Scheduling and queue management

DigiComm II

Traditional queuing behaviour in routers

- Data transfer:
  - datagrams: individual packets
  - no recognition of **flows**
  - connectionless: no signalling
- Forwarding:
  - based on per-datagram, forwarding table look-ups
  - no examination of "type" of traffic – no **priority** traffic
- Traffic patterns

DigiComm II

Let us first examine the service that IP offers. IP offers a **connectionless** datagram service, giving no guarantees with respect to delivery of data: no assumptions can be made about the delay, jitter or loss that any individual IP datagrams may experience. As IP is a connectionless, datagram service, it does not have the notion of **flows** of datagrams, where many datagrams form a sequence that has some meaning to an applications. For example, an audio application may take 40ms "time-slices" of audio and send them in individual datagrams. The correct sequence and timeliness of datagrams has meaning to the application, but the IP network treats them as individual datagrams with no relationship between them. There is no **signalling** at the IP-level: there is no way to inform the network that it is about to receive traffic with particular handling requirements and no way for IP to tell signal users to back-off when there is congestion.

At IP routers, the forwarding of individual datagrams is based on forwarding tables using simple metrics and (network) destination addresses. There is no examination of the type of traffic that each datagram may contain – all data is treated with equal **priority**. There is no recognition of datagrams that may be carrying data that is sensitive to delay or loss, such as audio and video.

As IP is a totally connectionless datagram traffic, there is no protection of the packets of one flow, from the packets of another. So, the traffic patterns of one particular user's traffic affects traffic of other users that share some part or of, or all of, the network path (and perhaps even traffic that does not share the same network path!).

Questions

- How do we modify router scheduling behaviour to support QoS?
- What are the alternatives to FCFS?
- How do we deal with congestion?

DigiComm II

So we can ask ourselves several questions.

Firstly, can we provide a better service that that which IP currently provides – the so-called best-effort?

The answer to this is actually, "yes", but we need to find out what it is we really want to provide! We have to establish which parameters of a real-time packet flow are important and how we might control them. Once we have established our requirements, we must look at new mechanisms to provide support for these needs in the network itself. here are many functional elements required in order to provide QoS in the network, some of which we will look at later. Here, we are essentially trying to establish alternatives to FCFS for providing better control of packet handling in the network.

We also need to consider how the applications gain access to such mechanisms, so we must consider any **application-level interface** issues, e.g. is there any interaction between the application and the network and if so, how will this be achieved.

In all our considerations, one of the key points is that of **scalability** – how would our proposals affect (and be affected by) use of IP on a global scale, across the Internet as a whole. Also, we must try to adhere, as much as possible, to the current service interface, i.e. a connectionless datagram delivery.

Scheduling mechanisms

DigiComm II

Scheduling [1]

- Service request at server:
  - e.g. packet at router inputs
- Service order:
  - which service request (packet) to service first?
- Scheduler:
  - decides service order (based on policy/algorithm)
  - manages service (output) queues
- Router (network packet handling server):
  - **service:** packet forwarding
  - **scheduled resource:** output queues
  - **service requests:** packets arriving on input lines

DigiComm II

In general, the job a a scheduler is to control access to resources at some kind of server. The server offer a service and receives service requests. the way in which the scheduler determines which service request is dealt with next is subject to some well-defined policy or algorithm.

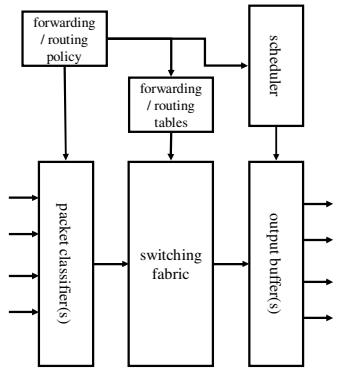
The main job of a scheduler is to make decisions regarding the order in which service request should be allowed access to resources. It has a secondary job which is to manage the service queues. This secondary roles reflects that fact that, in the face of excessive service requests, the finite resources of the server must be managed in a way that is consistent with the policy that is established at the server.

For a router (a network packet handling server), the service being offered is a packet forwarding service; the resource which is under the control of the scheduler is (are) the output queue(s) of the router; service requests are packets that need forwarding arriving on input line(s).

Scheduling [2]

Simple router schematic

- Input lines:
  - no input buffering
- Packet classifier:
  - policy-based classification
- Correct output queue:
  - forwarding/routing tables
  - switching fabric
  - output buffer (queue)
- Scheduler:
  - which output queue serviced next



DigiComm II

In a model of a simple router, we have the following functional elements.

forwarding/routing policy: this is part of the decision making process that is set-up by a network administrator. The policy includes information about how to classify packets as well as any constraints on routing an forwarding.

forwarding/routing tables: these are constructed by the normal operation of the routing protocols and algorithms that run in the router.

input lines: as packets arrive interrupts are generated to indicate that an input should be processed. We assume that for a QoS sensitive network, router do not perform input buffering. Although input buffering is possible, packets may have their QoS requirements violated as they wait in input buffers, so we assume that packets are processed at line speed.

packet classifier: this makes some policy-based decision in order to classify the packets. The packet classification will allow the scheduler to make decisions about the order in which packets are serviced.

switching fabric: this moves a packet in an input queue to the appropriate output queue.

output queue: hold packets awaiting transmission.

scheduler: makes decisions as to which output queue should be serviced next.

