Paternoster policing and scheduling

Mick Seaman

Paternoster is a simple real-time packet bandwidth reservation, policing, queuing, and transmission scheduling algorithm that provides bounded end-to-end delays without requiring clock synchronization between network nodes[1]. This updated note includes revised delay bounds (previously overstated) and describes its use with shared media and aggregate flows.

---

1. Introduction

The paternoster real-time forwarding algorithm provides bounded delays across the network and lossless service for flows that conform to their reservations. The basic algorithm uses four output queues per class of service per port, and one active counter for each flow (for pseudo-code see Annex A). Best effort traffic can make use of any remaining bandwidth (either unreserved or not currently used), with a relaxed upper delay bound (before discard).

Relationship to other algorithms

The algorithm can also be described as an improvement on the peristaltic shaper (CQF), with the node to node synchronization requirement removed; as deadline oriented, with not before/not after attributes for forwarded packets; or indeed as an improved credit based shaper.

Epochs and reservations

The ports that transmit and receive frames between the nodes in any particular network are all configured to use a basic epoch duration[2] τ, though no attempt is made to synchronize epochs for different ports. A reservation ρ_i[3] for flow i allows the source port, s_i, of that flow to transmit frames containing up to ρ_i octets in each successive epoch[4], without constraining when in each epoch frames are transmitted.

Queues, reception and transmission

Each port has four queues, each identified as the *prior*, *current*, *next,* or *last* queue for a given epoch. The port transmits frames from the *prior* queue (while any remain enqueued) and then from the *current* queue.

Frames received for flow i in an epoch that are to be transmitted through the port are added to the *current* queue, as long as those cumulative additions do not exceed ρ_i. They are then added to the *next* queue, and finally to the *last* queue (with the same ρ_i limits). Any further frames received in the epoch are discarded.

When a new epoch begins, the *current* queue becomes the *prior* queue, the *next* and *last* queues (and the remainder of their initial ρ_i reservations) become *current*, and *next* respectively. The former *prior* queue should be empty (if not, there has been a reservation or transmission error, and the queue is purged), is given a fresh reservation, and becomes the new *last* queue.

Best case forwarding

Figure 1 shows *Alice* transmitting frames (1, 2, 3, …) for a given flow at regular intervals, in each of her epochs (delineated by |). Each frame experiences a constant delay en route to *Bob* and is added to the current queue (*c* in the figure) for each of his epochs before being transmitted from that queue to *Charlie*, again with constant delay.

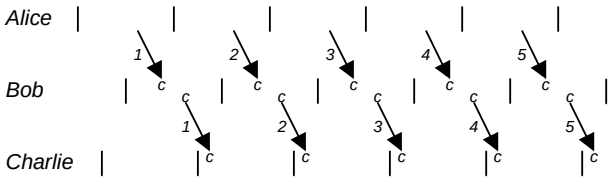

Figure 1—Best case forwarding (constant delays)

Figure 2 shows the same information with the timescales shifted by the constant transit delay between each pair of participants.

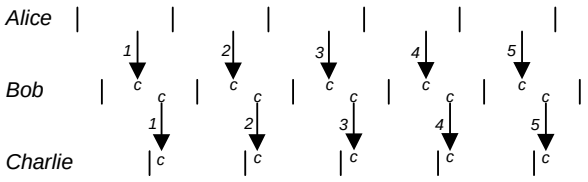

Figure 2—Best case, constant delay, time shifted

---

[1]The paternoster algorithm originated in a network delay analysis and a request from John Messenger at the January 2017 interim for a protocol description that met specific criteria, though we did not discuss this algorithm. Paternosters are described in https://en.wikipedia.org/wiki/Paternoster.

[2]The following symbol, tau, seems appropriate. https://en.wikipedia.org/wiki/Tau.

[3] "rho"$_i$. The shorthand "ρ_i transmission" (or reception) refers to frames for flow i containing up to ρ_i octets. "ρ_i frames" are frames containing ρ_i octets.

[4]As described in this note, the reservation includes any per frame overhead that is consistent throughout the network. The possibility of making a reservation for a flow passing through any given port has to take into account the possibility of headers/tags being been added to that frame (and to the frames of existing reservations) and any per frame overhead particular to that port. An awareness of the flow's packet size distribution can help maximize link utilization.

Transmission order not overconstrained

The queues do not have to be serviced as pure FIFOs, provided that the transmission selection used provides the reserved bandwidth to any frames eligible for transmission in an epoch, and the *prior* queue is emptied before any frames are taken from the *current* queue.[5] If one or more packet can be transmitted, then one of them should be.

