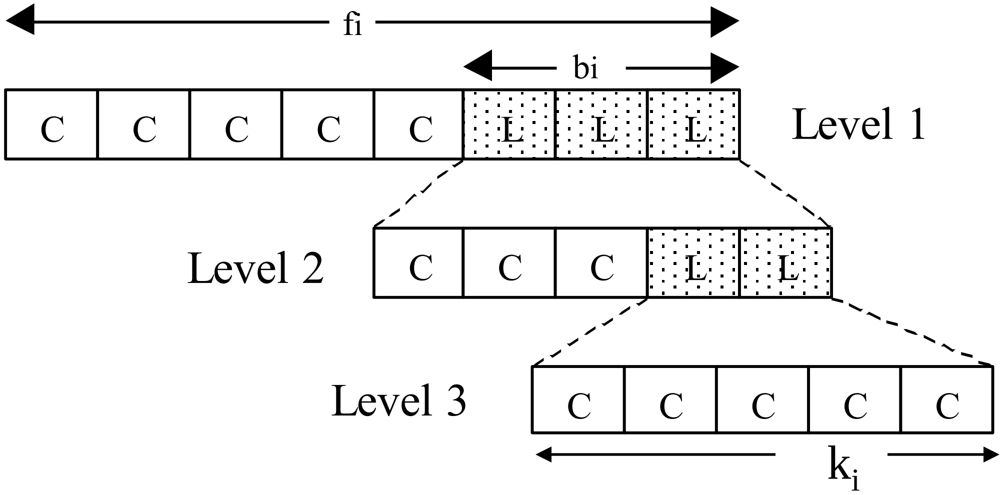
DRR Example

- Equal weight, variable packet sizes, three connections (A,B,C)
- Weight = Quantum =Q_i = 1000 bytes



Hierarchical Round Robin (HRR)

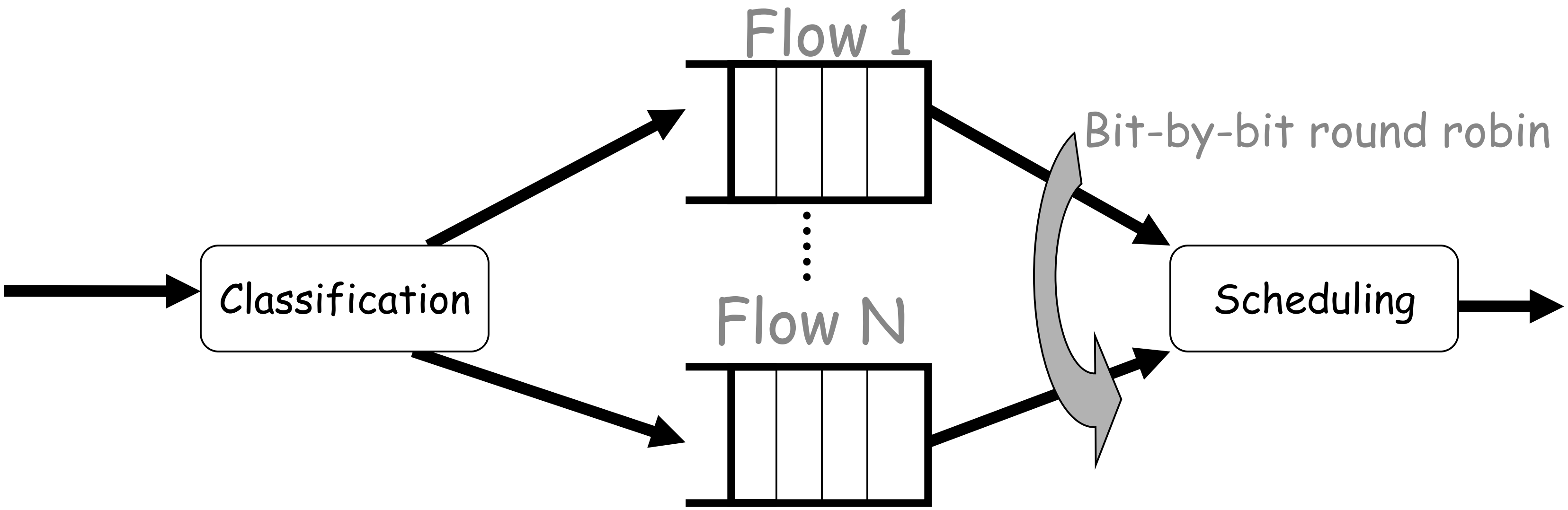
- The Hierarchical Round Robin (HRR) uses a framing strategy
- The frames are divided into various levels, each level can consist of different frame length
- At each level, slots are assigned to connections directly on that level or reserved for usage to lower levels



$$Connection \quad Bandwidth = \left(\sum_{i=1}^{L} k_i \frac{b_1 b_2 \ldots b_i}{f_1 f_2 \ldots f_i} \right) \times Link\,Capacity$$

Fair Queueing

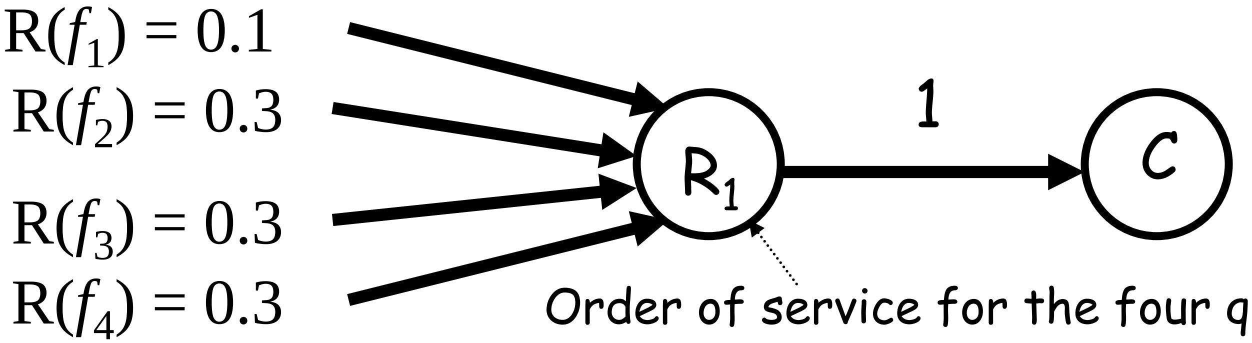
1. Packets belonging to a flow are placed in a FIFO. This is called "per-flow queueing".
2. FIFOs are scheduled one bit at a time, in a round-robin fashion (*approximation to GPS*).
3. This is called *Bit-by-Bit Fair Queueing*.



Courtesy Nick McKeown@Stanford

Weighted Bit-by-Bit Fair Queueing

Likewise, flows can be allocated different rates by servicing a *different number of bits* for each flow during each round.



Order of service for the four queues:
… $f_1, f_2, f_2, f_2, f_3, f_3, f_3, f_4, f_4, f_4, f_1$,…

Also called "Generalized Processor Sharing (GPS)"

Courtesy Nick McKeown@Stanford

Packetized Weighted Fair Queueing (WFQ)

Problem: We need to serve a whole packet at a time.

