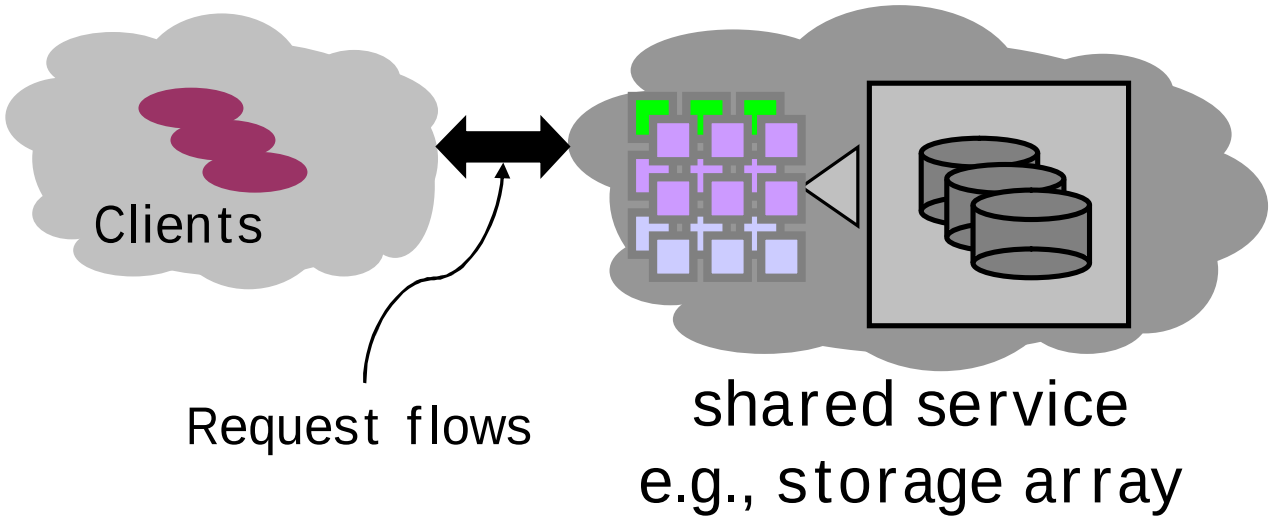
Interposed Proportional Sharing for a Storage Service Utility

Wei Jin
Jeff Chase
Duke University

Jasleen Kaur
UNC - Chapel Hill

Resource Sharing in Utilities



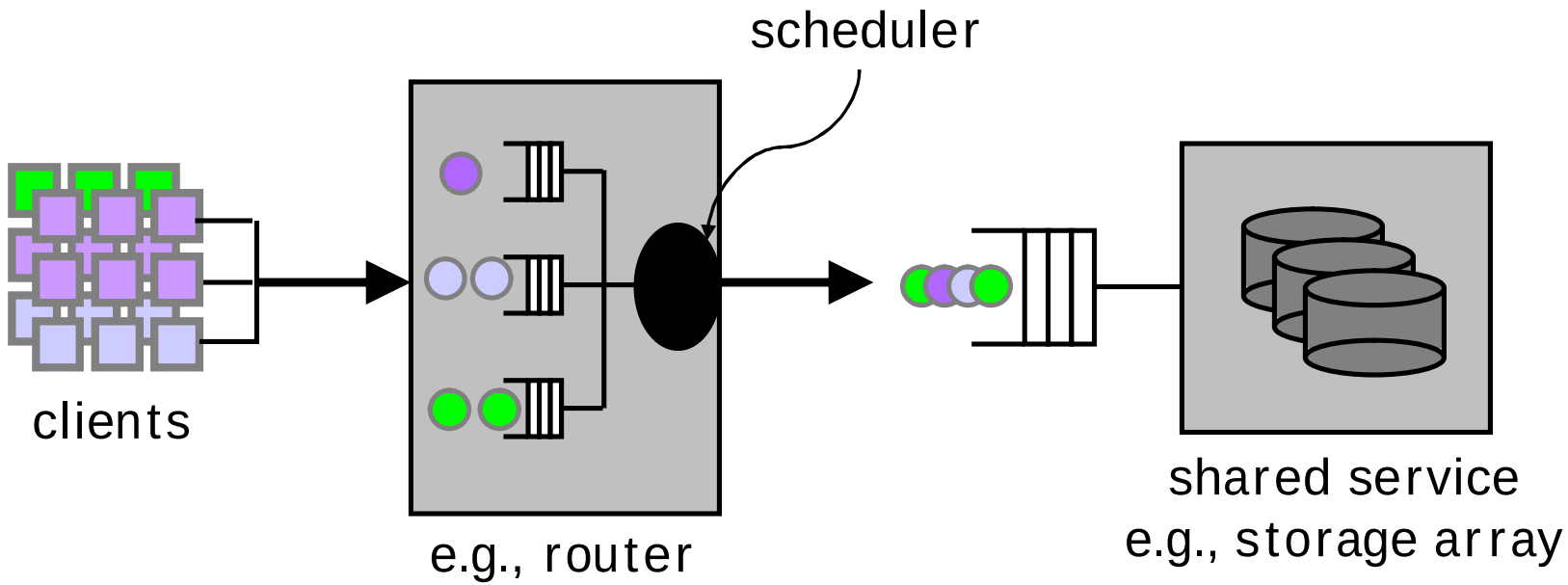
Resource efficiency
Adaptivity
Surge protection
Robustness
"Pay as you grow"
Economy of scale
Aggregation

- Resource sharing offers important benefits.
- But sharing must be "fair" to protect users.
- Shared services often have contractual performance targets for groups of clients or requests.
  - Service Level Agreements or SLAs

Goals

- Performance isolation
  - Localize the damage from unbudgeted demand surges.
- Differentiated service quality
  - Offer predictable, configurable performance (e.g., mean response time) for stable request streams.
- Non-invasive
  - External control of a “black box” or “black cloud”
  - Generalize to a range of services
  - No changes to service structure or implementation

Interposed Request Scheduling I



– Intercept and throttle or reorder requests on the path between the clients and the service [e.g., Lumb03].

– Build the scheduler into network switching components, or into the clients (e.g., servers in a utility data center).

– Manage request traffic rather than request execution.

Alternative Approaches

- Extend scheduler for each resource in a service.
  - Cello, Xen, VMware, Resource Containers, etc.
  - Precise but invasive, and must coordinate schedulers to manage sharing of an aggregate resource (server, array).
- Facade [Lumb03] uses Earliest Deadline First in an interposed request scheduler to meet response time targets.
  - Does not provide isolation, though priority can help.
  - Can admission control make isolation unnecessary?
- SLEDS [Chambliss03] is a per-client network storage controller using leaky bucket rate throttling.
  - Flows cannot exceed configured rate even if resources are idle.