More information router design and scheduling can be found in [KS98] and [KLS98].

FCFS scheduling

- Null packet classifier
- Packets queued to outputs in order they arrive
- Do packet differentiation
- No notion of flows of packets
- Anytime a packet arrives, it is serviced as soon as possible:
  - FCFS is a **work-conserving** scheduler

DigiComm II

In first-come first-served (FCFS) scheduling, the packets are scheduled in the order they arrive at the router. No other packet differentiation is performed. The router has no notion of flows of packets. In fact, FCFS scheduling is a very function and the main concern of most FCFS scheduled routers is queue management – coping with excessive bursts of packets with finite buffer space. We look at key management techniques and congestion control later.

There is one very important aspect of FCFS which we will examine before we go on to discuss other scheduling disciplines, and that is that it is a **work-conserving** scheduling disciple. That is, the router is never idle when there are packets waiting to be serviced.

Conservation law [1]

- FCFS is work-conserving:
  - not idle if packets waiting
- Reduce delay of one flow, increase the delay of one or more others
- We can not give *all* flows a lower delay than they would get under FCFS

$$\sum_{n=1}^{N} \rho_n q_n = C$$

$\rho_n = \lambda_n \mu_n$

ρ_n : mean link utlisation

q_n : mean delay due to scheduler

C : constant [s]

λ_n : mean packet rate [p/s]

μ_n : mean per – packet service rate [s/p]

DigiComm II

An important theorem due to Kleinrock (p.117 in [Kle75]) helps us in analysing scheduling disciplines. This theorem is **The Conservation Law**. Consider N flows arriving at a scheduler, so that the traffic rate of connection n ($1 \le n \le N$) is λ_n. If we assume that flow n has a mean service rate of μ_n. So, the mean utilisation of a link by flow n, ρ_n, is $\lambda_n \mu_n$. Let the mean waiting time due to the scheduler for the packets of flow n be q_n. According to the Conservation Law, the scheduler is then work conserving if:

$$\sum_{n} \rho_n q_n = C$$ where C is a constant value

This expression is important as it makes no reference to any particular scheduling disciplines. This expression also says that for any work conserving scheduling discipline, if one of the flows is given a lower delay by the scheduler, then it must be by increasing the delay for one or more of the other flows. This is a fundamental result.

Conservation law [2]

Example

- μ_n : 0.1ms/p (fixed)
- Flow f1:
  - λ_1 : 10p/s
  - q_1 : 0.1ms
  - $\rho_1 q_1 = 10^{-7}$s
- Flow f2:
  - λ_2 : 10p/s
  - q_2 : 0.1ms
  - $\rho_2 q_2 = 10^{-7}$s
- $C = 2\times10^{-7}$s

- Change f1:
  - λ_1 : 15p/s
  - q_2 : 0.1s
  - $\rho_1 q_1 = 1.5\times10^{-7}$s
- For f2 this means:
  - decrease λ_2?
  - decrease q_2?
- Note the trade-off for f2:
  - **delay vs. throughput**
- Change service rate (μ_n):
  - change service **priority**

Let us demonstrate this using an example. Assume we have two equal flows, initially, f1 and f2. We assume that packet service rate is a function of the hardware/software of the router (though it is possible it is also a function of the average packet size in the flow) and is therefore fixed and equal for all flows. For the initial situation shown ab0ve, we can use the conservation law to evaluate *C*. If we now change the attributes f flow 1, so that we would like it to have a higher data rate, we find that the Conservation Law tells us that we must either decrease the data rate of f2 to maintain its data rate, or decrease its data rate to maintain the delay.

It is possible to change the service rate of each flow by changing the **priority** of the flow. Here, we have assumed that both flows have the same, i.e. FCFS scheduling is being used.

Non-work-conserving schedulers

- Non-work conserving disciplines:
  - can be idle even if packets waiting
  - allows "smoothing" of packet flows
- Do not serve packet as soon as it arrives:
  - what until packet is **eligible** for transmission
- Eligibility:
  - fixed time per router, or
  - fixed time across network

✓ Less jitter

✓ Makes downstream traffic more predictable:
  - output flow is controlled
  - less bursty traffic

✓ Less buffer space:
  - router: output queues
  - end-system: de-jitter buffers

× Higher end-to-end delay

× Complex in practise
  - may require time synchronisation at routers

We have defined a work-conserving scheduler as one that is never idle if there are packets waiting. So what is a non-work conserving scheduler and why might they be useful? In a non-work-conserving scheduler, the scheduler can be idle even though there are packets waiting for transmission. The reason it remains idle is that waits for packets to become eligible for transmission. It is a bit like a clever set of traffic lights at a junction – not only does it know where you are going, it also knows how fast you should be going and only lets you move on when the time is right, even if there is no other traffic going through at other junctions. (Continuing the analogy, a work-conserving discipline is more like a junction with stop signs – as soon as there is a chance for you to go through, then you do.)