Worst case flow interference

Paternoster makes no assumptions about the relative speeds of links and fan-in (how many links might contribute frames to a flow on a single outbound link). All the frames that could be received for a given outbound link might arrive at the same time, or with any other inconvenient timing. Delays caused by interfering flows are not predictable, except for the fact they do not compromise the delay bound. Consider, for example, a flow passing through *Alice*, *Bob*, and *Charlie*, top to bottom in Figure 3 .

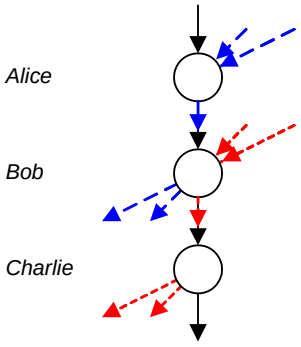

Figure 3—Cross-flow interference

Frames for some other flows sharing the *Alice-Bob* link might arrive at *Alice* and depart at *Bob* (as shown in blue), while others (in red) share the *Bob-Charlie* link. If these cross-flows each comprise a single frame per epoch, each arrive on separate links, and collectively dominate the bandwidth on the shared links, frames for our victim flow (top to bottom) can be queued for transmission at any time within their selected epoch. The delay variation for one hop can, with the interfering cross-flows as shown, be independent of the delay on the next.[6]

Bunching and queue draining

Alice could transmit frames for a given reservation towards the end of one epoch and towards the beginning of the next. *Bob* might then receive both (sets of) frames[7] in a single epoch, adding the first to the *current* queue and the second to the *next* as in the two scenarios shown in Figure 4.

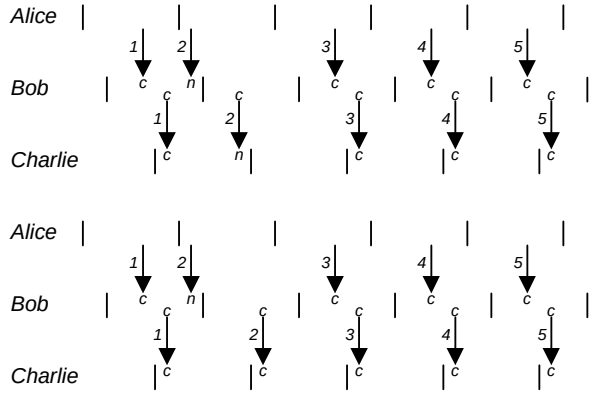

Figure 4—Variable transmission timing

Because frames for a flow i can arrive and be added to the *current* queue at the end of an epoch, paternoster permits the transmission of up to $2\rho_i$ octets of that flow in an epoch: i.e. all the frames in prior epoch's *current* queue (now labelled *prior*) in addition to any frames received in the present epoch for which bandwidth is available. See Figure 5.

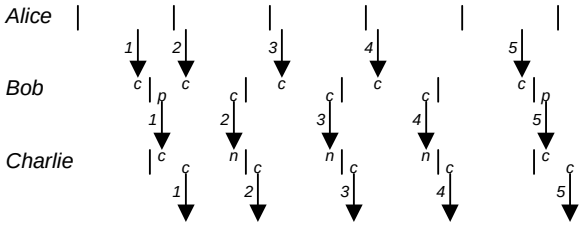

Figure 5—Bunching

The bandwidth allocated for any service class should exceed the sum of the reservations for that class by at least one maximum sized packet (for that class) per epoch. Thus, in any epoch, a completely full *prior* queue can be transmitted and an opportunity provided to transmit at least one frame from the *current* queue.[8]

[5]FIFO transmission is required for aggregate flows, see later.

[6]Given more information about other flows the delay variance, and indeed the end to end delay bound for the victim flow, can be reduced, but that would run counter to the spirit of paternoster (simple calculation without the assuming overall knowledge of flow patterns) and tend towards Asynchronous Traffic Shaping. In the latter's terms paternoster assumes a worst case multiplexing delay at each hop.

[7] The figures show selected packets and their timing relative to each node's epochs. They should not be taken to imply that reservations accommodate only one or two packets per epoch—other packets with similar timing (or timing irrelevant to the point being made) may have been transmitted.

[8]If the *current* queue is empty at the start of the epoch it is not necessarily the case that one of its packets will be transmitted, even if it is not empty at the end of the epoch. However if the *current* queue (the prior epoch's *next* queue) is not empty at epoch start it will contain at least one packet less than its reservation permits by epoch end, thus allowing at least two packets to be taken from the *next* queue (if it contains those packets at epoch end) when it becomes the *current* queue in the following epoch. Thus the queues will progressively drain, even if the current queue is replenished a steady rate, up to the point where intervals of no reception allows following bunched reception to occur again.