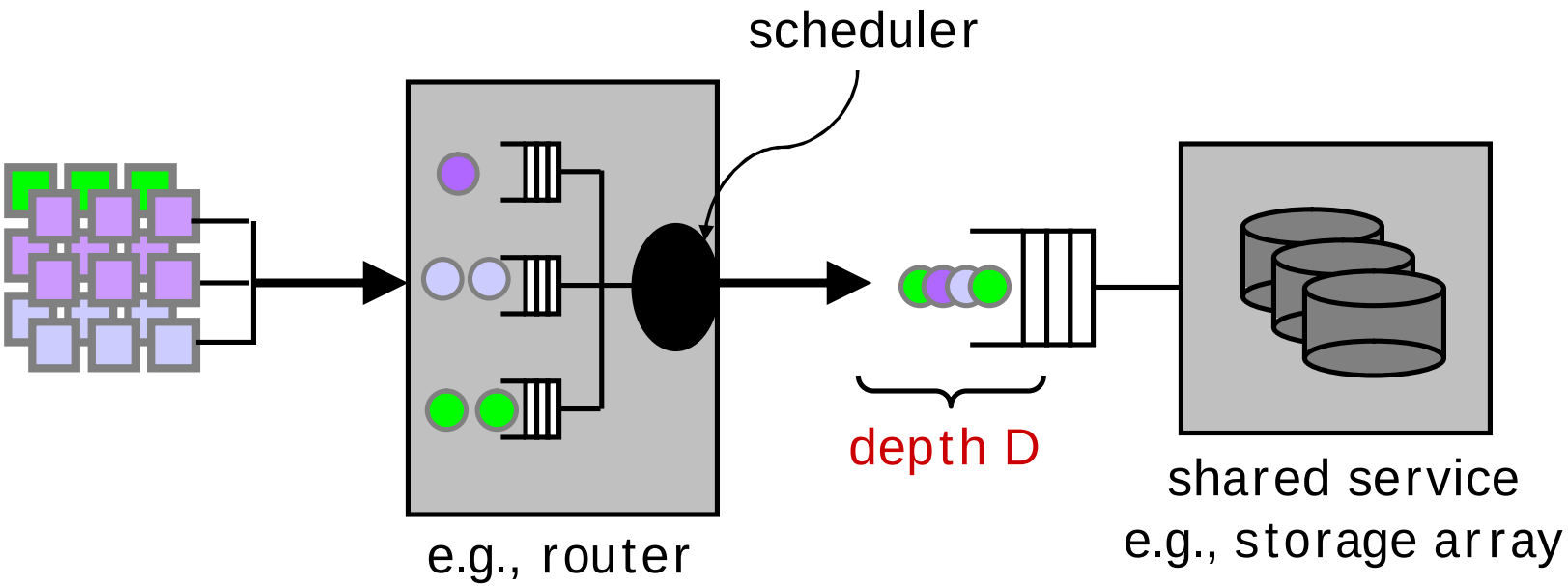
Proportional Sharing

- Each flow is assigned a weight $\bullet$.
- Allocate resources among active flows in proportion to their weights.
  - Work-conserving: allocate surplus proportionally
- Fairness
  - Lag is the difference in weighted work done on behalf of a pair of flows.
  - Prove a constant worst-case bound on lag for any pair of flows that are active over any interval.
  - "Use it or lose it": no penalty for consuming surplus resources.

Weights as Shares

- Weights define a configured or assured service rate.
  - Adjust weights to meet performance targets.
- Idealize weights as shares of the service's capacity to serve requests.
  - Normalize weights to sum to one.
- For network services, your mileage may vary.
  - Delivered service rate depends on request distribution, cross-talk, hotspots, etc.
  - Premise: behavior is sufficiently regular to adjust weights under feedback control.

Interposed Request Scheduling II

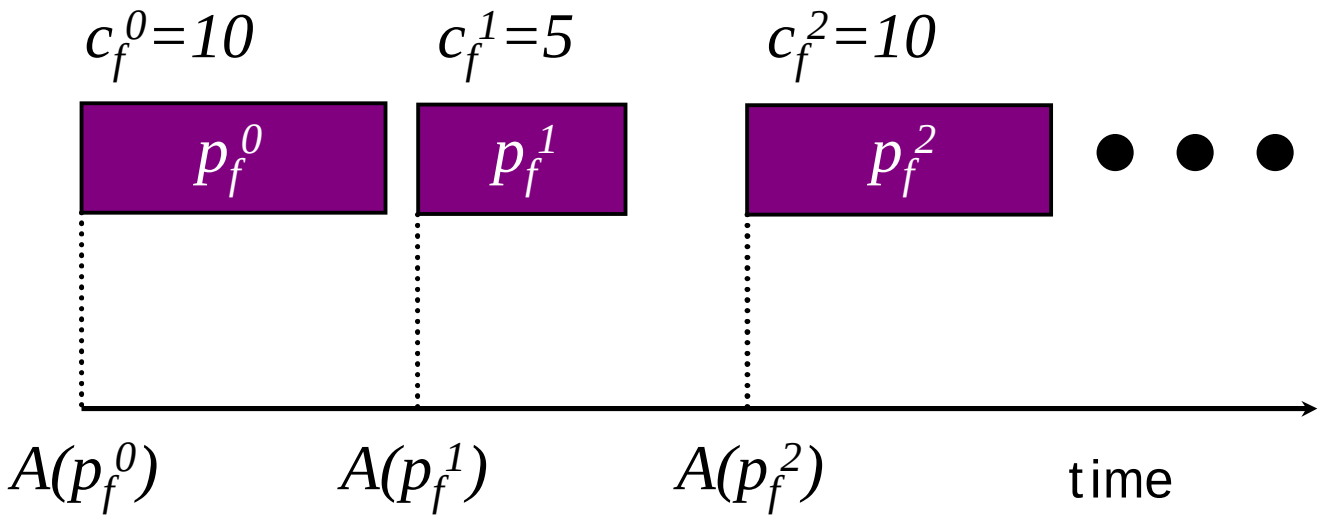


- Dispatch/issue up to D requests or D units of work.
- Issue requests to respect weights assigned to each flow.
- Choose D to balance server utilization and tight resource control.
- Request concurrency is defined/controlled by the server.

Overview

- Background on proportional share scheduling
  - Virtual Clock [Zhang90]
  - Weighted Fair Queuing [Demers89]
  - Start-time Fair Queuing or SFQ [Goyal97]
- New depth-controlled variants for interposed scheduling
  - Why SFQ is not sufficient: concurrency.
  - New algorithm: SFQ(D)
  - Refinement: FSFQ(D)
- Decentralized throttling with Request Windows (RW)
- Proven fairness results and experimental evaluation

A Request Flow



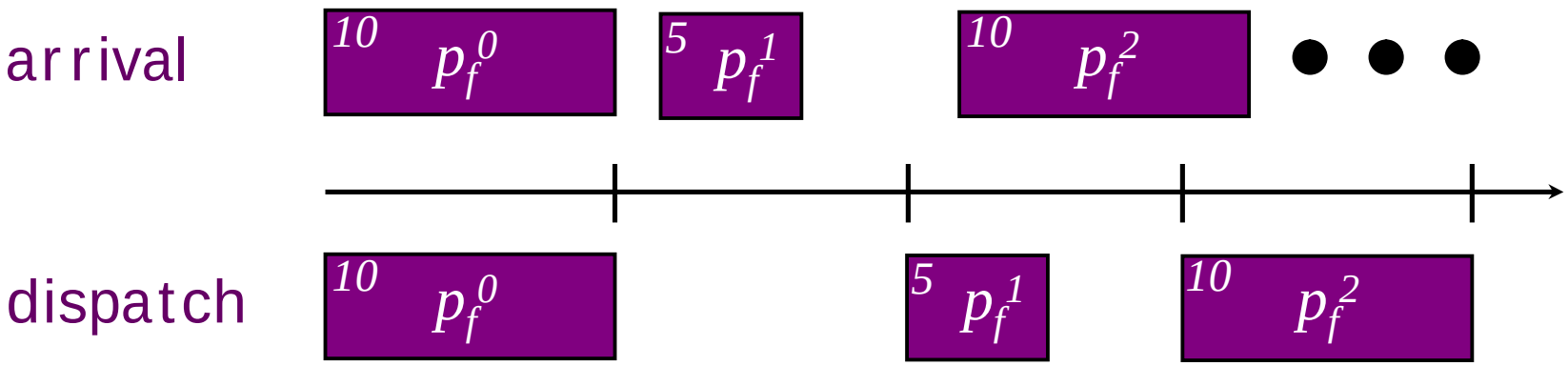
Consider a flow f of service requests.

- Could be packets, CPU demands, I/Os, requests for a service
- Each request has a distinct arrival time (serialize arrivals).
- Each request has a cost: packet length, service duration, etc.

Request Costs

- Can apply to any service if we can estimate the cost of each request.
- Relatively easy to estimate cost for block storage.
- Fairness results are relative to the estimated costs; they are only as accurate as the estimates.

A Flow with a Share

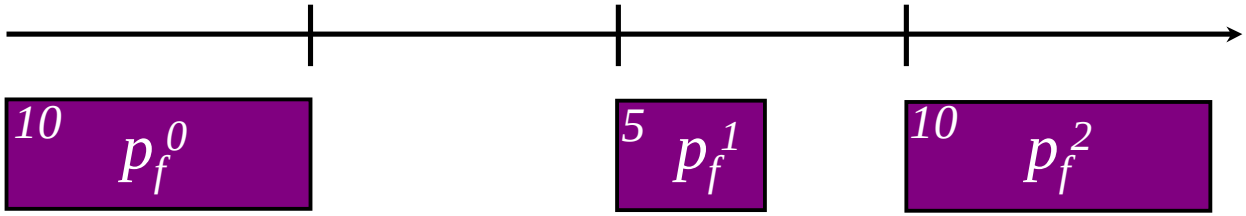


Consider a sequential unit resource: capacity is 1 unit work/time unit.