The eligibility of a packet can be determined in different ways. One way is to ensure that it always spends no more than a fixed time at a router. In this way the end-to-end jitter is controlled. This is equivalent to control the data rate and the jitter, and so such mechanisms are called **delay-jitter** regulators. Another way to determine eligibility is establish the notion of a fixed end-to-end delay for a packet. Then, routers decide a packet is eligible based on the time budget that remains for each packet. This is called a **delay-jitter** regulator as both the end-to-end delay and the end-to-end jitter are controlled.

The advantages of non-work conserving disciplines is that the reduce jitter, making downstream traffic more predictable. This means that there are fewer and smaller traffic burst, which reduces the buffering requirements at router and end-systems. However, we have already discussed that end-systems can de-jitter streams as required.

However, the cost of using such mechanisms is higher end-to-end delay overall. Also, such systems can be complex to build in practise. For example, delay-jitter regulators require that the network operator know the propagation on each link in the network, and that all router maintain a synchronised clock with which they can establish the eligibility of packets. Consequently, non-work-conserving disciplines are not used are are still considered a research issue.

Scheduling: requirements

- Ease of implementation:
  - simple → fast
  - high-speed networks
  - low complexity/state
  - implementation in hardware
- Fairness and protection:
  - local fairness: **max-min**
  - local fairness → global fairness
  - protect any flow from the (mis)behaviour of any other
- Performance bounds:
  - per-flow bounds
  - deterministic (guaranteed)
  - statistical/probabilistic
  - data rate, delay, jitter, loss
- Admission control:
  - (if required)
  - should be easy to implement
  - should be efficient in use

DigiComm II

There are four key requirements that can be listed for any scheduling mechanism.

• **ease of implementation:** the mechanisms should be easy to implement. Keeping the mechanism as simple as possible makes it faster, it should be easier to implement. Also, if we can keep the amount of algorithmic complexity low, as well as minimise the amount of state information needed by the algorithm, then the mechanisms may lend itself more easily to implementation in hardware, making it suitable for high-speed networking.

• **fairness and protection:** flows should receive fair allocation of resource, all other things being equal, they should not receive any less than any other flow also asking to use the same resources. In fact, it is possible to be much more precise about how resources should be allocated, using the **max-min fairness** criteria. The max-min fairness criteria allows per-hop local fairness for each flow, which results in global fairness for all of the flows. The protection requirement states that no flow should suffer due to the (mis)behaviour or characteristics of any other flow. We can see from the Conservation Law that FCFS does not provide protection as if one flow was to increase its data rate, all the other flows would suffer.

• **performance bounds:** for supporting QoS in networks, it should be possible to specify performance bounds to describe so that we can be sure that our flow is handled appropriately by the network. The exact nature of these performance bounds depends on the kind of QoS guarantee that we need for our flow. Where a scheduler is being used to provide a guaranteed service, it would have to offer the ability to work with deterministic performance bounds. If relative priorities for flows were being specified, then the scheduler should allow the specification of performance bounds in a statistical or probabilistic way.

• **admission control:** where deterministic performance bounds are specified, the network may need to perform admission control before it can allow a flow to start transmission. In this case the admission control mechanism should be easy to implement and efficient to operate.

The max-min fair share criteria

- Flows are allocated resource in order of increasing demand
- Flows get no more than they need
- Flows which have not been allocated as they demand get an equal share of the available resource
- Weighted max-min fair share possible
- If max-min fair → provides protection

$$m_n = \min(x_n, M_n) \quad 1 \le n \le N$$

$$M_n = \frac{C - \sum_{i=1}^{n-1} m_i}{N - n + 1}$$

C : capacity of resource (maximum resource)

m_n : actual resource allocation to flow n

x_n : resource demand by flow $n, x_1 \le x_2 \cdots \le x_N$

M_n : resource available to flow n

Example:
C = 10, four flow with demands of 2, 2.6, 4, 5
actual resource allocations are 2, 2.6, 2.7, 2.7

DigiComm II

In **max-min fair share**, resources are allocated according to some simple rules. The demands of the resource flows are ordered in increasing demand so that flows with the lowest demands are allocated resources first. This means that resources with small demands are likely to be allocated all that they ask for (as the simple example above illustrates.)

It is possible to assign weights to the resource demands so that some demands receive a greater share of the resources than others. In this case th rules ofr max-min fair share are modified as follows:

• flows are allocated resource in order of increasing **weighted** demand (the demands are normalised by the weight)

• no flow gets more than it needs

• flows which have not been allocated as they demand get a **weighted** share of the available resource

Scheduling: dimensions

- Priority levels:
  - how many levels?
  - higher priority queues services first
  - can cause starvation lower priority queues
- Work-conserving or not:
  - must decide if delay/jitter control required
  - is cost of implementation of delay/jitter control in network acceptable?
- Degree of aggregation:
  - flow granularity
  - per application flow?
  - per user?
  - per end-system?
  - cost vs. control
- Servicing within a queue:
  - "FCFS" within queue?
  - check for other parameters?
  - added processing overhead
  - queue management

DigiComm II

When designing a scheduling mechanism, we have several **dimensions** or **degrees of freedom** in which we can work.