In Figure 5, frames 2, 3, and 4 are received into *Charlie*'s *next* queue, with reception into *Charlie*'s *current* queue only when *Alice's* in-epoch transmission timing of frame 5 reverts to that used for frame 1. If *Alice* persists with the timing used for frame 4, the delay associated with *Charlie* receiving into *next* and transmitting in the following epoch will also persist, and the effect of allowing *Bob* to transmit from both *prior* and *current* queues in a single epoch has been to move the delay from *Bob* to *Charlie*. That persistent delay is a consequence of the general rule that a flow will not drain from network queues if the applied load matches the network service rate. ρ_i is an upper bound for Alice's transmission of flow *i* in an epoch, not a desirable operating point.

Alice could decide to transmit one less frame than permitted in each epoch. However, end-to-end delay is a critical parameter for applications requiring bounded delay and is a multiple of τ, so the latter may be chosen to accommodate transmission of a just one frame from each of a number of participants. Doubling ρ_i to facilitate queue draining would double delay bounds and halve the available bandwidth. A flow source does better by using a slightly longer epoch: with $\tau_{si} = 1.25\tau$ and a single frame per epoch, a network node can reduce any backlog every fifth epoch (on average). See Figure 6.

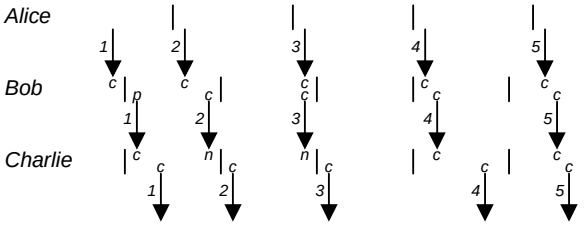

Figure 6—Queue draining

Charlie does not add a frame to his next queue in the epoch following the reception of frame 3, so the following frame 4 is added to the current queue (and is eligible for transmission) in the epoch after that.

Transmit bursts

If frames containing $2\rho_i$ octets for flow *i* are transmitted in a single epoch (transmission of a full *prior* queue, holding frames all received into the *current* queue of the previous epoch, plus complete use of the present epoch's *current* queue), then at most ρ_i octets will have been transmitted in the immediately prior epoch (from the *prior* queue for that epoch) and at most ρ_i octets can be transmitted in the following epoch (because the present epoch's *current* queue will be exhausted).[9] See *Bob's* transmissions 1 and 2 in Figure 5.

Receive bursts

A receiver (whose epochs are assumed not to be aligned with the transmitters passing it packets) can thus receive, in one of its own epochs, at most $3\rho_i$ octets for flow *i* ($2\rho_i$ towards the end of one of the transmitter's epochs and ρ_i from the beginning of the next, or ρ_i from the end of one and $2\rho_i$ from the beginning of the next). Those frames are added to the *current*, *next*, and *last* queues. See Figure 7.

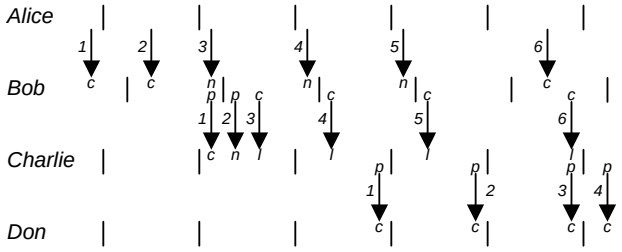

Figure 7—A receive burst

While a frame can spend up to four epochs at a node[10] (e.g. frame 3 at *Charlie* in Figure 7) that is only possible if prior frames have been delayed and the frame in question has caught up (concertinaed) with them. The delay expands the burst, resulting in less delay at subsequent nodes. Alternatively the burst can propagate, with the later packets spending less time at the intervening node (e.g. propagating from *Charlie* to *Don* in Figure 8, with frame 3 spending at most two epochs at *Charlie*).

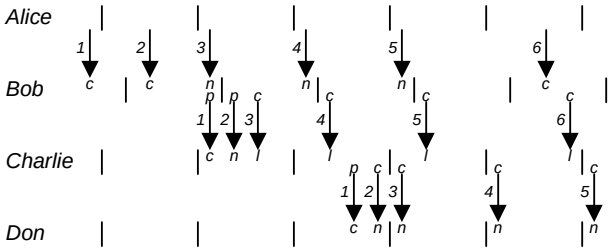

Figure 8—A receive burst (2)

[9]Because transmitting more than one epoch's worth of traffic in one of the transmitter's epochs depends on carrying over traffic from the prior epoch, once a transmitter has transmitted two epoch's worth in a single epoch it cannot do so again until it has accumulated an epoch's worth of backlog. The scenario 2|1|2 is not possible.