- Suppose flow f has a configured share of 50% ($\bullet_f = 0.5$).
- f is assured T units of service in $T/\bullet_f$ units of real time.
- How to implement shares/weights in an interposed request scheduler?

Virtual Clock

Each arriving request is tagged with a start (eligible) time and a finish time.



$S(p_f^0) = 0$ $S(p_f^1) = 20$ $S(p_f^2) = 30$

$F(p_f^0) = 20$ $F(p_f^1) = 30$ $S(p_f^2) = 50$

$$S(p_f^i) = F(p_f^{i-1})$$

$$F(p_f^i) = S(p_f^i) + \frac{c_f^i}{\bullet_f}$$

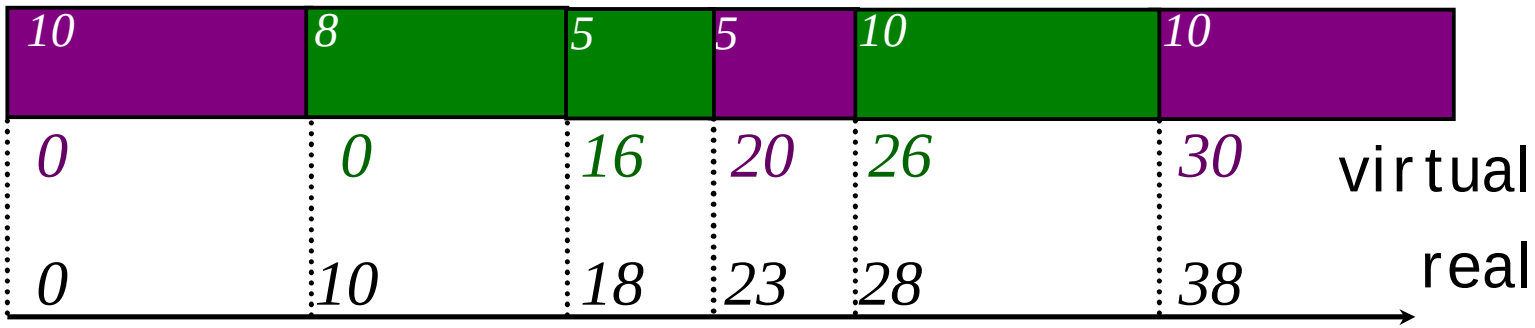
[Zhang90]

View the tags as a virtual clock for each flow.

Each request advances the flow's clock by the amount of real time until its next request must be served.

If the flow completes work at its configured service rate, then virtual time • real time.

Sharing with Virtual Clock



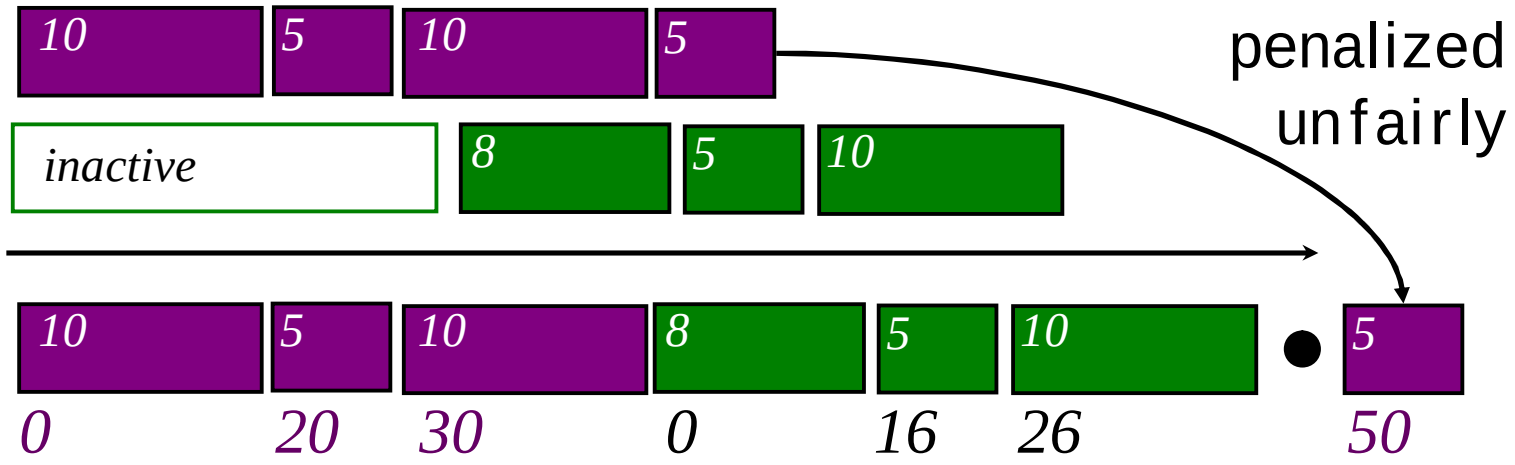
Virtual clock scheduler [Zhang90] orders the requests/packets by their virtual clock tags.

This example:

- shows two flows each at • =50%
- assumes both flows are active and backlogged

What if a flow does not consume its configured share?

Virtual Clock is Unfair



A scheduler is work-conserving if the resource is never left idle while a request is queued awaiting service.

Virtual Clock is work-conserving, but it is unfair: an active flow is penalized for consuming idle resources.

The lag is unbounded: really want a “use it or lose it” policy.

Weighted Fair Queuing

$$S(p_f^i) = max\,(v(A(p_f^i)), F(p_f^{i-1}))$$

$$F(p_f^i) = S(p_f^i) + \frac{c_f^i}{\bullet_f}$$

$$\frac{\bullet v(t)}{\bullet t} \bullet \frac{C}{\bullet\bullet_i} \quad for\ active\ flows\ i$$

Define system virtual time v(t), which advances with the progress of the active flows.

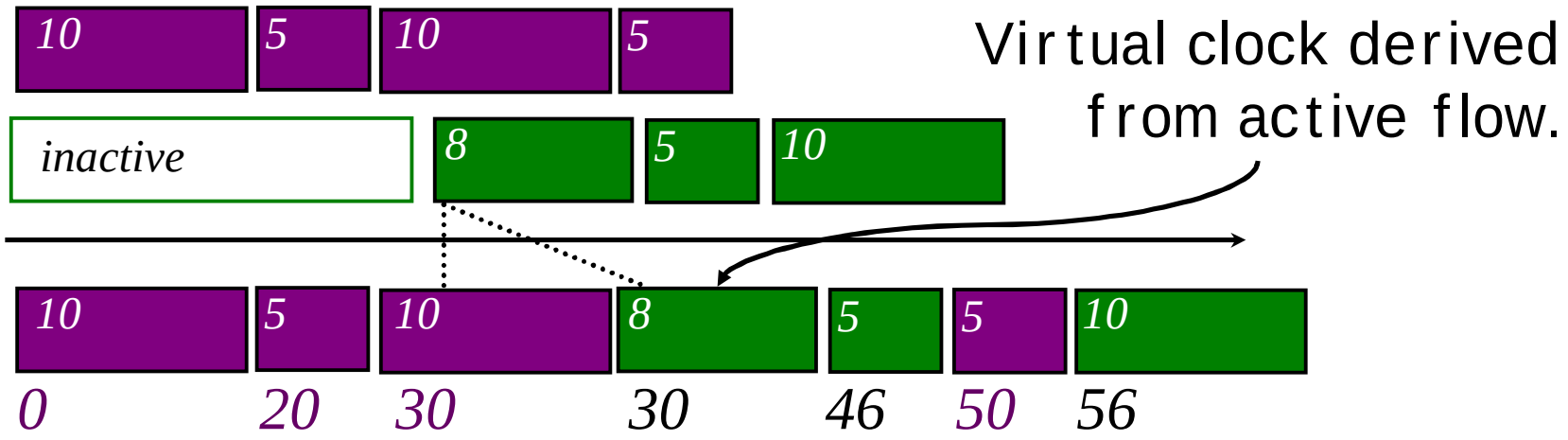
- Less competition speeds up v(t); more slows it down.