The first of these is the use of **priority levels**. Here we assign different priorities to different output queues. Queues with higher priorities receive service first. Queues with lower priorities are only serviced when the higher priorities queues are empty. We can decide to have an arbitrary number of priority levels to reflect our QoS policy.

We can also choose for our scheduling mechanism to be **work-conserving** or **non-work-conserving**. We have already discussed the differences between these two, and the decision boils down to two questions:

• do you need to have good delay and jitter control?

• if so, is the additional cost of implementing non-work-conserving schedulers acceptable?

When we design our scheduling policy, we must decide on the granularity of our flows, i.e. we must decide on the level of aggregation of traffic that we want. We can have large levels of **aggregation** and have relatively poor control over individual application-level flows, or we can have per-application flow state and have very fine grained control. The choice is a trade-off between the amount of control we have for our flows (with respect to individual packet sources) versus the amount of state we store for the scheduler and the processing overhead in involved with that state.

Finally, within an individual queue, we need to decide on how packets should be **serviced within that queue**. We can just decide to service the packets in the order in which they arrived within the queue (i.e. treat each queue as a single FCFS queue waiting fro transmission) or we can examine the contents of individual packets in the queue and make fine-grained policy decisions as to which should be serviced first. Again, the trade-off is between the level of control desired and the cost of the processing required. Queue servicing may be required for queue management, which we look at later.

Simple priority queuing

- K queues:
  - $1 \le k \le K$
  - queue $k + 1$ has greater priority than queue k
  - higher priority queues serviced first
- ✓ Very simple to implement
- ✓ Low processing overhead
- Relative priority:
  - no deterministic performance bounds
- ✗ Fairness and protection:
  - not max-min fair: starvation of low priority queues

DigiComm II

In priority queuing, K queues are established, numbered 1 … K. Higher numbered queues are serviced before lower numbered queues – they have higher priority. A lower priority queue will only be serviced if there are no packets awaiting service in a higher priority queue. This means that busy higher priority queues could prevent lower priority queues from being serviced – situation known as starvation. This system would need to be used with some other mechanism to police the traffic into the queues, e.g. a token bucket filter for each queue, plus admission control at the edge of the network.

Note that the queues assign relative priority, so this mechanism by itself would not be suitable for providing deterministic guarantees for performance bounds.

Generalised processor sharing (GPS)

- Work-conserving
- Provides max-min fair share
- Can provide weighted max-min fair share
- Not implementable:
  - used as a reference for comparing other schedulers
  - serves an infinitesimally small amount of data from flow *i*
- Visits flows round-robin

$\phi(n) \quad 1 \le n \le N$

$S(i,\tau,t) \quad 1 \le i \le N$

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

$\phi(n)$: weight given to flow n

$S(i,\tau,t)$: service to flow i in interval $[\tau,t$

flow i has a non – empty queue

DigiComm II

Generalised processor sharing (GPS) is a work-conserving scheduler that provide max-min fairness and protection. It can be used to provide probabilistic/statistical (relative) as well as deterministic performance bounds. Unfortunately, it is not implementable as the analysis requires that for each flow *i*, only an infinitesimally small amount of data is served. However, the analysis is useful in that it provides us with a reference with which we can assess other schedulers by comparison. All flows are serviced round-robin. GPS is described in [GP93] and [GP94].

GPS – relative and absolute fairness

- Use fairness bound to evaluate GPS emulations (GPS-like schedulers)
- Relative fairness bound:
  - fairness of scheduler with respect to other flows it is servicing
- Absolute fairness bound:
  - fairness of scheduler compared to GPS for the same flow

$$RFB = \left| \frac{S(i,\tau,t)}{g(i)} - \frac{S(j,\tau,t)}{g(j)} \right|$$

$$AFB = \left| \frac{S(i,\tau,t)}{g(i)} - \frac{G(i,\tau,t)}{g(i)} \right|$$

$S(i,\tau,t)$: actual service for flow i in $[\tau,t]$

$G(i,\tau,t)$: GPS service for flow i in $[\tau,t]$

$g(i) = \min\{g(i,1),\cdots,g(i,K)\}$

$$g(i,k) = \frac{\phi(i,k)r(k)}{\sum_{j=1}^{N}\phi(j,k)}$$

$\phi(i,k)$: weight given to flow i at router k

$r(k)$: service rate of router k

$1 \le i \le N$ flow number

$1 \le k \le K$ router number

DigiComm II

In [Gol94] is described two fairness bounds that can be used to assess the performance of the service, *S*, provides by an emulation of GPS (i.e. a scheduler that attempts to achieve a GPS-like service). The first fairness bound is the **relative fairness bound (RFB)**, which attempts to provide evaluate how fairly resources are allocated between flows being serviced by the GPS emulation. The second fairness bound is the **absolute fairness bound (AFB)**, which tries to make a comparison of the service provided by the GPS emulation to that which would be provided by GPS. The closer that the RFB and AFB are to zero the better. It is normally hard to obtain an evaluation of the AFB fro most emulations, so the RFB is used.

In the expressions above, it is assumed that we examine the *K* routers along a path that serve *N* flows. *S*(.) is the service given to a flow by the GPS emulation and *G*(.) is the service that would be given by GPS. g(*i*, *k*) is the relative service rate given to flow *i* at router *k* along the path. g(*i*) is, effectively, the bottleneck rate along the path.