[10]Less the allowance made for queue draining (previously described) and for variable transit delays (see later). The fact that *Charlie* does not transmit anything in the epoch in which he receives frames 1, 2, and 3 may require explanation. These may have all been received too late to be transmitted in that epoch (frames for other flows may also have been received after the start of the epoch, but before these frames, and are taking the bandwidth). Frames in the *prior* queue at the start of an epoch are guaranteed to be transmitted by the end of the epoch, other frames can be delayed to a following epoch.

Buffering requirements

The buffering at a node, for a service class, is bounded by the total per epoch reservation for that service class, multiplied by the number of per epoch queues (four—*prior*, *current*, *next*, and *last*).[11] This may seem high, but is a consequence of the worst case receive assumptions described above.

Transit delay variations

If the participant to participant transit delay varies then the frames of two transmission epochs can be received in a single epoch, even if there is no variation in initiating transmission. See Figure 9.

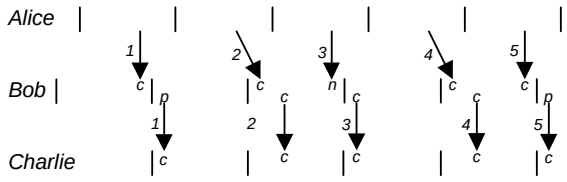

Figure 9—Variable transit delay

If transit delay is measured from a transmitter's acquisition of the last octet of a frame to be forwarded to the recipient's acquisition of that last octet[12], then variations in packet size imply variations in transit delay. Preemption can also significantly increase the on-the wire transit time, and transit time variability, of a preempted frame. The receiving node can contribute a variable delay if, for example, the assignment of a received frame to an epoch queue is only made once that frame has transited an internal forwarding fabric.

However, if the time to transmit a full *prior* queue, plus the transit delay variation, plus any difference between participants' epoch durations does not exceed τ, then extra queues are not required: four epoch's reservations cannot be received in a single epoch (in Figure 8 frames 1, 2, and 3 from *Bob* are not delayed to the extent that *Charlie* receives them in the same epoch as frame 4). Any bandwidth given up to satisfy this constraint can be used by best effort traffic.

End-to-end delay bounds

While the maximum frame delay at a single node is node is 4τ, this delay cannot be encountered at every node (see discussion of Figure 8). The worst case end-to-end delay can be bounded without attempting to enumerate every possible forwarding pattern by observing that earlier arrival of any given frame at any node will not lead to its later departure. We can, therefore, bound delays based on those experienced by a set of frames spaced out for easy analysis. See Figure 10.

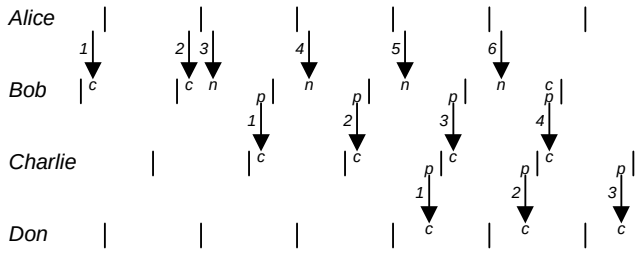

Figure 10—End-to-end delay

An initial frame 1, transmitted by *Alice* at the end of an epoch, arrives at each subsequent node just after (arbitrarily close to) the beginning of an epoch, is received into the *current* queue for that node, and is transmitted from the *prior* queue just before (arbitrarily close to) the end of the following epoch. Ignoring the constant portion of the node to node transit delay[13], this delays the frame by 2τ for each hop internal to the network, a total of $(2h - 1)\tau$ (where h is the number of hops from source to destination). A second frame, transmitted τ after the first, experiences the same delay. *Alice* transmits frame 3 at the beginning of an epoch, arbitrarily soon after frame 2. That frame will experience a 3τ delay at Bob, and then 2τ at each subsequent node for an end-to-end total of $2h\tau$. Frames 4 and 5, transmitted at *Alice's* earliest following opportunities also experience an end-to-end delay of (at most) $2h\tau$, our end-to-end bound. While detailed analysis of over-provisioning, fan-in, transmission speeds, and packet sizes can reduce this bound, the attractiveness of a good simple scheme is not having to do that analysis.