Advance (lagging) clock of a newly active flow to the system virtual time, to relinquish its claim to resources it left idle.

How to maintain v(t)?

- Too fast? Reverts to FIFO.
- Too slow? Reverts to Virtual Clock.

Start-Time Fair Queuing (SFQ)



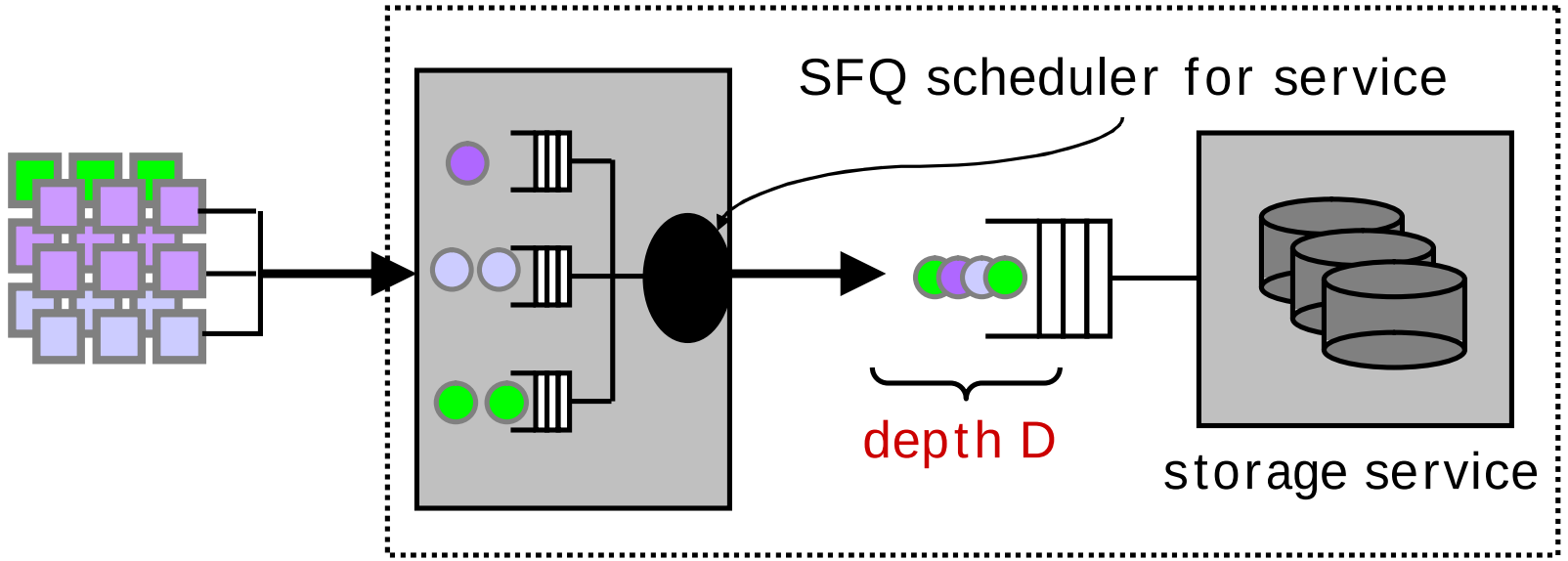
SFQ derives v(t) from the start tag of the request in service.

Use the resource itself to drive the global clock.

- Order requests by start tag [Goyal97].
- Cheap to compute v(t).
- Fair even if capacity (service rate) C varies.
- Lag between two backlogged flows is bounded by:

$$\frac{c_f^{max}}{\bullet_f} + \frac{c_g^{max}}{\bullet_g}$$

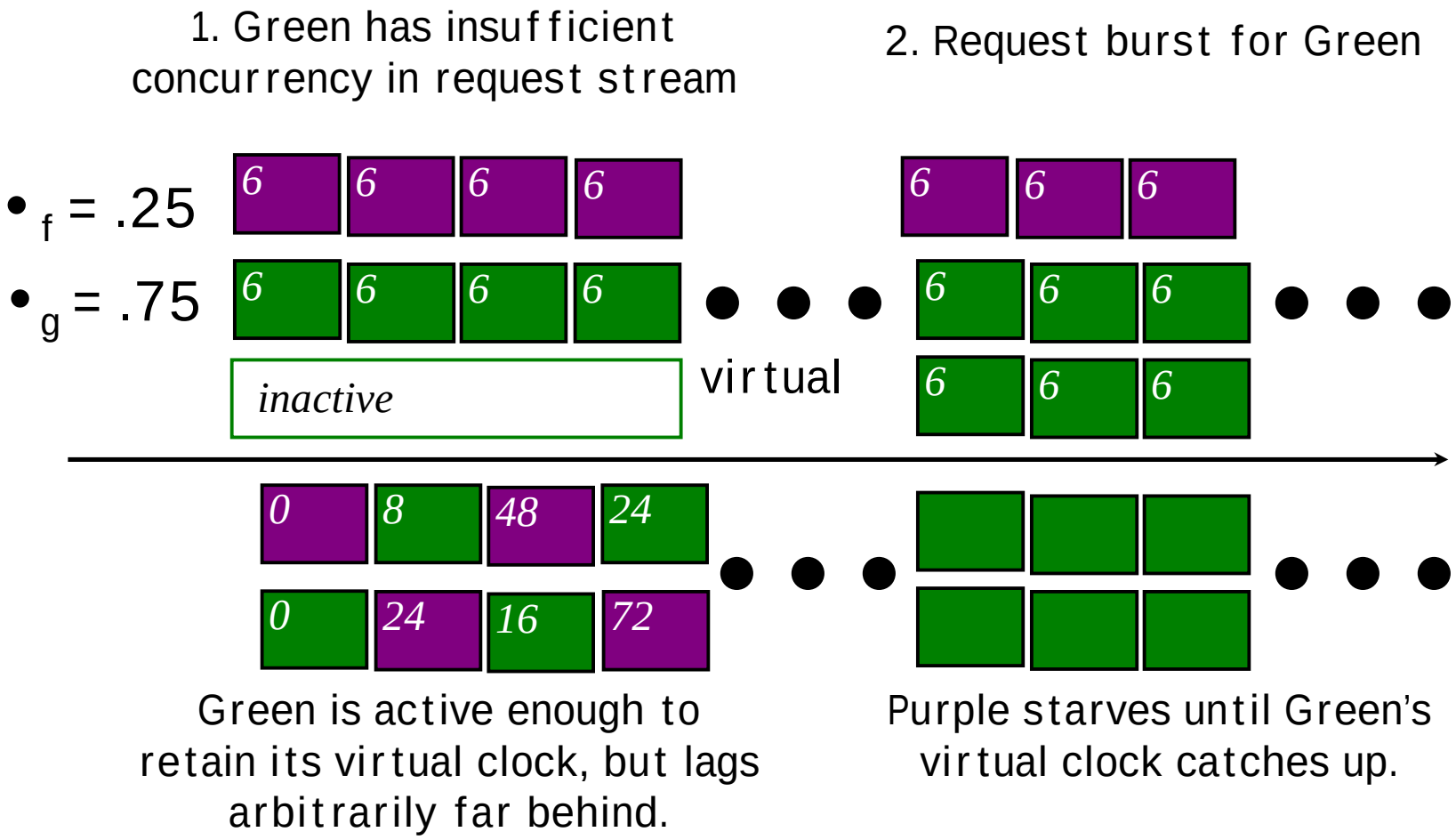
SFQ for Interposed Scheduling?



Challenge: concurrency.