Solution:

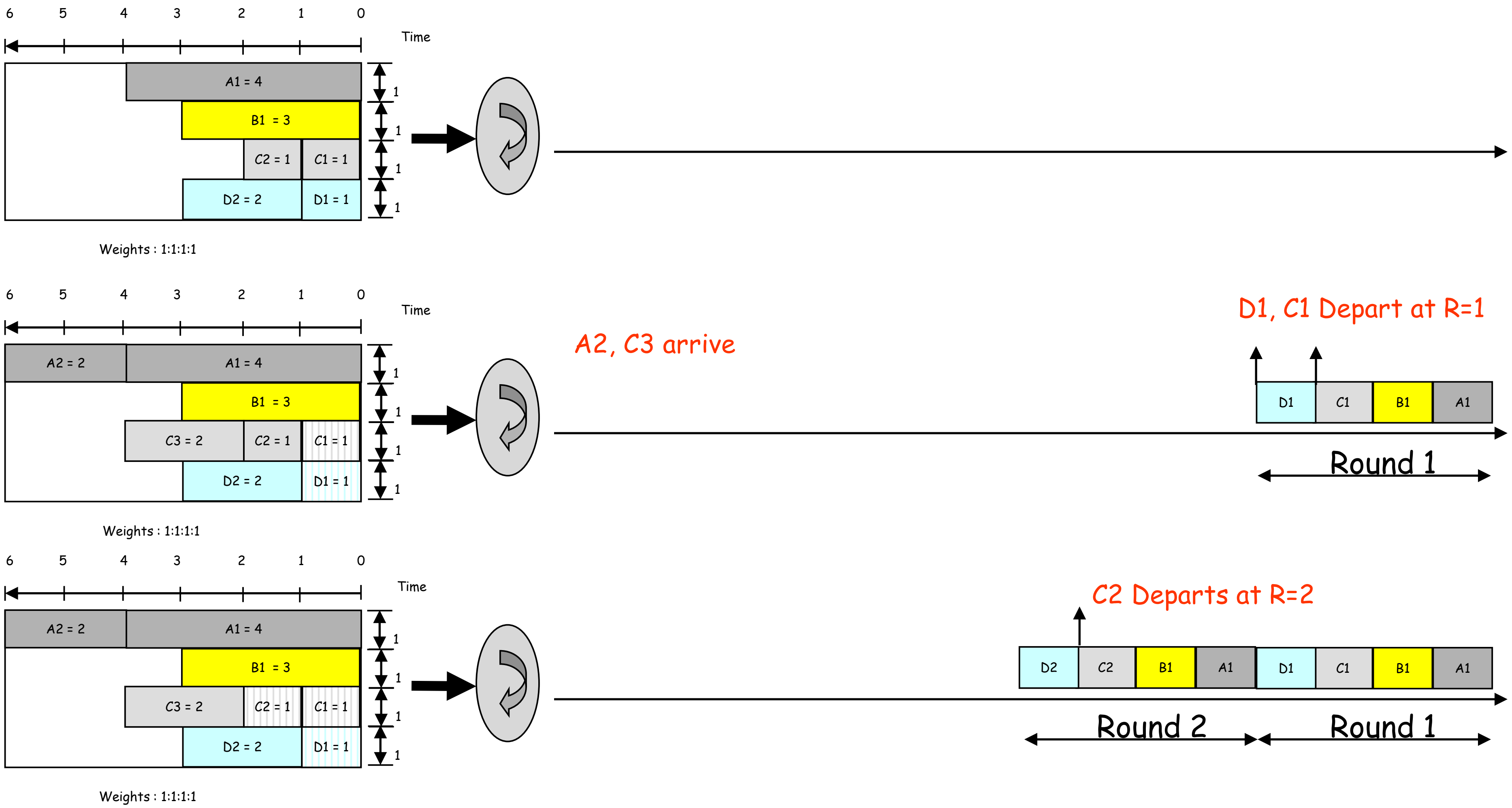
1. Determine what time a packet, p, would complete if we served flows bit-by-bit. Call this the packet's finishing time, F_p.
2. Serve packets in the order of increasing finishing time.

Also called "Packetized Generalized Processor Sharing (PGPS)"