Weighted round-robin (WRR)

- Simplest attempt at GPS
- Queues visited round-robin in proportion to weights assigned
- Different means packet sizes:
  - weight divided by mean packet size for each queue
- Mean packets size unpredictable:
  - may cause unfairness
- Service is fair over long timescales:
  - must have more than one visit to each flow/queue
  - short-lived flows?
  - small weights?
  - large number of flows?

DigiComm II

The simplest emulation of GPS is **weighted round-robin (WRR)**. Here, queues are serviced round-robin, in proportion to a weight that is assigned to each flow/queue, i.e. flows/queues with a greater weight have more service. In each round each queue is visited once. Normally, at least one packet is transmitted from each queue that is non-empty. The weight assigned to each flow/queue is normalised by dividing the by the average packet size for each flow/queue. If this is not know before hand, them WRR may not be able to offer the correct service rate for the flow. For many applications, it is not easy to evaluate the average packet size before hand. The service provided by WRR can be shown to be fair over long-lived flows but for short lived flows, flows with small weights or where there are a large number of flows, WRR may exhibit unfairness.

Deficit round-robin (DRR)

- DRR does not need to know mean packet size
- Each queue has deficit counter (dc): initially zero
- Scheduler attempts to serve one quantum of data from a non-empty queue:
  - packet at head served if size ≤ quantum + dc
    dc ← quantum + dc – size
  - else dc += quantum
- Queues not served during round build up "credits":
  - only non-empty queues
- Quantum normally set to max expected packet size:
  - ensures one packet per round, per non-empty queue
- RFB: 3T/r (T = max pkt service time, r = link rate)
- Works best for:
  - small packet size
  - small number of flows

DigiComm II

In **deficit round-robin (DRR)**, an attempt is made to by-pass the requirement to know the average packet size for each flow. Each non-empty queue initially has a deficit counter (dc) which starts of at the value zero. As the scheduler visits each non-empty queue, it compares the packet at the head each of these queues and tries to serve one quantum of data. If dc is zero for a queue, then the packet is served if the size of the packet at the head of the queue is less than or equal to the quantum size. If a packet can not be served, then the value of the quantum is added to the dc for that queue. If the scheduler visits a queue with a dc > 0, the a packet from the queue is served if quantum + dc is greater than or equal to the size of the packet at the head of the queue. If the scheduler sees a queue with a non-zero dc, yet it is empty, then the dc is reset to zero so that it can not keep acquiring deficit. The quantum size used is normally that largest expected packet size in the network. Example, three flows A, B, C wit packets of size 600, 1000, 1400 bytes. The quantum size is set to 1000 bytes. In the first round:

• queue A: packet is served, dc = 1000 – 600 = 400

• queue B: packet is served, dc = 1000 – 1000 = 0

• queue C: packet is not served, dc = 1000

In the second round:

• queue A: no packet, dc set to zero

• queue B: no packet, dc set to zero

• queue C: packet is served, dc = 1000 + 1000 – 1400 = 600

In third round:

• queue A: not served

• queue B: not served

• queue C: no packet, dc set to zero

If weights are assigned to the queues, then the quantum size applied for each queue is multiplied by the assigned weight.

DRR works best for small packets sizes and a small number of flows, suffering similar unfairness behaviour to that for WRR.

Weighted Fair Queuing (WFQ) [1]

- Based on GPS:
  - GPS emulation to produce **finish-numbers** for packets in queue
  - Simplification: GPS emulation serves packets bit-by-bit round-robin
- **Finish-number**:
  - the time packet would have completed service under (bit-by-bit) GPS
  - packets tagged with finish-number
  - smallest finish-number across queues served first
- **Round-number**:
  - execution of round by bit-by-bit round-robin server
  - finish-number calculated from round number
- If queue is empty:
  - finish-number is: *number of bits in packet* + *round-number*
- If queue non-empty:
  - finish-number is: *highest current finish number for queue* + *number of bits in packet*

DigiComm II

WFQ is an emulation of GPS that actually uses a GPS computations to control its behaviour. In fact it runs a GPS computations and uses these to tag packets in flows to indicate the time the would have finished being served had they been subject to GPS scheduling. WFQ itself approximates GPS by using a model of bit-by-bit round robin service, i.e. for a packet of size N bits at the head of a queue, it will have complete service after the scheduler has performed N rounds of visiting all the other queues. This is bit-by-bit fair. However, we can not actually transmit packets one bit at a time from multiple queues, so instead the packets are tagged with a **finish-number**, which indicates when they would have completed service if served bit-by-bit. So, we can see that packets with the smallest finish number are the ones that should be transmitted first. Let us take the simple case that the queue is empty and a packet of size 50 bits arrives in the queue. As it is served bit-by-bit round robin, it will be serviced in 50 rounds. So its finish number will simply be the current round-number plus its own size in bits. So if the round number is 20, the finish-number for our packet is 70. If the queue is non-empty, then packets already in the queue will be served first. So the finish number of this new packet, which is at the back of the queue, will be the finish number for the packet in front of it plus its own size in bits. When the round-number is equal to (or greater than) the finish-number of the packet, it should be served immediately.