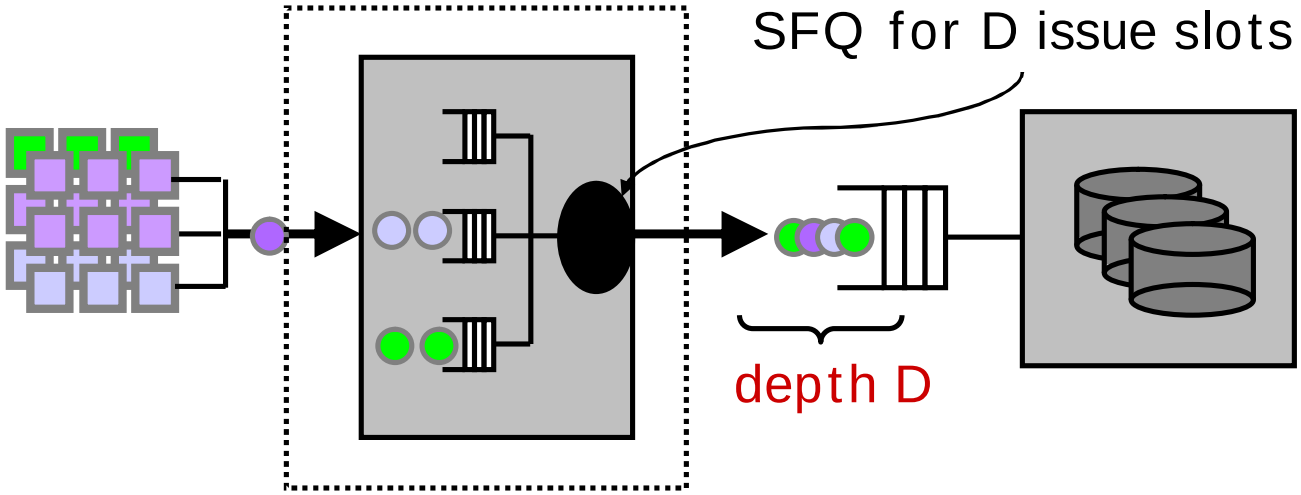
- Up to D requests are “in service” concurrently.
- SFQ virtual time v(t) is no longer uniquely defined.
- Direct adaptation: Min-SFQ(D) takes min of requests in service.

Min-SFQ is Unfair



Problem: v(t) advances with the slowest active flow: clock skew causes the algorithm to degrade to Virtual Clock, which is unfair.

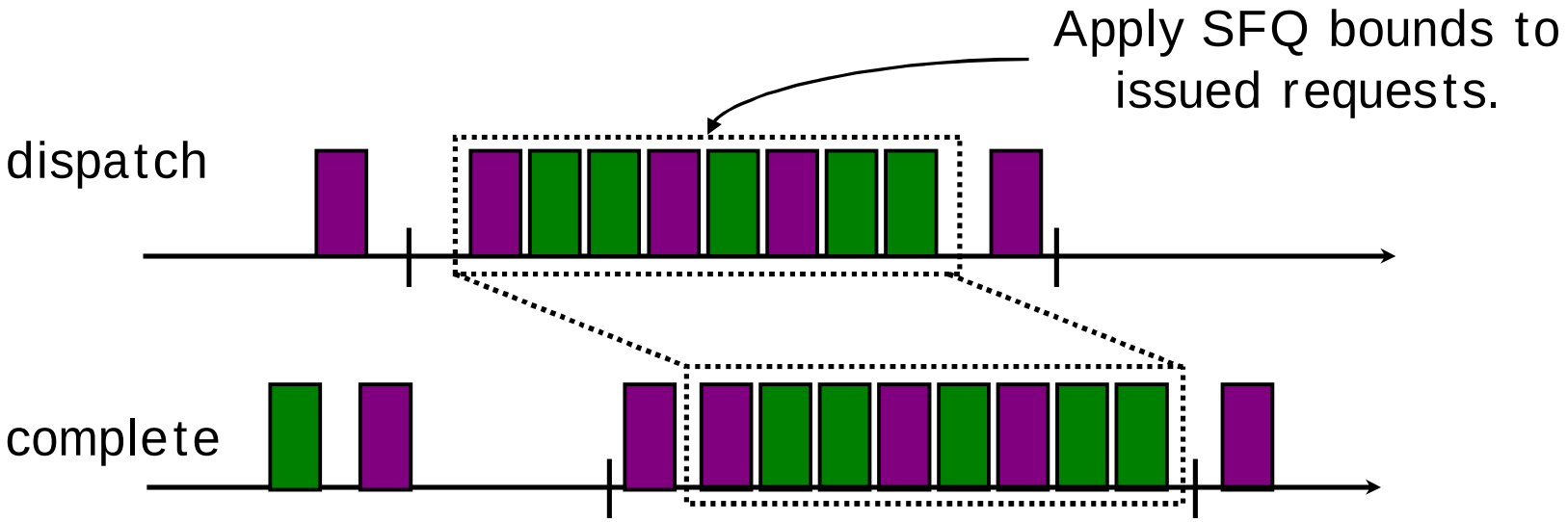
SFQ(D)



Solution: take v(t) from clocks of backlogged flows.

- Take v(t) as min tag of queued requests awaiting dispatch.
  - (The start tag of the request that will issue next.)
- Implementation: take v(t) from the last issued request.
- Equivalent to scheduling the sequence of issue slots with SFQ.

SFQ(D) Lag Bounds



SFQ lag bounds apply to requests issued under SFQ(D).

From this we can derive the lag bound for requests completed under SFQ(D).

Lag between two backlogged flows f and g is bounded by:

$$(D+1)\left(\frac{c_f^{max}}{\bullet_f} + \frac{c_g^{max}}{\bullet_g}\right)$$

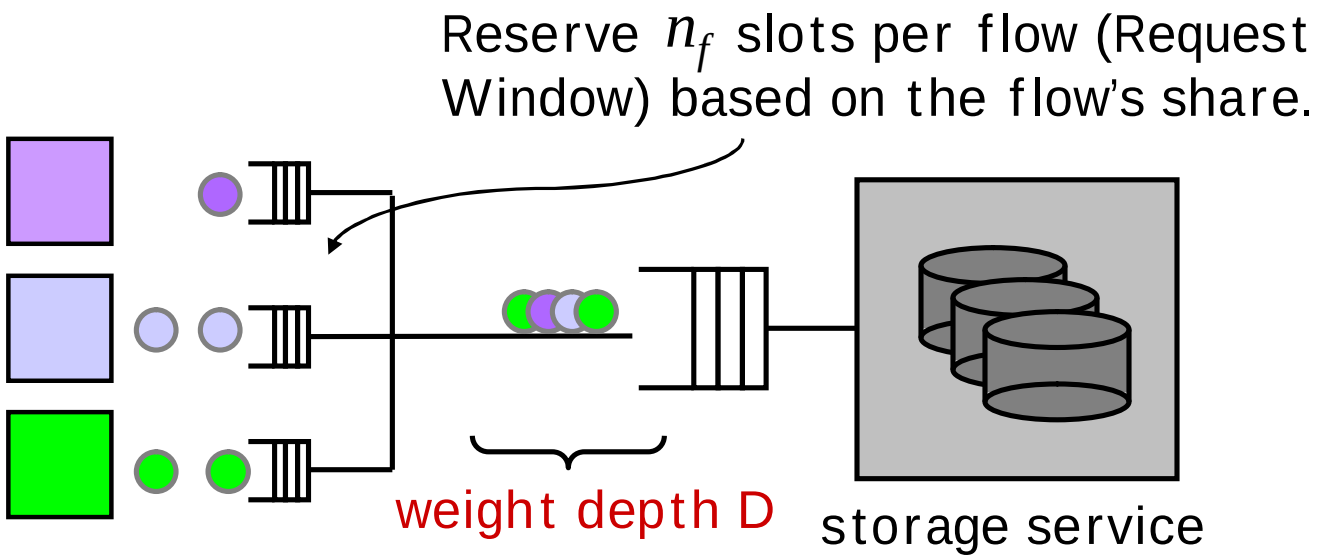
Refining SFQ(D)

- SFQ(D) virtual time advances monotonically, but advances at most once per request issue.
- Bursts of requests may receive the same start tag, including requests from active flows that are “ahead”.
- To be fair, the scheduler should bias against flows that hold more than their share of issue slots.
- Four-tag Start-time Fair Queuing (FSFQ(D)) is a refinement to SFQ(D).
  - Break ties with a second pair of “adjusted” tags derived from Min-SFQ(D).

Request Windows: Motivation

- SFQ(D) and FSFQ(D) assume a central point of control over the request flows.
  - Designed to reside within a service switch, e.g., a network storage router.
  - Single point of complexity and vulnerability.
- Any central scheduler requires log(F) overhead to select the next request.
- Throttling can improve delay bounds by reserving issue slots.