Courtesy Nick McKeown@Stanford

Understanding bit by bit WFQ

4 queues, sharing 4 bits/sec of bandwidth, Equal Weights



A2, C3 arrive

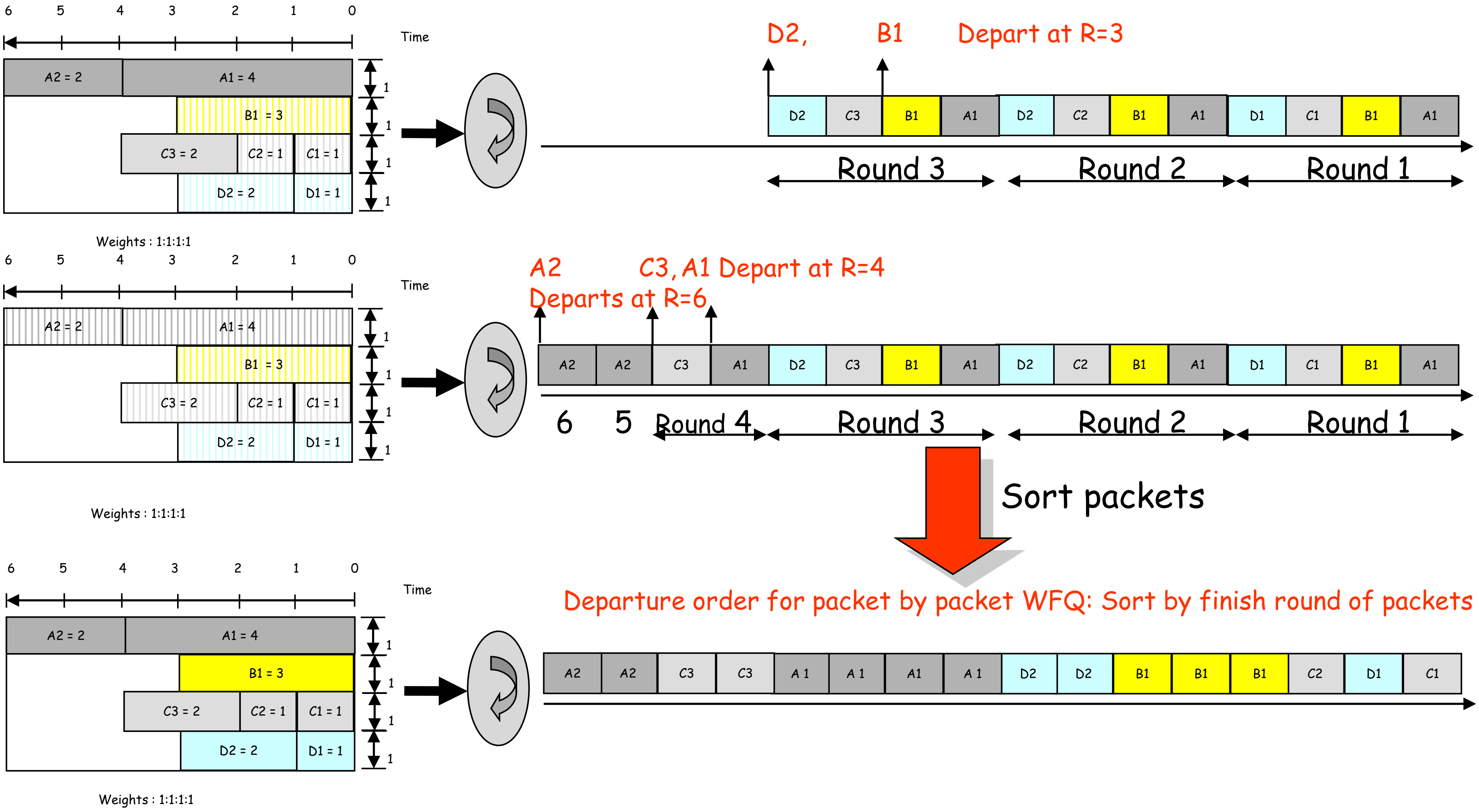
D1, C1 Depart at R=1

C2 Departs at R=2

Courtesy Nick McKeown@Stanford

Understanding bit by bit WFQ

4 queues, sharing 4 bits/sec of bandwidth, Equal Weights


6 5 4 3 2 1 0
Time
A2 = 2
A1 = 4
B1 = 3
C3 = 2
C2 = 1
C1 = 1
D2 = 2
D1 = 1
1
Weights : 1:1:1:1
D2 C3 B1 A1 D2 C2 B1 A1 D1 C1 B1 A1
Round 3
Round 2
Round 1
A2 A2 C3 A1 D2 C3 B1 A1 D2 C2 B1 A1 D1 C1 B1 A1
6
5
Round 4
A2 A2 C3 C3 A 1 A 1 A1 A 1 D2 D2 B1 B1 B1 C2 D1 C1

D2, B1 Depart at R=3

A2 Departs at R=6

C3, A1 Depart at R=4

Sort packets

Departure order for packet by packet WFQ: Sort by finish round of packets

Courtesy Nick McKeown@Stanford

Weighted Fair Queueing (WFQ)

- Deals better with variable size packets and weights
- GPS is fairest discipline
- Find the *finish time* of a packet for a connection *i*, *had we been doing GPS*
  - ◆ *F(i,k,t)=max{F(i,k-1,t), R(t)}+P(i,k,t)*
  - ◆ *F(i,k,t) is the finish number of k packet at time t*
  - ◆ *R(t) is the round number*
  - ◆ *P(i,k,t) is the size of* k^{th} *packet arrives at time t*
- Then serve packets in order of their finish times
- We will first study how to compute the round number when all weights are equal

WFQ: first cut

- Suppose, in each *round,* the server served one bit from each active connection (model it as bit-by-bit round robin)
- *Round number* (or Virtual Time) is the number of rounds already completed
  - can be fractional
  - Each round takes a variable amount of time depending upon how many connections are active; i.e., *the length of round is proportional to the number of active connections and round number increases at a rate inversely proportional to the number of active connections*
- If a packet of length *p* arrives to an empty queue when the round number is *R*, it will complete service when the round number is *R + p => finish number* is *R + p*
  - independent of the number of other connections!
- If a packet arrives to a non-empty queue, and the previous packet has a finish number of *f*, then the packet's finish number is *f+p*
- Serve packets in order of finish numbers

WFQ continued

- To sum up, assuming we know the current round number *R*
- Finish number of packet of length *p*
  - if arriving to active connection = previous finish number + *p*
  - if arriving to an inactive connection = *R* + *p*
- (How should we deal with weights?)
- To implement, we need to know two things:
  - is connection active?
    - ☞ A connection is *active* if the last packet served from it, or in its queue, has a finish number greater than the current round number
  - if not, what is the current round number?

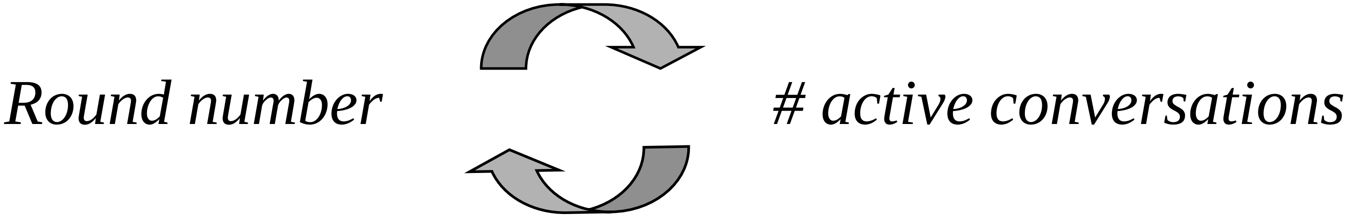
WFQ: computing the round number

- Naively: round number = number of rounds of service completed so far
  - what if a server has not served all connections in a round?
  - what if new conversations join in halfway through a round?
- *Redefine* round number as a real-valued variable that increases at a rate inversely proportional to the number of currently active connections
  - this takes care of both problems
  - *$F(i,k,t)=\max\{F(i,k-1,t), R(t)\}+P(i,k,t)/\phi(i)$ where $\phi(i)$ is the weight of the connections*
  - *Round number increases ($\partial R/\partial t$) with a slope $1/\sum\phi(i)$ where the sum is over all active connections (N_{ac})*
- With this change, WFQ emulates GPS instead of bit-by-bit RR

Round Number

- Consider the following example. Suppose that a packet of size 100 bits arrives at time 0 on conversation A. During [0,50), since *Nac* = 1, and $\partial R(t)/\partial t = 1/\,N\,ac$, $R(50\ ) = 50$.
- Suppose that a packet of size 100 bits arrives at conversation B at time 50. It will be assigned a finish number of 150 (=50 + 100).
- At time 100, P_0^A has finished service. However, in the time interval [50, 100), *Nac* = 2, and so $R(\ 100\ ) = 75$. Since $F^A = 100$, A is still active, and *Nac* stays at 2.
- At $t = 200$, P_0^B completes service.
- What should $R(200\ )$ be? The number of conversations went down to 1 when $R(t) = 100$. This must have happened at $t = 150$, since $R(100\ ) = 75$, and $\partial R(t)/\partial t = 1/2$. Thus, $R(200\ ) = 100 + 50 = 150$.
- Note that each conversation departure speeds up the $R(t)$, and this makes it more likely that some other conversation has become inactive. Thus, it is necessary to do an *iterated deletion* of conversations to compute $R(t)$

Problem: iterated deletion



Round number

# active conversations

- Main problem in computing a packet’s finish number is determining the current round number of the simulated GPS Server
- A sever recomputes round number on each packet arrival
- At any recomputation, the number of conversations can go up at most by one, but can go down to zero
- => overestimation
- Trick
  - use previous count to compute round number
  - if this makes some conversation inactive, recompute
  - repeat until no conversations become inactive

WFQ implementation

- On packet arrival:
  - use source + destination address (or VCI) to classify it and look up finish number of last packet served (or waiting to be served)
  - recompute round number
  - compute finish number
  - insert in priority queue sorted by finish numbers
  - if no space, drop the packet with largest finish number
- On service completion
  - select the packet with the lowest finish number

Analysis

- Unweighted case:
  - Pmax(i) size of largest possible packet sent on connection i
  - Pmax size of largest packet in the network
  - g(i) is the smallest of rates allocated to connection along its path (Network max-min allocation)
  - G(i, τ, t) service received by connection i using GPS
  - It can be shown that WFQ can lag GPS at most Pmax/g(i)
    - $S(i, \tau, t)/g(i) \geq G(i, \tau, t)/g(i) - Pmax/g(i)$
- However, under WFQ, a connection can receive more service than GPS => absolute fairness bound > *Pmax/g(i)*
- To reduce bound, choose smallest finish number only among packets that have started service in the corresponding GPS system (Worst case fair WFQ, WF^{2}Q)
  - requires a regulator to determine eligible packets