Class of service epochs

The duration of an epoch, τ, does not have to be the same for each class of service (though must be consistent network wide). If differing epochs are used they are arranged and used in a way that ensures each does provide the requisite bandwidth for each class of service in each epoch. One possibility is to use strict priority transmission selection, with lower priority classes using a period of twice the duration of the higher priority classes and an epoch start that is aligned with that of alternate high priority epochs. In this arrangement the amount of bandwidth that the higher priority classes can take from that available to those of lower priorities is consistent for each of the

[11] See later for circumstances when the use of additional epochs (and hence the use of additional queues) might be warranted.

[12] A useful measure for present purposes since it measures the interval between the time at which the transmitter can check the FCS and make decisions on the frame, including initiating transmission, and the time at which the recipient can do likewise.

[13] Assuming that the variable portion is at its maximum, taking up all the slack provided for it.

latter's epochs (which would not be the case if the epoch starts were not aligned).

Additional epochs

Additional epochs, with *next_but_one*, and *next_but_two*, … queues before *last* can be used to accommodate network links with greater transit delay variation,[14] without the need to have a separate network-wide class of service (with increased τ) to carry traffic that transits (or might transit) those links. The additional queues are needed only in the nodes attached to those links. They provide play-out buffering to shape traffic entering regions of the network using the usual four queues.

Best effort

As described above, best effort frames can be transmitted at a strictly lower priority, filling in the transmit opportunities left by reserved traffic in any given epoch. The bandwidth remaining after fixed reservations should allow for at least one maximum sized best effort frame per epoch , so the transmission of a best effort frame that extends from the end of one epoch into the start of another does not violate the reserved bandwidth commitment for the latter.

The amount of best effort traffic already queued can also be compared with the spare bandwidth available for forthcoming epochs and further best effort packets dropped if their anticipated transmission time is too far into the future—effectively sizing the best effort queue to provide delay bounds. A multi-queue algorithm can be used to bound transmission delays without restricting the amount of bandwidth used in an epoch.

Comparison with peristaltic shaping

The peristaltic shaper (802.1Qch, Cyclic Queue and Forwarding) synchronizes the epochs used by bridges throughout the network and (in paternoster algorithm terms) queues each relayed frame for the *next* epoch and transmits only from the *current* epoch. The peristaltic shaper's worst case forwarding delay through a single bridge is 2τ. However the peristaltic shaper's synchronization means that the delay across a network of h hops is between $(h - 1)\tau+\delta$ and $(h + 1)\tau+\delta$, where δ is the forwarding delay through a single relay, ignoring the eventual transmitting port's queuing strategy. The paternoster algorithm's network delay will be between $h\delta$ and $2h\tau$, though the average delay is likely to be strongly weighted to the lower of these—if none of the inputs to the network vary each relayed frame will be queued and transmitted within the *current* epoch.

As compared with the peristaltic shaper then, the paternoster algorithm gives up some delay predictability in exchange for not requiring clock synchronization and for reducing the average delay. It should also be pointed out that the constraints on epoch duration τ are not the same for both algorithms. If the peristaltic shaper receives more than an epoch's permitted reservation within an epoch, the excess has to be discarded, whereas the paternoster algorithm can distribute the unevenly spaced input over two successive epochs, and can thus provide the same service with half the epoch duration. Against this has to be set the possible difficulty of making small reservations when using very short epochs.

Average end-to-end delays

More needs to be said about the potential benefit of reducing average delays. A recent but generally expressed view is that the users of time sensitive networks only care about the delay bound guarantee, and that any earlier delivery of any particular frame is irrelevant. In the short term, and for particular uses cases, this may be true: in simple control applications bounding the delay in a negative feedback loop is vital to stability; control theory is an intensely difficult subject, and forcing re-engineering of existing applications when introducing a new sensor network may be prohibitively expensive. However in the longer term focusing solely on the delay bound might causes us to miss significant opportunities. A delay bound requirement for satisfactory operation can be significantly tighter than that for stability limits.[15] Designing to that tighter bound may preclude the replacement of TDM small cell networks with packet networks in some applications. The speed with which a controlled task can be accomplished is often important, and can benefit from lower delays than those essential for maintaining control stability while it is being performed. Some control paradigms use models of the expected plant response to efferent copies of control inputs together with time compensated feedback, and we should be able to take advantage of those paradigms.

[14]Connecting local sites across provider networks, for example.

[15]The timing requirements for virtual reality applications serve as an example. Is it sufficient for the user to be just on the right side of throwing up throughout the whole performance, or is something much better 99.99% of the time desirable. Considerations of external factors point out other examples where wild excursions from the controlled ideal are tolerated as long as they are infrequent (and not attributable to a design defect).