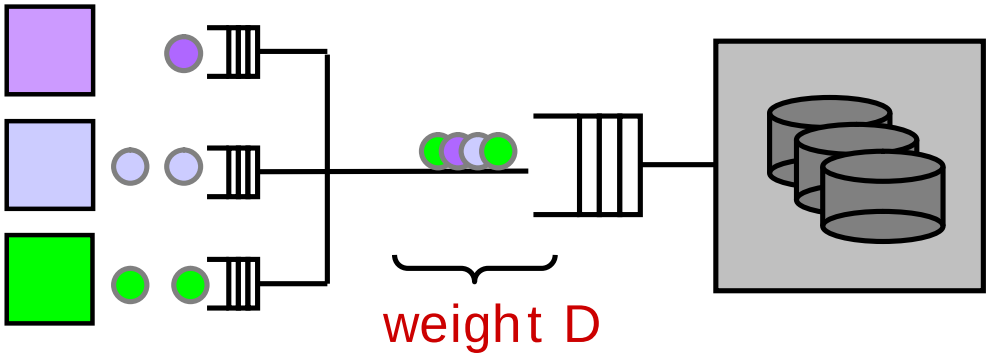
Request Windows



Limit each flow to its share of the total weight (D) allowed into the system from all flows.

$$n_f = D \frac{\bullet_f}{\bullet\bullet_i} \quad \textit{for each flow f and all flows i}$$

Behavior of Request Windows



Is RW work-conserving? It does allow a flow to exceed its configured service rate under light load.

- Window constrains the outstanding requests, not rate.
- Per-flow issue rate increases with service rate.
- Balance tight control with concurrency under light load.

Theorem: Lag between any two persistently backlogged flows is bounded by 2D for a FIFO server.

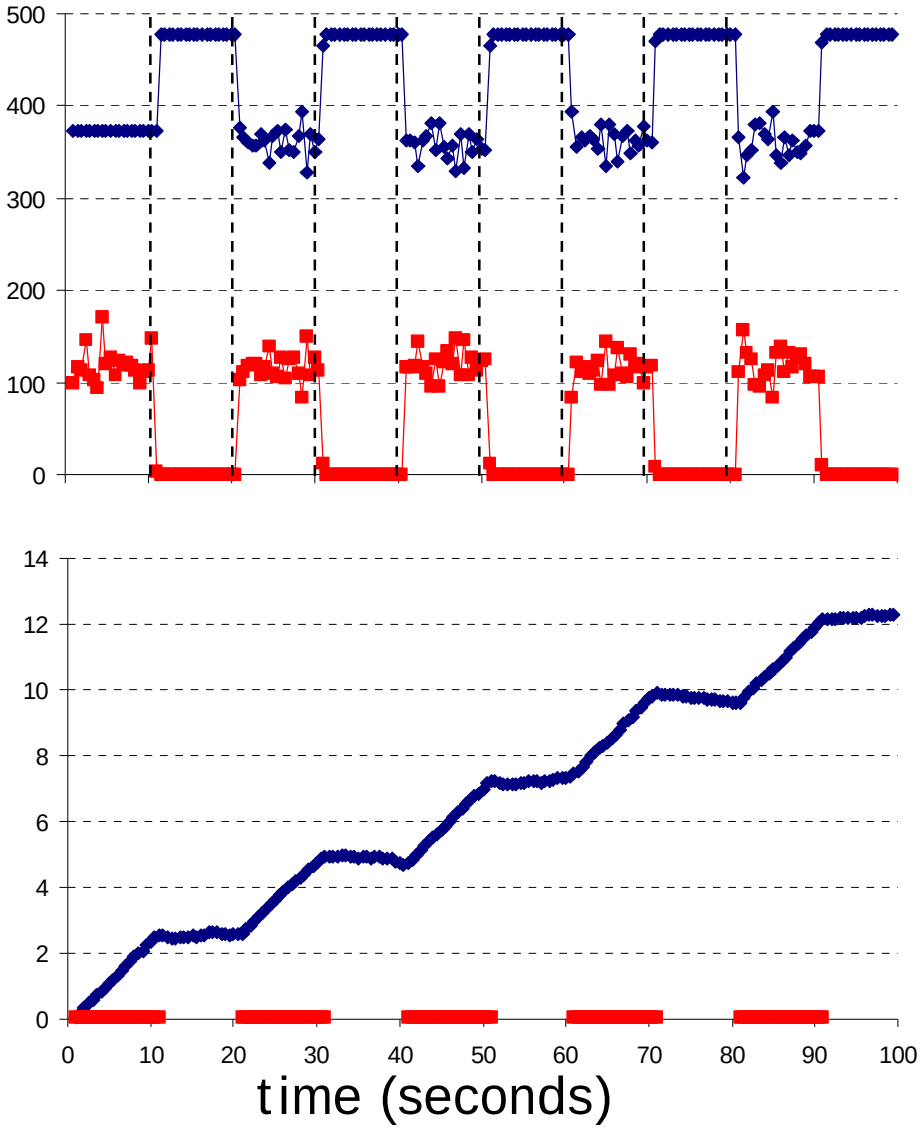
Experiments

- Implemented an NFS proxy for interposed request scheduling.
  - Extends Anypoint [Yocum03] redirecting switch prototype.
  - SFQ(D), FSFQ(D), EDF in about 1000 lines of code.
- Implemented a disk array simulator.
- Used prototype to validate simulator for random read workloads (fstress load generator [Anderson02]).
- Simulated random read workloads with varying depth, arrival rate, and shares.

Performance Isolation with FSFQ

throughput (IOPS)

response time (seconds)



Blue: 480 IOPS
67% share

Red: ON/OFF
0/120 IOPS
10 sec intervals
33% share

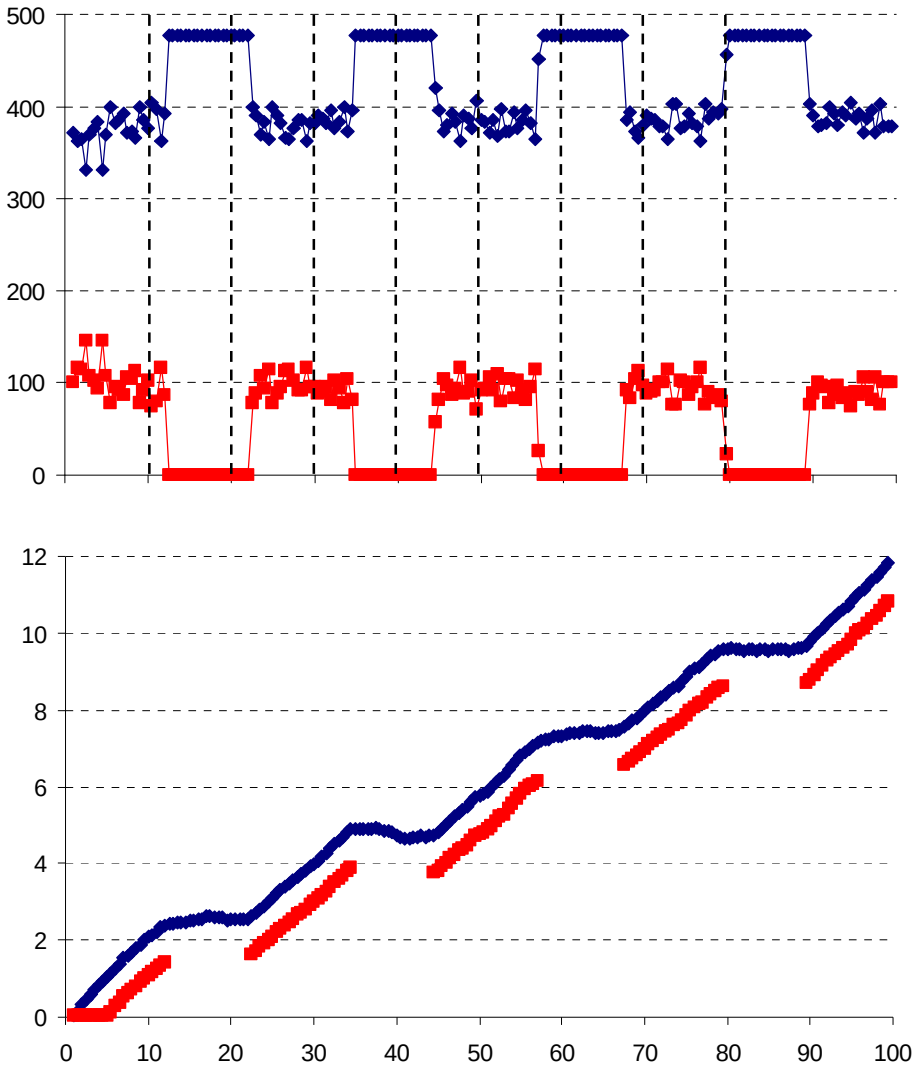
Server saturation
@500 IOPS

FSFQ(16)
or RW(16)