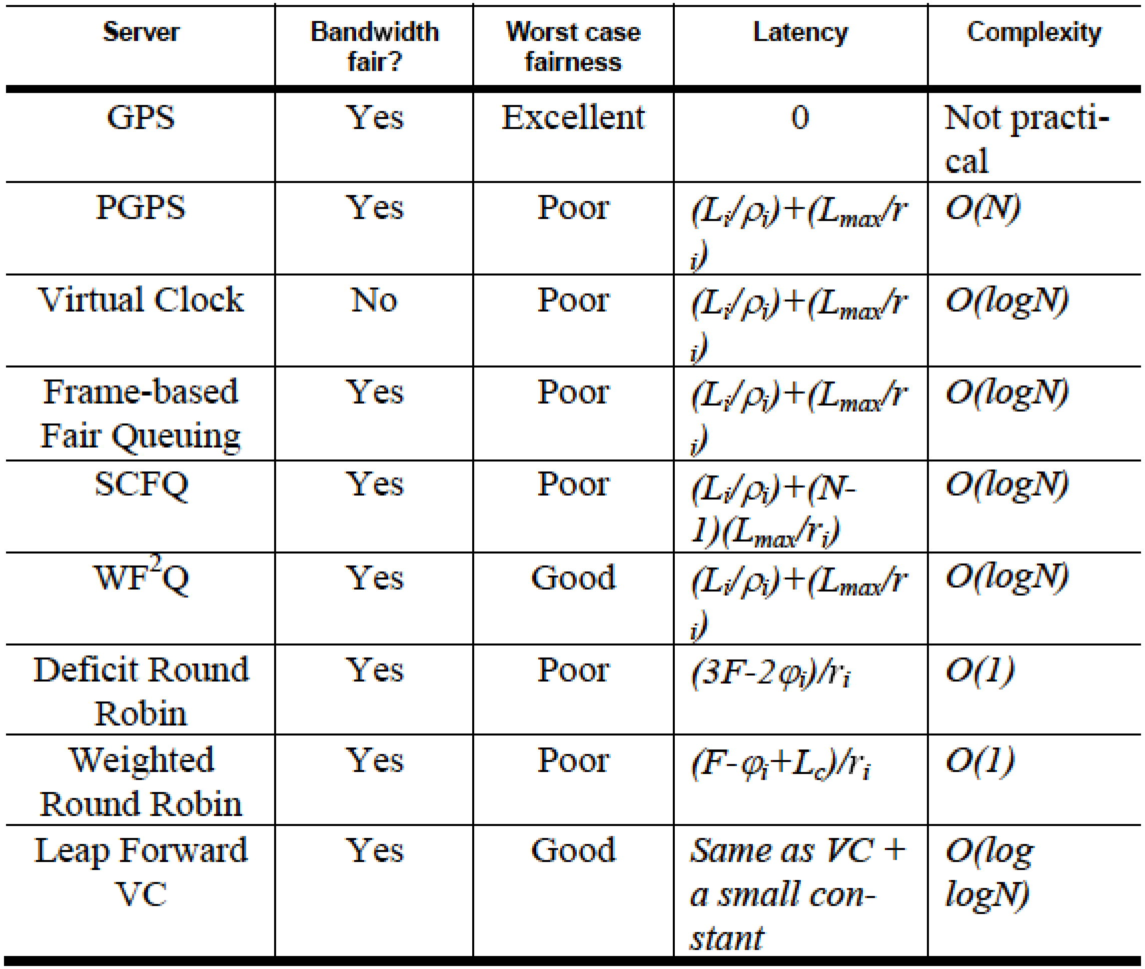
WFQ Evaluation

- Pros
  - like GPS, it provides protection
  - can obtain worst-case end-to-end delay bound
  - gives users incentive to use intelligent flow control (and also provides rate information implicitly)
- Cons
  - needs per-connection state
  - iterated deletion is complicated
  - requires a priority queue

Variants of WFQ

- Self-Clocked Fair Queuing (SCFQ)
  - When a packet arrives to empty queue, instead of using the round number to compute its finish number, it uses the finish number of the packet *currently in service*
  - *$F(i,k,t)=\max\{F(i,k-1,t), CF\}+P(i,k,t)/\phi(i)$ where CF is the finish time of packet currently in service*
  - Can be unfair
- Start-time Fair Queueing (SFQ)
  - Compute both finish and start number of each arriving packet
  - Start number of a packet arriving at an inactive connection is set to the current round number
  - Otherwise it is set to finish number of previous packet
  - Service is in the order of increasing start times
  - Worst case delay is lower than SCFQ

Comparison of Servers



Server	Bandwidth fair?	Worst case fairness	Latency	Complexity
GPS	Yes	Excellent	0	Not practical
PGPS	Yes	Poor	$(L_i/\rho_i)+(L_{max}/r_i)$	$O(N)$
Virtual Clock	No	Poor	$(L_i/\rho_i)+(L_{max}/r_i)$	$O(logN)$
Frame-based Fair Queuing	Yes	Poor	$(L_i/\rho_i)+(L_{max}/r_i)$	$O(logN)$
SCFQ	Yes	Poor	$(L_i/\rho_i)+(N-1)(L_{max}/r_i)$	$O(logN)$
WF^2Q	Yes	Good	$(L_i/\rho_i)+(L_{max}/r_i)$	$O(logN)$
Deficit Round Robin	Yes	Poor	$(3F-2\varphi_i)/r_i$	$O(1)$
Weighted Round Robin	Yes	Poor	$(F-\varphi_i+L_c)/r_i$	$O(1)$
Leap Forward VC	Yes	Good	*Same as VC + a small constant*	$O(log\ logN)$

Outline

- What is scheduling
- Why we need it
- Requirements of a scheduling discipline
- Fundamental choices
- Scheduling best effort connections
- Scheduling guaranteed-service connections
- Packet drop strategies

Scheduling guaranteed-service connections

- With best-effort connections, goal is fairness
- With guaranteed-service connections
  - what performance guarantees are achievable?
  - how easy is admission control?
- We now study some scheduling disciplines that provide performance guarantees

WFQ and Performance Guarantees

- Turns out that WFQ also provides performance guarantees
- Bandwidth bound
  - ratio of weights * link capacity
  - e.g. connections with weights 1, 2, 7; link capacity 10
  - connections get at least 1, 2, 7 units of b/w each
- End-to-end delay bound
  - *Key is to assume* that the connection doesn't send 'too much' (otherwise its packets will be stuck in queues)
  - more precisely, connection should be *leaky-bucket* regulated or Linear Bounded Arrival Processes (LBAP).
  - The linear function is characterized by two parameters σ and ρ so that the # bits sent in time $[t_1, t_2] <= \rho\ (t_2 - t_1) + \sigma$
  - ρ is the long term average rate and σ is the longest burst

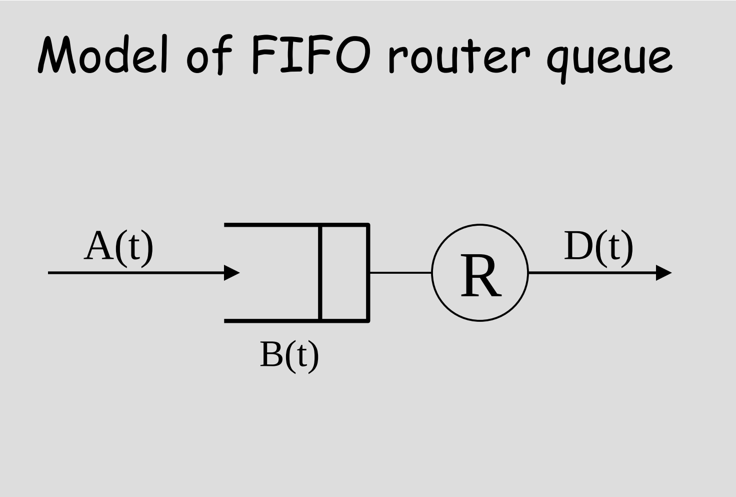
So how can we control the delay of packets?