The closest analogy (at least that I can think of at this time!) is probably that of using a number system when you go to the delicatessen counter at the supermarket. All the people (packets) are of unit size, so this simplifies things. They are served one at a time (unit-by-unit) and people can appear at the delicatessen counter from any of a number of aisles in the supermarket. A person takes a piece of paper with a number on it from the from the counter. This is your finish-number and is one more than the person in front of you. It does not matter where you are physically, but when your number comes up on the big counter display (the round-number), it is your turn to be served.

Weighted Fair Queuing (WFQ) [2]

$$F(i,k,t) = \max\{F(i,k-1,t), R(t)\} + P(i,k,t)$$

$F(i,k,t)$: finish - number for packet k on flow i arriving at time t

$P(i,k,t)$: size of packet k on flow i arriving at time t

$R(t)$: round - number at time t

$$F_\phi(i,k,t) = \max\{F_\phi(i,k-1,t), R(t)\} + \frac{P(i,k,t)}{\phi(i)}$$

$\phi(i)$: weight given to flow i

- Rate of change of $R(t)$ depends on number of active flows (and their weights)
- As $R(t)$ changes, so packets will be served at different rates
- Flow completes (empty queue):
  - one less flow in round, so
  - R increases more quickly
  - so, more flows complete
  - R increases more quickly
  - etc. …
  - **iterated deletion** problem
- WFQ needs to evaluate R each time packet arrives or leaves:
  - processing overhead

DigiComm II

More formally, the evaluation of the finish-number, F, is as shown above.

The 'W' in WFQ comes from applying weights to evaluation of the finish-number, as shown in the expression for F_ϕ.

We can see here that the finish time is depended on R. Also, F and R are both functions of absolute time t. As the number of flows change, so does the rate of change of R. As more flows become active, so the rate of change of R decreases. As the number flows decreases, so the rate of change of R increases. This latter property leads to a problem know as **iterated deletion**. A flow is deleted if it has completed, i.e. the queue becomes empty. When this happens, it means that R will increase at a faster rate. This means that the R will approach the finish-number of other packets, already queued, faster than originally evaluated. This may cause some of the other flows to complete also, leading to a further increase in the rate of change of R. In order to keep track of R, a WFQ system must evaluate the value of R every time a packet arrives or leaves.

Weighted Fair Queuing (WFQ) [3]

- Buffer drop policy:
  - packet arrives at full queue
  - drop packets already in queued, in order of decreasing finish-number
- Can be used for:
  - best-effort queuing
  - providing guaranteed data rate and deterministic end-to-end delay
- WFQ used in “real world”
- Alternatives also available:
  - self-clocked fair-queuing (SCFQ)
  - worst-case fair weighted fair queuing (WF^2Q)

DigiComm II

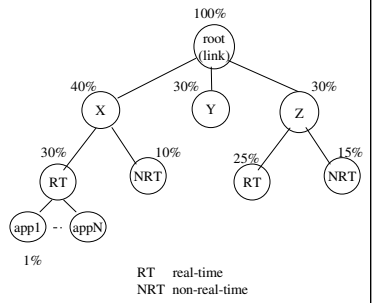
WFQ also proposes a a buffer drop policy that is max-min fair. If a packet arrives to a full queue, enough packets are dropped to allow the new packet to be queued. Packets are dropped in order of decreasing finish-number, i.e. most recent arrivals are dropped first.

WFQ can be used for scheduling multiple best effort flows, providing protection for each flow. With a few modifications – the weighted part of WFQ which we have not examined – it can also provide services for flows that require guaranteed data rate and an upper bound on the end-to-end delay.

Even though it is computationally expensive, and need to hold per-flow state, WFQ is used by many router manufacturers. Alternatives to WFQ, such as self-clocked fair-queuing (SCFQ) and worst-case fair weighted fair queuing (WF^2Q). SCFQ has similar properties to WFQ, except that it removes the expensive round-number computation but also has a higher end-to-end delay bound than WFQ. WF^2Q has an AFB that more closely approaches the AFB that is lower than WFQ (i.e. it has a a better AFB than WFQ), but requires an additional implementation complexity and has higher processing overhead in operation.

Class-Based Queuing

- Hierarchical link sharing:
  - link capacity is shared
  - class-based allocation
  - policy-based class selection
- Class hierarchy:
  - assign capacity/priority to each node
  - node can “borrow” any spare capacity from parent
  - fine-grained flows possible
- Note: this is a queuing mechanism: requires use of a scheduler



DigiComm II

CBQ was designed to allow sharing of link capacity within a class-based hierarchy [FJ95] [WGCJF95]. We see an example showing the link capacity as a root node in a tree at 100%. Organisations X, Y and Z that share the link are assigned 40%, 30% and 30% of the link capacity, respectively. Within their own allocations of capacity, the organisations can choose to partition the available capacity further by creating sub-classes within the tree. Organisation X decides to allocate 30% to real-time traffic and 10% to all non-real-time traffic. Within the real-time allocation, X decides to allocate capacity to individual applications. Organisation Z also divides its allocation into real-time and non-real-time, but with a different share of the available link capacity. Organisation Y decides not to further refine its allocation of link capacity. The percentages indicate the minimum share of the link capacity a node in the tree will receive. Child nodes effectively take capacity from their parent node allocation. If some sibling nodes are not using their full allocations, other siblings that might be overshooting their own allocation are allowed to “borrow” capacity by interacting with the parent node. With an appropriate scheduling mechanism, this allows support for QoS sensitive flows. Classifications could be made per application, per flow, per IP source address, etc., as dictated by the policy chosen by the individual organisations in conjunction with their Internet connectivity provider. Not shown above is that different priority levels can be assigned to each classification. For example, real-time traffic could be given priority 2 and non-real-time priority 1, where priority 2 is higher (receive better service) than priority 1.