2. Shared media

Paternoster can be used with shared media that supports (at least) two levels of per frame priority.[16]

Priority use

Precedence is given to frames transmitted from the *prior* queue for each port attached to the shared media. Frames in the *current* queue on any port are not transmitted until all the *prior* queues have been drained.[17] The various port's epochs do not have to be aligned[18]: as long as the sum of the reservations for all the ports do not exceed the medium's capacity, each of the *prior* queues will drain in its own port's epoch.

Examples

Figure 11 shows the *prior* queue occupancy over time for three ports[19] attached to the same shared medium. Received frames are not added directly to a *prior* queue. At the beginning of each of a port's epochs its *prior* queue contains the frames (if any) left on the previous epoch's *current* queue. At a maximum this will be the sum of the reservations for that port. For simplicity the figure shows that maximum, neglecting the possibility that frames from that queue could have been transmitted in the previous epoch at a time when all the *prior* queues are empty.

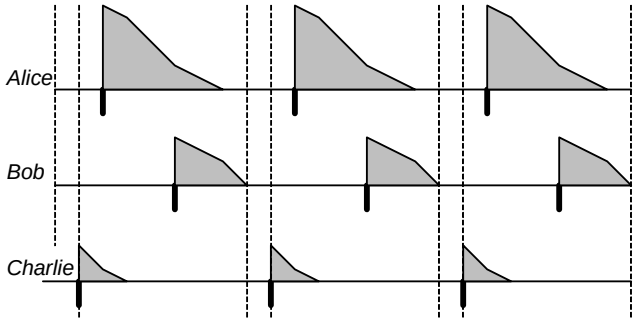

Figure 11—Shared media use (1)

Since the sum of the reservations for all the ports is less than the total available bandwidth, there will be at least one period in each epoch when all the *prior* queues have been drained. Figure 12 and Figure 13 show the *prior* queue occupancies for the same reservations in scenarios where Bob's epochs are offset.

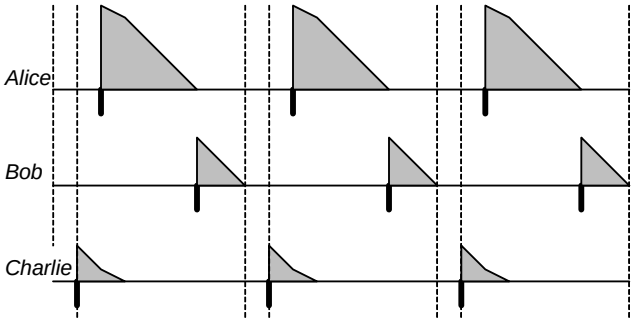

Figure 12—Shared media use (2)

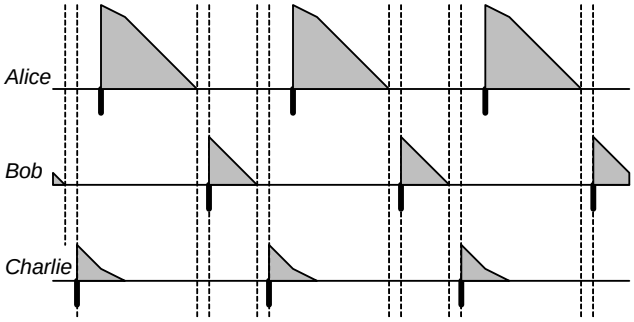

Figure 13—Shared media use (3)

Urgency not 'fairness'

In each of the figures, the queues are shown as draining and equal rates when two (or more) current queues are occupied at the same time. This is not an essential feature of the algorithm, it is sufficient that the available bandwidth be used to drain any non-empty *prior* queue. Above the MAC, paternoster is using priority to communicate urgency[20]—meeting delay bounds depends on draining those queues. Per packet priority is sufficient, and indeed all the frames for the highest priority for a given port might be transmitted before giving the next port a chance to transmit, though delay variance should improve with a more granular approach to frame interleaving.

[16] A priority mechanism has been proposed (though not currently adopted) for use with 802.3 PLCA (Physical Layer Collision Avoidance). As I understand this proposal, two (or more) levels of priority are encoded in the Beacon that is passed around the (logical) ring of stations attached to the shared media. A station that has a highest priority frame to transmit can take immediate advantage of the passing Beacon, a station with a frame of lower priority might encode that priority in the Beacon and transmit if the Beacon comes round again with no increase in its priority field. There are some obvious details/variants to such a scheme, and here is no need to spell them out here, other than to note that the goal is absolute priority - not the physical layer's opinion of fairness. There is no need to revisit the discussions of the early 1980's.