Assume continuous time, bit-by-bit flows for a moment…

1. Let’s say we know the arrival process, $A_f(t)$, of flow f to a router.
2. Let’s say we know the rate, $R(f)$ that is allocated to flow f.
3. Then, in the usual way, we can determine the delay of packets in f, and the buffer occupancy.

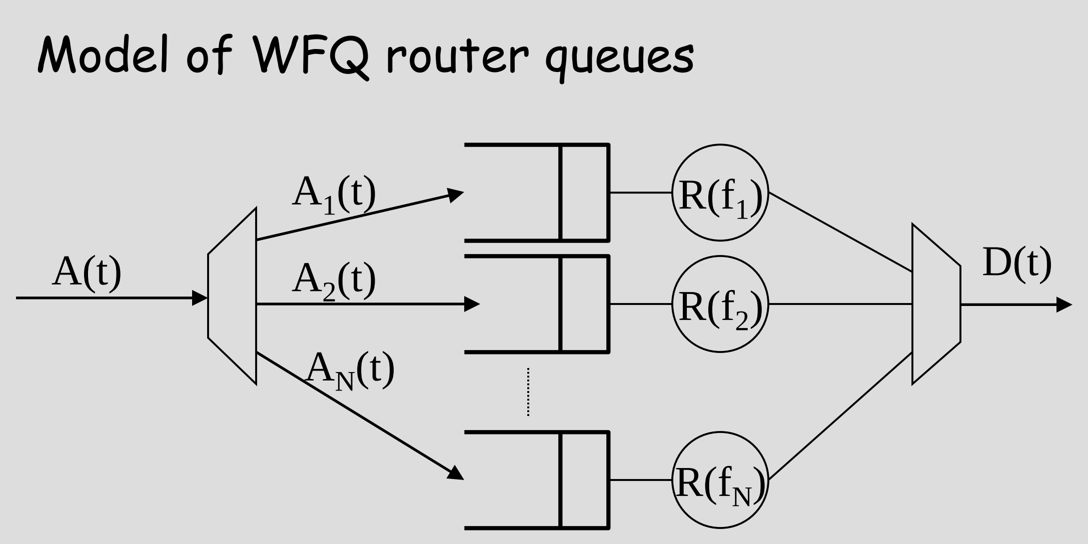
Courtesy Nick McKeown@Stanford

Router Without QoS:



Model of FIFO router queue

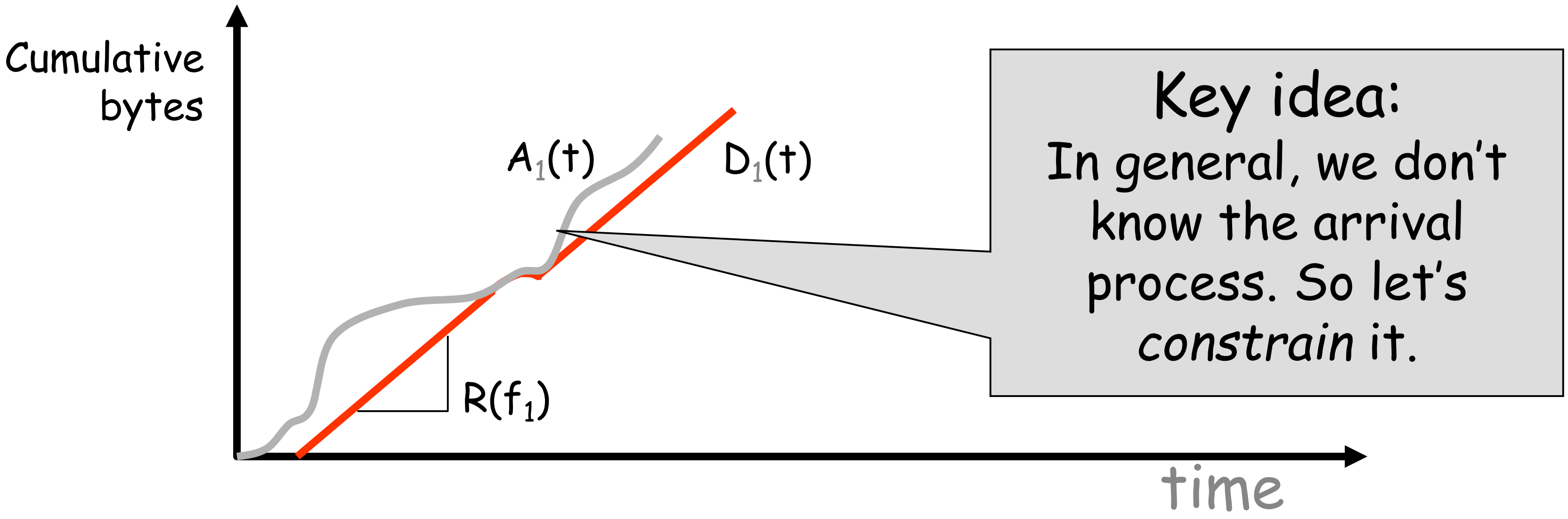
Router With QoS:



Model of WFQ router queues

Courtesy Nick McKeown@Stanford

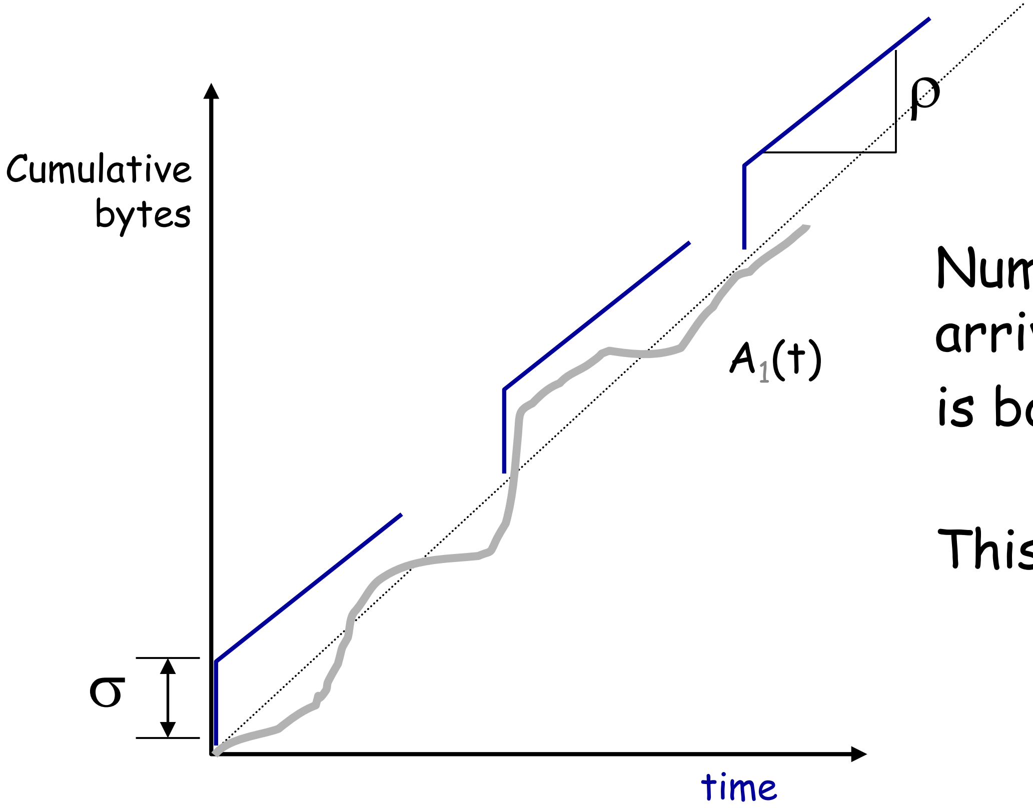
How to bound packet delay?