Queue management and congestion control

DigiComm II

Queue management [1]

- Scheduling:
  - which output queue to visit
  - which packet to transmit from output queue
- Queue management:
  - ensuring buffers are available: memory management
  - organising packets within queue
  - **packet dropping when queue is full**
  - **congestion control**

DigiComm II

Schedulers are used to select which packet from which queue will receive service next. Some, such as WFQ may also suggest ways in which queues may be managed under overflow conditions, i.e when there is a scarcity of buffers. However, **queue management** is a topic in its own right, although it is often related closely to the operation of scheduling. Queue management may include several function, including memory management and organisation/ordering of packets within a queue. However the two functions of queue management that we will discuss are **packet dropping** and **congestion control**.

Queue management [2]

- Congestion:
  - misbehaving sources
  - source synchronisation
  - routing instability
  - network failure causing re-routing
  - congestion could hurt many flows: aggregation
- Drop packets:
  - drop “new” packets until queue clears?
  - admit new packets, drop existing packets in queue?

Congestion can occur for a range of reasons. Sources may be mis-behaving, generating traffic above their normal rate. Even when sources are all behaving, correctly, pathological occurrences such as source synchronisation may cause congestion. Routing instability or network path changes due to network failure close to a node may cause a temporary burst of congestion. The result of congestion could hurt many flows, and this is more so as you move towards the core of the network and find a higher aggregation of traffic.

When congestion occurs, the router has run out of buffers and must drop packets. However, which policy should the router adopt in dropping packets, and what are the effects of using one particular dropping policy over another?

Packet dropping policies

- Drop-from-tail:
  - easy to implement
  - delayed packets at within queue may “expire”
- Drop-from-head:
  - old packets purged first
  - good for real time
  - better for TCP
- Random drop:
  - fair if all sources behaving
  - misbehaving sources more heavily penalised
- Flush queue:
  - drop all packets in queue
  - simple
  - flows should back-off
  - inefficient
- Intelligent drop:
  - based on level 4 information
  - may need a lot of state information
  - should be fairer

The granularity of the drop algorithm has the same choices as for scheduling. We can decide to drop packets from individual application-level flows, or have a courser grain of dropping. With a fine granularity, the cost, as always is the additional state to be held at the router and the overhead of maintaining and processing that state information. With a courser granularity, we have reduce the amount of state information and lose some control over which flow’s packets are dropped. If packets are queued per-flow, then we also offer some protection as a mis-behaving source will only fill and overflow its own queue and not hurt any other flows.

We can decide to drop form the tail of the queue. This is easiest to implement, as it means that there is no need to adjust any pointers in any other part of the queue. However, packets already in the queue and already delayed, and their usefulness may expire while they wait in the queue, i.e. they get to the receiver too late to be used by the application. Yet the receiver may not become aware of the congestion until it sees dropped packets, i.e. until the drop packets at the end of the queue are not received.

So an alternative is drop form the head of the queue. This means that older packets – packets that have been in the network longer – are purged. This is good for real-time data, as theses packets would no longer be as useful anyway. This is also good for protocols like TCP, as the loss is seen sooner and so TCP’s congestion control/avoidance mechanisms can operate. However drop from head typically has a higher processing overhead than drop from tail.

We can also drop a packet from a random place in the queue. This should be fair if all sources are behaving. If there are any mis-behaving sources, they are likely to be occupying more buffers and so are more likely have packets dropped than other flows, which is also beneficial.

Another very simple policy to implement is dropping a whole queue. This has the advantage that frees up buffers immediately, and this could be useful if the router is experiencing extreme congestion. However, this policy could be inefficient, as it may lead to many more re-transmission than are necessary, and may compound the burstiness of traffic. The exact behaviour of sources when such an event occurs may be application specific. For real-time applications, to have such a large contiguous hole in the flow may not be acceptable. This may be useful for “reprimanding” mis-behaving flows, when per-flow queuing is used.

End system reaction to packet drops

- Non-real-time – TCP:
  - packet drop → congestion → slow down transmission
  - slow start → congestion avoidance
  - network is happy!
- Real-time – UDP:
  - packet drop → fill-in at receiver → ??
  - application-level congestion control required
  - flow data rate adaptation not be suited to audio/video?
  - real-time flows may not adapt → hurts adaptive flows
- Queue management could protect adaptive flows:
  - smart queue management required

DigiComm II

We are considering packet dropping to control congestion. However, what do applications do when they see packet loss? TCP uses loss as an indicator of congestion, and slows down its transmission rate. TCP has well-defined behaviour in the form slow-start and congestion avoidance, and generally does a reasonable job of being kind to the network.