[17] Frames in the *next* and *last* queues are not candidates for transmission until changes in epoch identify them as on the *current* queue. As per the paternoster algorithm they are not transmitted: to do so early would risk frame loss further on the path to their destination. Best effort frames can share the lower priority level with frames from the *current* queue, though they are queued separately. An obvious plan would prefer transmission from the *current* queue (if not empty) though if there really is no value in delivering earlier than necessary to meet the guaranteed delay bound, *best effort* should be preferred over *current*.

[18] Though there are, as in the basic paternoster algorithm

[19] Our familiar friends *Alice*, *Bob*, and *Charlie*, this time in a different configuration.

[20] A more sophisticated scheduling algorithm than paternoster might use finer grades of relative urgency to coordinate access to the shared media, reflecting a frame's queue residence time, but I would expect diminishing returns for increased complexity, with priority signalling still the appropriate mechanism.

3. Aggregate flows

While the per-flow state required by paternoster may meet the needs of networks using per-flow reservation and monitoring, there is always interest in reducing or eliminating per-flow state. This section considers the effect of applying a single bandwidth allocation to multiple individual flows, assuming that each first hop polices its flows separately and that each node has sufficient flow recognition capability to determine which flows are, or are not, in an aggregate.

Forwarding constraints

An aggregate flow's reservation can be large, possibly covering all the frames on a link for a given service class. The bounded delay and lossless characteristics of individual flows are preserved by constraining how they are multiplexed into an aggregate, how the aggregate is forwarded, and how individual flows are demultiplexed from the aggregate.

Individual flow forwarding (recap)

When frames for an individual flow are forwarded, the first frame transmitted from a *prior* queue to the next port on the path to its destination will be[21]:

a) received into the *current* queue, and forwarded by before the beginning of receiver's next epoch; or,

b) received into the *current* queue, and forwarded from the receiver's *prior* queue in its next epoch; or,

c) received into the *next* queue, and forwarded from the *current* queue in the receiver's next epoch; or,

d) received into the *next* queue, and forwarded from the *prior* queue in the receiver's next but one epoch.

Any subsequent frames from that *prior* queue will be treated in the same way, or as specified by a later item in the above list, or:

e) received into, and forwarded from the *current* queue in the receiver's next epoch; or,

f) received into the *current* queue in the receiver's next epoch, and forwarded from its *prior* queue in the following epoch.

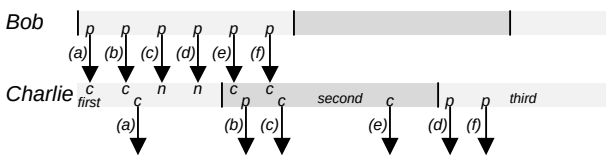

Figure 14—*prior* queue next hop possibilities

Figure 14 illustrates these possibilities for a frame forwarded by *Bob* to *Charlie*, with *Charlie's* epochs identified (*first*, *second*, ...) relative to the first that could receive *Bob's* initial transmission.

Potential aggregate multiplexing issues

If *Charlie* multiplexes the frames from *Bob* with those of another flow (from *Brian*, say) received on a different port, using only parameters associated with their combined reservation($\rho_{aggregate}$, say) some of *Brian's* frames that his individual flow reservation would have forced into *Charlie's next* queue [e.g. cases (c) and (d) above] could take advantage of that larger reservation and be received in the *current* queue [as per case (b)]. Those frames, by lowering the remainder of $\rho_{aggregate}$ associated with that *current* queue could displace one or more of Bob's frames that his individual reservation would have enqueued in *current* [as per case (b)] into *next*, potentially delaying their transmission [as per case (d)]. Frames are not lost as a result of this shift, as there is both sufficient bandwidth to accommodate the aggregate flow and that bandwidth is being used (otherwise the shift would not occur). However the transit delay experienced by *Bob's* frames can have a greater variance. A burst from *Brian* (as in Figure 7) could result in frames from *Bob's prior* queue being enqueued in *Charlie's last* queue, and an additional epoch delay before they are transmitted by *Charlie*.

An individual flow might be multiplexed into a larger aggregate at multiple nodes on the path to its destination, each adding this extra delay.

Aggregate multiplexing

To avoid the delays described above, a node that multiplexes flows received on two or more ports into an aggregate flow transmitted on a third port uses the reservations for each of the flows received on each reception port to select between the outbound port's *current*, *next*, and *last* transmit queues. The delays experienced by the individual flows' frames within an aggregate are those that would occur if their individual reservations had been used.