Key idea:
In general, we don't know the arrival process. So let's *constrain* it.

Courtesy Nick McKeown@Stanford

Let’s say we can bound the arrival process

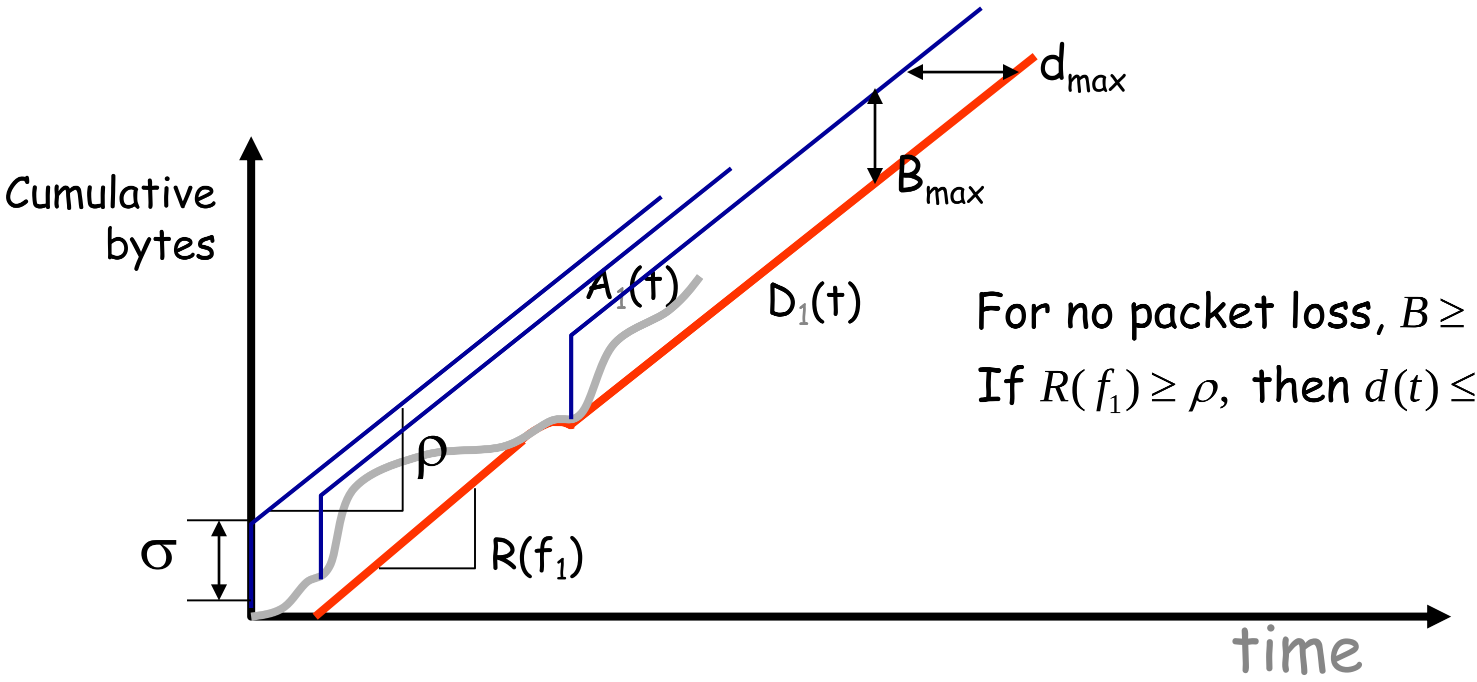


Number of bytes that can arrive in any period of length t is bounded by: $\sigma + \rho t$

This is called “(σ,ρ) regulation”

Courtesy Nick McKeown@Stanford

(σ,ρ) Constrained Arrivals and Minimum Service Rate



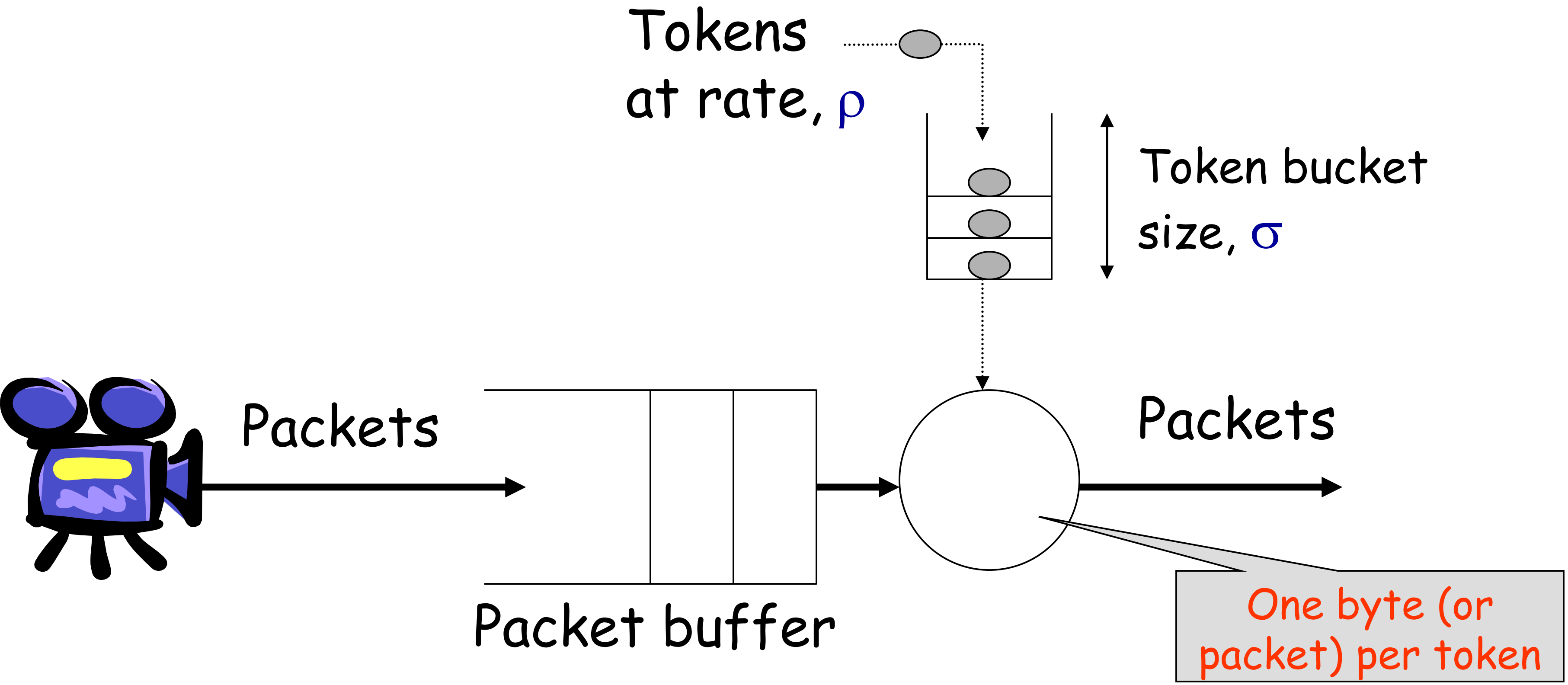
For no packet loss, $B \geq \sigma$.