Real-time applications, however, use UDP, which has now congestion control mechanisms. The packet loss may be seen at the receiver and he receiver may perform some application-specific action, but there is unlikely to be any feedback to the sender. Some form of application-level congestion control is required. This may require, for example, adapting the flow rate by hanging audio or video coding. However, the application may not e suited to such adaptation. yet, without such adaptations, congestion events may continue to occur. Indeed, adaptive flows (such as TCP) that do back down in the face of congestion my be forced down to low data rates as unconstrained, non-adaptive real-time flows continue unabated.

However, smart queue management (coupled with an appropriate scheduling mechanism) could be used to protect adaptive flows from is-behaving flows whilst at the same time controlling congestion.

RED [1]

- Random Early Detection:
  - spot congestion before it happens
  - drop packet → pre-emptive congestion signal
  - source slows down
  - prevents real congestion
- Which packets to drop?
  - monitor flows
  - cost in state and processing overhead vs. overall performance of the network

DigiComm II

One widely implemented intelligent drop mechanism is Random Early Detection (RED) [#FJ93#]. This drops packets before congestion actually occurs by monitoring queue lengths at a router and increasing the probability of a packet drop as the queue length builds up. The drop of a packet from a flow will act as an indication of congestion and cause it to back of is transmission rate. The packet drop occurs before real congestion starts and is intended to prevent real congestion occurring.

The cost of this mechanism is the requirement to monitor flows and queue lengths, and the processing overhead of dealing with the state information being held. This increase in performance cost is seen as acceptable as the overall performance of the network is improved by avoiding real congestion.

RED [2]

- Probability of packet drop ∝ queue length
- Queue length value – exponential average:
  - smooths reaction to small bursts
  - punishes sustained heavy traffic
- Packets can be dropped or marked as "offending":
  - RED-aware routers more likely to drop offending packets
- Source must be adaptive:
  - OK for TCP
  - real-time traffic → UDP ?

DigiComm II

RED drops a packet from a flow with probability, *p*, which increase linearly as the queue length increases. However, the actual value of queue length used to evaluate *p* is an exponentially-weighted moving average (EWMA) of the actual queue length. This is to allow small bursts of traffic to be allowed through the network, but to spot sustained, heavy traffic. The dampening effect of the EWMA also helps to stabilise the dropping mechanism, so it does not "over-react" to short-lived bursty traffic. As well as dropping packets, RED can mark IP packets as "offending", i.e. in violation of some traffic profile, and not drop them when traffic load is light. However, offending packets can be spotted by RED-aware routers downstream and can sill be dropped if congestion appears.

While RED can control queue length regardless of end-system co-operation, as we have noted already, if the end-system does reduce its transmission rate in response to such congestion signals, this would be helpful to the health of the network as a whole. We have also noted that while dynamically adaptive behaviour does not harm non-real-time applications, real-time applications, such as voice and video applications, normally do not find such behaviour acceptable.

TCP-like adaptation for real-time flows

- Mechanisms like RED require adaptive sources
- How to indicate congestion?
  - packet drop – OK for TCP
  - packet drop – hurts real-time flows
  - use ECN?
- Adaptation mechanisms:
  - layered audio/video codecs
  - TCP is unicast: real-time can be multicast

DigiComm II

In recognition of the usefulness of mechanisms like RED, and the requirement for congestion control in real-time end-systems, there are several proposals to deal with congestions.

The first one we shall examine how to indicate congestion to end-systems. The lost packet is an effective indicator of congestion to TCP, but real-time flows may not be monitoring for packet loss in the same way. There is a proposal to provide **explicit congestion notification (ECN)** [RFC2481] at the IP level. This would be a single-bit flag which would normally be clear, but which would be set by a congested router and so act as an explicit signal to a receiver that this packet has passed through a part of the network that is experiencing congestion.

Other, ore complex mechanisms concentrate on building congestion control into the application-layer protocol. We have already stated that real-time flows such as video do not normally adapt well. However, by using **layered-codecs** we can adjust the transmission rate dynamically. A layered video codec encodes video as several separate flows, each flowing carrying different frequency (spatial) components. There is a single base-layer which must be received, but several additional layers just adder finer detail to the base layer, and can be received optionally depending on the availability of network resources. In [VRC98] is described an experiment to provide TCP-like adaptation for video.

Scheduling and queue management: Discussion

- Fairness and protection:
  - queue overflow
  - congestion feedback from router: packet drop?
- Scalability:
  - granularity of flow
  - speed of operation
- Flow adaptation:
  - non-real time: TCP
  - real-time?
- Aggregation:
  - granularity of control
  - granularity of service
  - amount of router state
  - lack of protection
- Signalling:
  - set-up of router state
  - inform router about a flow
  - explicit congestion notification?

DigiComm II

A good analysis of requirements for the future Internet is given in [She95].

Summary

- Scheduling mechanisms
  - work-conserving vs. non-work-conserving
- Scheduling requirements
- Scheduling dimensions
- Queue management
- Congestion control

DigiComm II