EDF Alone is Not Sufficient

throughput (IOPS)

response time (seconds)



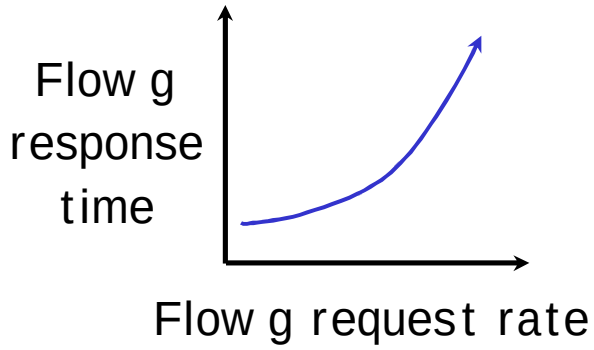
Blue: 480 IOPS
target = 1 second

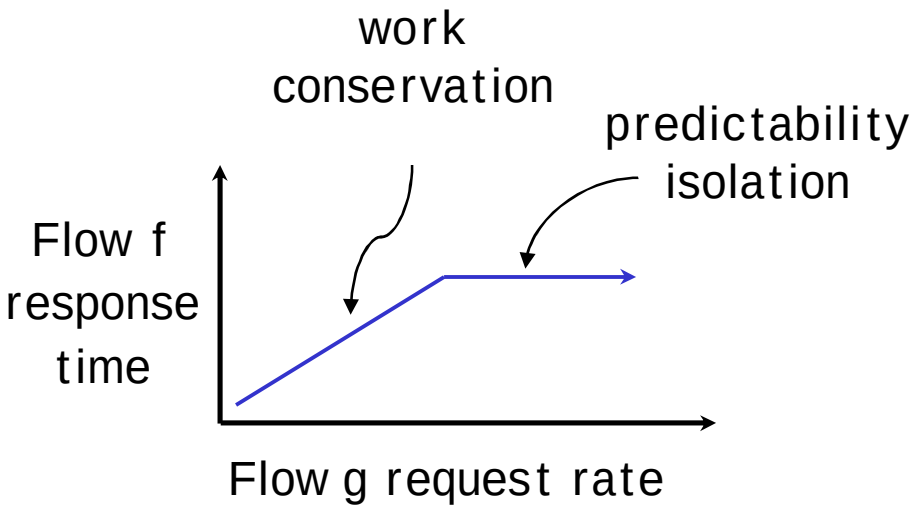
Red: ON/OFF
0/120 IOPS
10 sec intervals
target = 10 ms

Server saturation
@500 IOPS

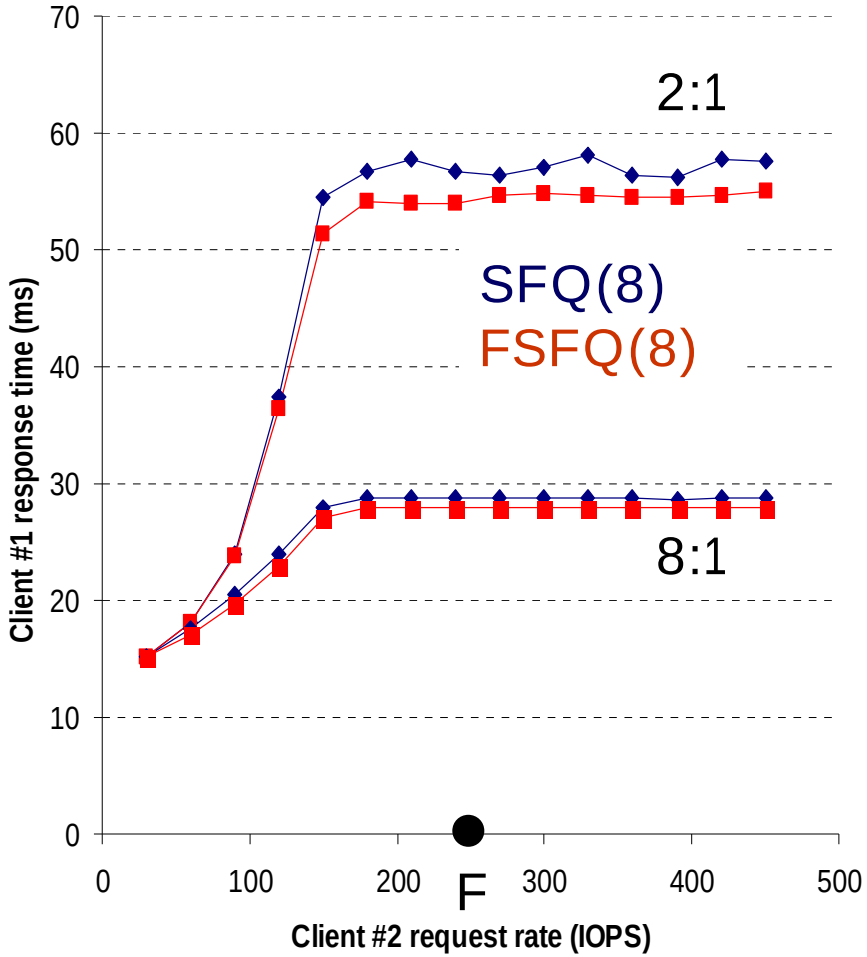
Preview

- Flow f issues requests at a fixed arrival rate.
- Competitor g increases its request rate on X-axis.
- Plot mean response time for f on Y-axis.
- Evaluate performance isolation, work conservation.





SFQ and FSFQ

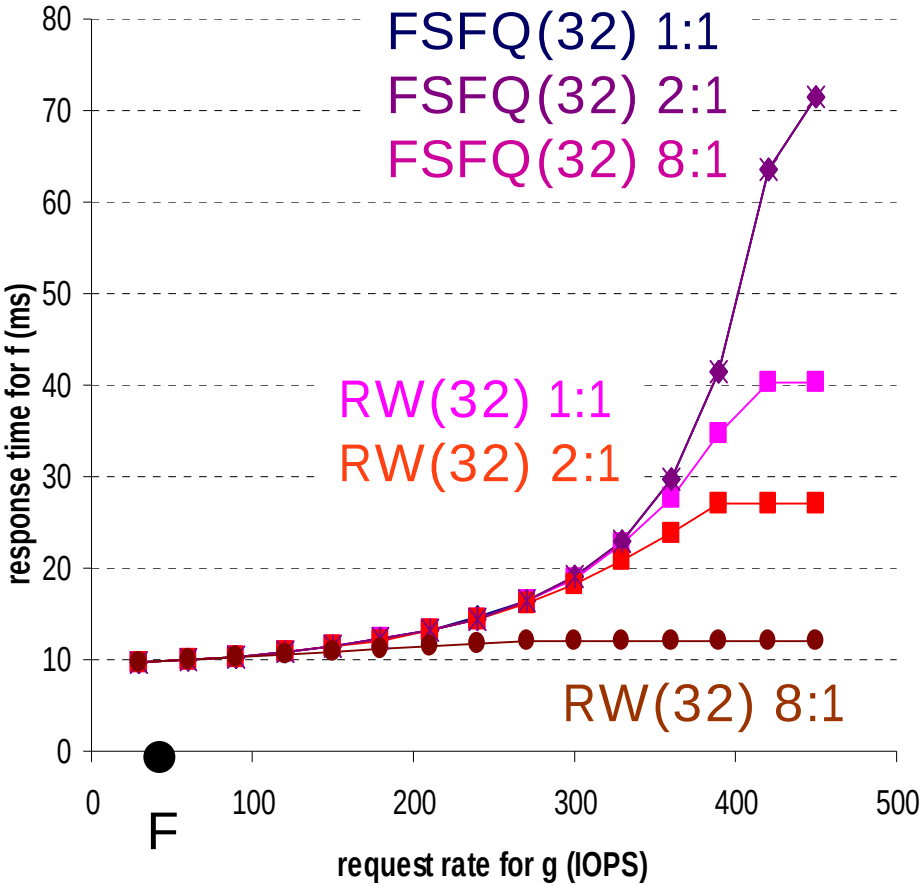


- f's response time stabilizes at a level determined by its weight.
- f's response time improves when g's load is low.
- FSFQ improves fairness modestly.