If $R(f_1) \geq \rho$, then $d(t) \leq \sigma / R(f_1)$.

Theorem [Parekh,Gallager '93]: If flows are leaky-bucket constrained, and routers use WFQ, then end-to-end delay guarantees are possible.

Courtesy Nick McKeown@Stanford

The leaky bucket “(σ,ρ)” regulator



Tokens at rate, ρ

Token bucket size, σ

Packets

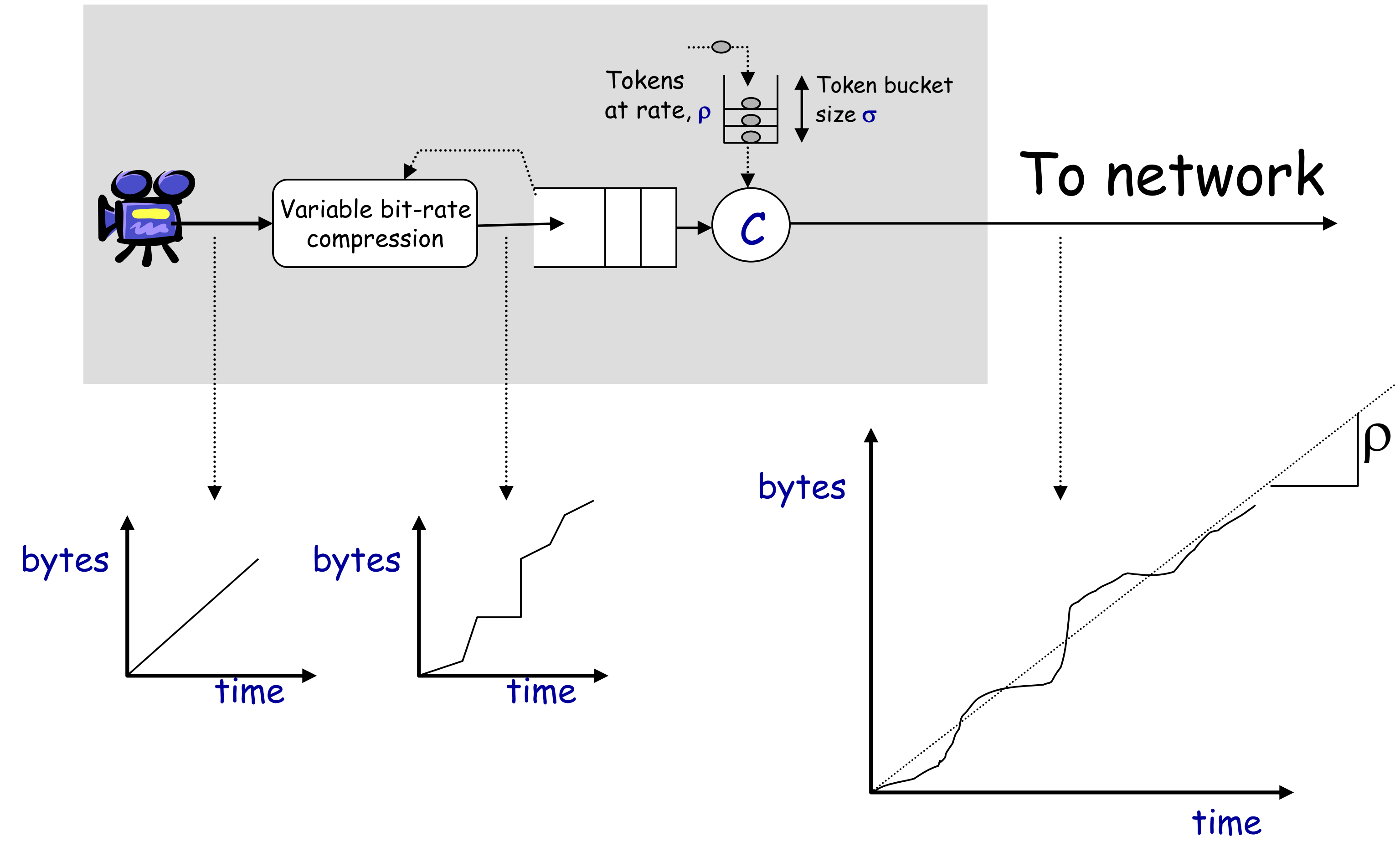
Packet buffer

Packets

One byte (or packet) per token

Courtesy Nick McKeown@Stanford

How the user/flow can conform to the (σ,ρ) regulation



Courtesy Nick McKeown@Stanford

Providing Delay Guarantees: Summary

1. Before starting a new flow, source asks network for end-to-end delay guarantee.
2. Source *negotiates* (σ,ρ) values with each router along the path so at to achieve end-to-end delay guarantee. Routers perform *admission control* to check whether they have sufficient resources (link data rate and buffers).
3. Each router along path *reserves* resources.
4. Flow starts, and source starts sending packets according to agreed (σ,ρ) values.
5. Router determines which flow each arriving packet belongs to, and puts it in the right queue.
6. Router serves queues, using WFQ, so as to bound packet delay through the router.

Courtesy Nick McKeown@Stanford

Parekh-Gallager theorem

- Let a connection be allocated weights at each WFQ scheduler along its path, so that the least bandwidth that is allocated is *g*
- Let it be leaky-bucket regulated such that # bits sent in time $[t_1, t_2] <= \rho (t_2 - t_1) + \sigma$
- Let the connection pass through *K* schedulers, where the *k*th scheduler has a rate *r(k)*
- Let the largest packet allowed in the network be *P*

$$end\_to\_end\_delay \le \sigma / g + \sum_{k=1}^{K-1} P / g + \sum_{k=1}^{K} P / r(k)$$

Significance

- Theorem shows that WFQ can provide end-to-end delay bounds
- So WFQ provides both fairness and performance guarantees
- Bound holds regardless of cross traffic behavior
- Can be generalized for networks where schedulers are variants of WFQ, and the link service rate changes over time

Problems

- To get a delay bound, need to pick *g*
  - the lower the delay bounds, the larger *g* needs to be
  - large *g* => exclusion of more competitors from link
  - *g* can be very large, in some cases 80 times the peak rate!
- Sources must be leaky-bucket regulated
  - but choosing leaky-bucket parameters is problematic
- WFQ couples delay and bandwidth allocations
  - low delay requires allocating more bandwidth
  - wastes bandwidth for low-bandwidth low-delay sources

Steps Involved in Providing Delay Guarantees

- Per-session
  - Call setup, call admission and resource reservation

    "Can the network accept the call and provide the QoS?"
- Per-packet
  - Packet Classification: "What flow does this packet belong to, and when should I send it?"
  - Shaping: "Am I keeping my side of the contract?"
  - Policing: "Did the user keep his/her side of the contract?"
  - Packet Scheduling: "Sending the packet at the right time."

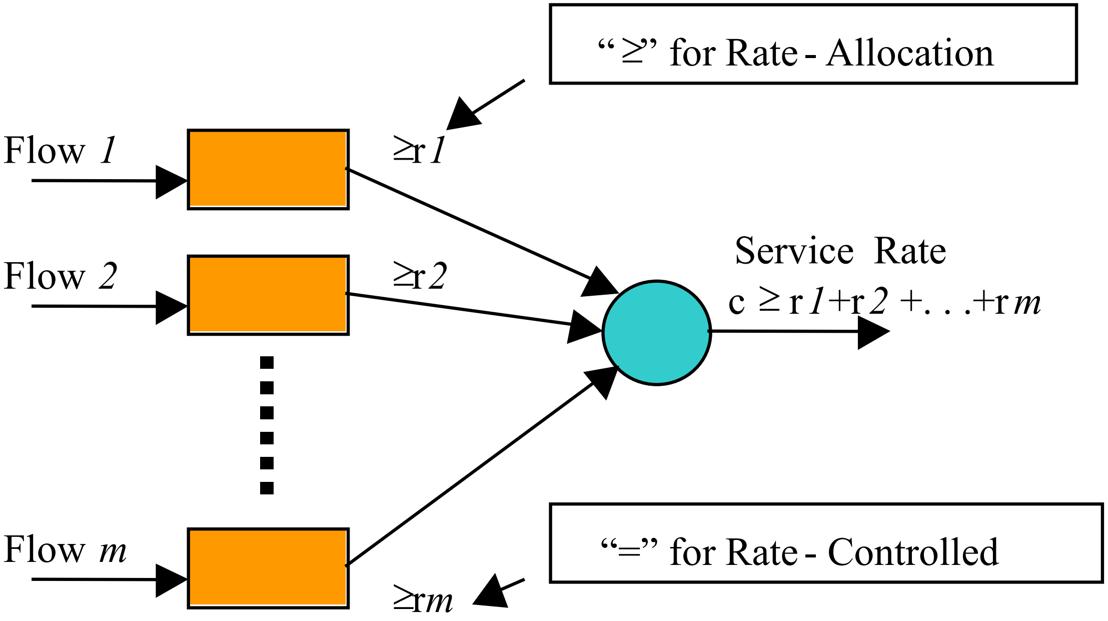
Courtesy Nick McKeown@Stanford

Delay-Earliest Due Date

- Earliest-due-date: packet with earliest deadline selected
- Delay-EDD prescribes how to assign deadlines to packets
- A source is required to send slower than its *peak rate*
- Bandwidth at scheduler reserved at peak rate
- Deadline = expected arrival time + delay bound
  - If a source sends faster than contract, delay bound will not apply
- Each packet gets a hard delay bound
- Delay bound is *independent* of bandwidth requirement
  - but reservation is at a connection’s peak rate
- Implementation requires per-connection state and a priority queue

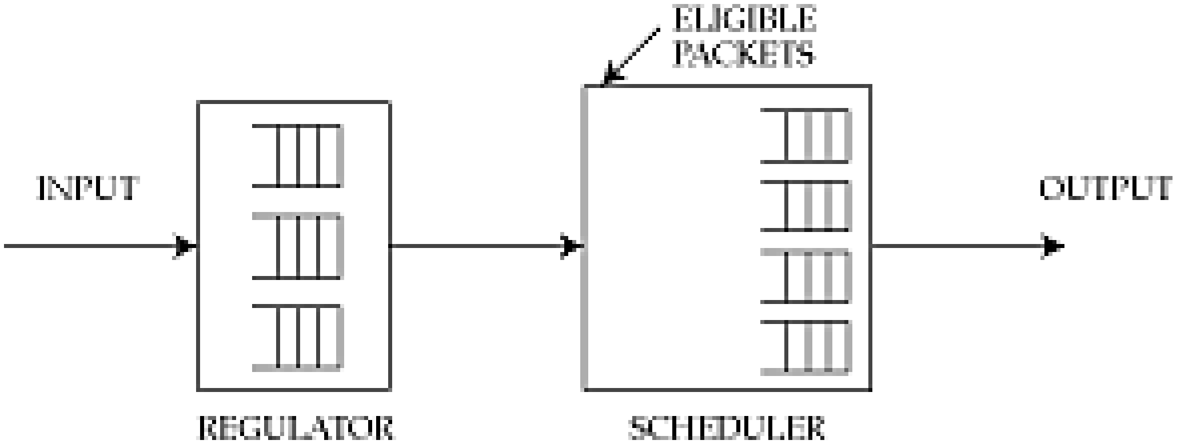
Fair-Share Scheduling (recall)

- The fair-share scheduling algorithms guarantee certain minimum rate between the flows
- These "rate-based" mechanisms can be classified into two categories
  - *rate-allocation service* discipline
  - *rate-controlled* service discipline


"≥" for Rate - Allocation
Flow *1*
≥r *1*
Flow *2*
≥r *2*
Service Rate
c ≥ r *1*+r *2* +. . .+r *m*
Flow *m*
≥r*m*
"=" for Rate - Controlled

Rate-controlled scheduling

- A *class* of disciplines
    - two components: regulator and scheduler
    - incoming packets are placed in regulator where they wait to become eligible
    - then they are put in the scheduler
- Regulator *shapes* the traffic, scheduler provides performance guarantees



Examples

- Regulators
  - rate-jitter regulator
    - scheduler bounds *maximum outgoing rate* by conforming to a rate descriptor
  - delay-jitter regulator
    - Scheduler *compensates for variable delay* at previous hop by guaranteeing the sum of queueing delay in previous switch and regulation delay in the current switch to be constant
- Rate-jitter regulator + FIFO
  - similar to Delay-EDD
- Rate-jitter regulator + multi-priority FIFO
  - gives both bandwidth and delay guarantees
- Delay-jitter regulator + EDD
  - gives bandwidth, delay,and delay-jitter bounds (Jitter-EDD)

Analysis

- First regulator on path monitors and regulates (*policing*) traffic on each connection=> bandwidth bound
- End-to-end delay bound
  - delay-jitter regulator
    - reconstructs traffic => end-to-end delay is fixed (= worst-case delay at each hop)
  - rate-jitter regulator
    - partially reconstructs traffic
    - can show that end-to-end delay bound is smaller than (sum of delay bound at each hop + delay at first hop)
  - Thus for both regulators, we need to only compute the worst case delay bound

Evaluation

- Pros
  - flexibility
  - can decouple bandwidth and delay assignments
  - end-to-end delay bounds are easily computed
  - do not require complicated schedulers to guarantee protection
  - can provide delay-jitter bounds
- Cons
  - require an additional regulator at each output port
  - delay-jitter bounds at the expense of increasing mean delay
  - delay-jitter regulation is expensive (clock synch, timestamps)

Summary

- Two sorts of applications: best effort and guaranteed service
- Best effort connections require fair service
  - provided by GPS, which is unimplementable
  - emulated by WFQ and its variants
- Guaranteed service connections require performance guarantees
  - provided by WFQ, but this is expensive
  - may be better to use rate-controlled schedulers