Other results

- g's response time degrades without bound as its load exceeds its share.
- When f generates low load, response times improve for both flows, and the stable level is less sensitive to weight.

FSFQ and RW

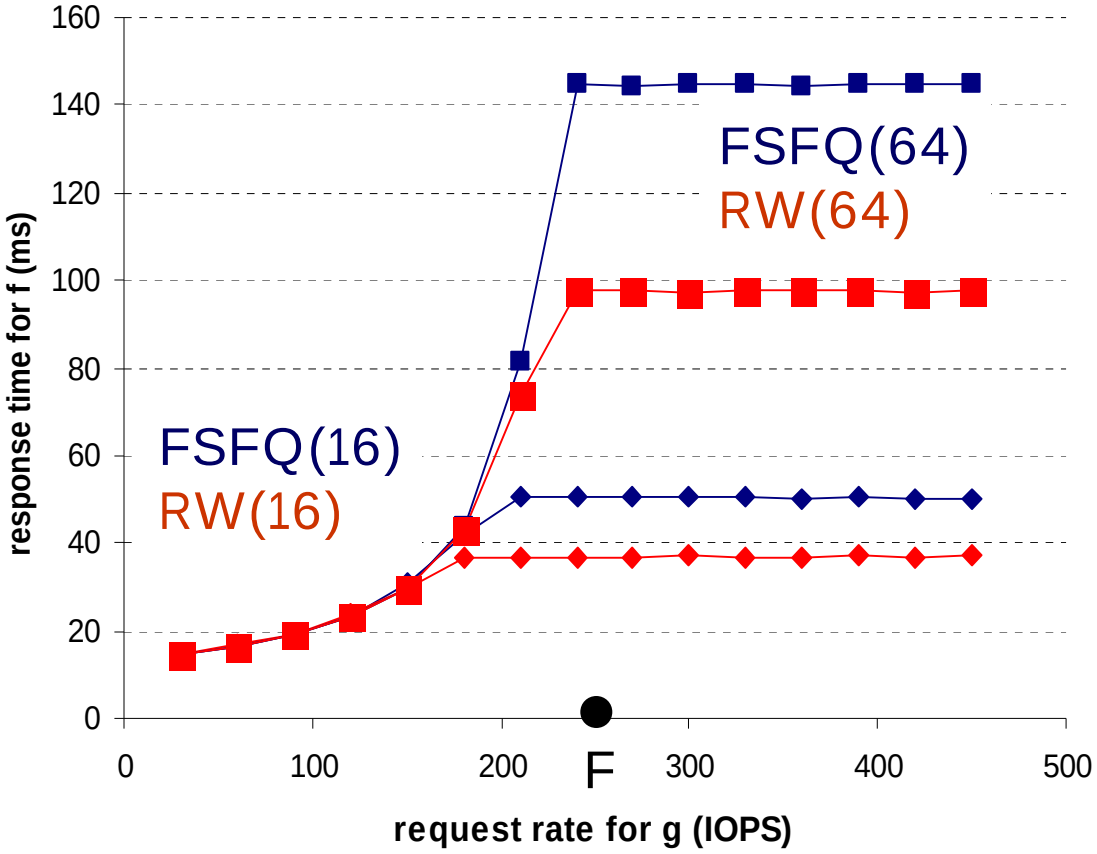


- As expected....
- RW isolates f more effectively than FSFQ because it limits the ability of g to consume slots left idle by f.

Other results

- FSFQ(32) is similar to RW(32) with f@240 IOPS.
- FSFQ(32) is less effective than FSFQ(8).

Effect of Depth



- 2:1 weights
- Increasing D weakens control
- RW offers tighter control than FSFQ.
- FSFQ uses surplus resources more aggressively.

Summary of Results

- Interposed request scheduling with *SFQ and RW offers acceptable performance isolation and is non-invasive.
  - Predictable, configurable differentiated service.
  - With larger systems depth must increase. The algorithms are fair and isolating even with high D, but cannot support tight response time bounds.
  - In a work-conserving system, a flow with low utilization of its share experiences weaker isolation.
  - FSFQ(D) yields modest improvements over SFQ(D).
  - RW(D) offers stronger isolation than *SFQ, but is “less work-conserving” (more like a reservation).

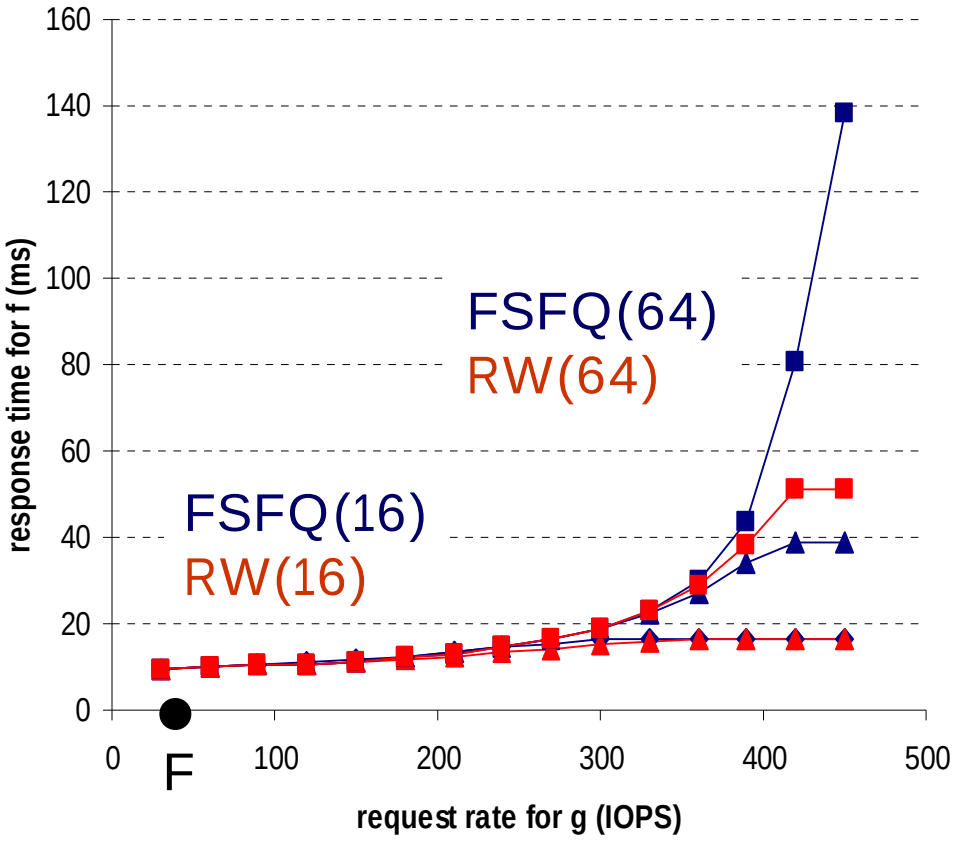
Further Study

- How precisely can we estimate costs?
  - Workload crosstalk, e.g., disk arm movement
- Assumes internally balanced load
  - Internal bottlenecks can slow service rate and “bleed over” into other shares.
  - May need some component-local status/control if/when significant load imbalances exist (e.g., Stonehenge).
- Explore hybrids of *SFQ(D) and RW(D) for varying balances of decentralization and control.
  - Degree of control is reduced as we increase parallelism within the cloud.
- Sizing shares for response-time SLAs.

http://issg.cs.duke.edu/publications/shares-sigmet04.pdf
(Enhanced/corrected version of paper)

http://www.cs.duke.edu/~chase

Effect of depth for a low-demand flow



- 2:1 weights
- Increasing D weakens control
- f response times increase; g response times decrease
- RW offers tighter control than FSFQ
- FSFQ uses surplus resources more aggressively