Fifo forwarding

As previously described, the paternoster algorithm does not constrain transmission order beyond requiring transmission of all frames from a *prior* queue before any from the same port's *current* queue. However that does raise the possibility of preferring

[21] There is no later reception queue than *last*. Frames transmitted can be transmitted from both *prior* and *current*, and those from *current* are guaranteed to be received without loss. Since *current* could fill the *last* queue, frames from *prior* must be received into *current* or *next*. Frames from a single transmit queue cannot overrun a single reception queue, so if frames from *prior* were being received into *next* in the receiver's preceding epoch and that *prior* queue is not yet exhausted its remaining frames will be received into *current* (renamed on the change in the receiver's epoch) in the receiver's present epoch. If they were being received into *current* in the preceding epoch they will be now be received into the new, separate, *current* queue for the present epoch.

(deliberately or accidentally) frames of one individual flow constituent of an aggregate flow over another, progressively shuffling some flows' frames earlier (and those of other flows' later) in the outbound queues of successive ports along the aggregate's path. This shuffling increases the probability of any given port queue (*current*, *next*, or *last*) containing more frames for an individual flow than would be permitted by that flow's individual reservation. This is not a problem while the aggregate is forwarded as a whole, but can be when it is demultiplexed. The effect is reduced, and its analysis simplified, by requiring FIFO transmission of the *prior* and *current* queues when they include an aggregate flow.

Aggregate forwarding and individual flow bursts

If any aggregated individual flow *i* does not consume its ρ_i contribution to the $\rho_{aggregate}$ permitted in a forwarding port's transmission queue, another flow *j* can add more frames than ρ_j would permit. If $\rho_{aggregate}$ is much greater than ρ_j, this accumulation could occur at every hop on the aggregate flow's path. While *j's* frames near the front of the queue could have been delayed by up to 2τ per aggregate hop, the following frames might have experienced minimal delays unchecked by ρ_j. A given queue at aggregate hop *a* could contain an excess of a much as $2a\rho_j$ octets for *j* (see Figure 15[22]). The excess could also be distributed over adjacent queues, though (as in individual flows) cannot be repeated until after a lull. A limit (in a single queue) to this accumulation is reached as $a\rho_j$ reaches $\rho_{aggregate}$.

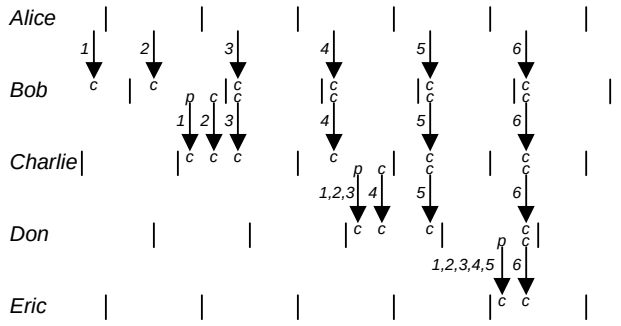

Figure 15—Flow burst within an aggregate

Aggregate demultiplexing

Each frame demultiplexed from an aggregate flow (into an individual flow or another aggregate) is assigned to a queue that reflects its eligibility time for transmission, using additional epochs (each with its queue) to remove any burst accumulation within the aggregate. These queue can require additional buffering, since it can be occupied for longer before being recycled,[23] with an upper bound based on a hypothetical worst case as follows: the demultiplexed flow has been experiencing the maximum possible cumulative delay through the nodes in the aggregate's path, due to the presence of cross-traffic (not in the aggregate), despite the fact that all the other flows in the aggregate are quiet, then the cross flows become silent, allowing all the delayed frames to arrive at the demultiplexing point at the total aggregate rate, while they depart at a rate set by the reservation for the demultiplexed flow. In this scenario, the additional buffering for a demultiplexed flow $_j$ that is small compared to the aggregate, which is itself small compared to the overall link bandwidth is $2a\rho_j$. The worst case additional buffering for an aggregate that is demultiplexed occurs when demultiplexing a flow that has a reservation that is half of the aggregate's total, and is $a\rho_{aggregate}/2$.

Aggregate delays

The maximum end-to-end delay of $(2h - 1)\tau$ for an individual flow is not increased by including that flow in one or more aggregates along its transmission path. The additional epochs required to 'play out' individual bursts within an aggregate do not add to the delay bound, since they are only used when frames have arrived early relative to previously delayed frames.

[22] In the figure, *Eric* does not receive a frame 7 in the same epoch as frames 1 through 6. That is a possibility, but is difficult to show latest possible transmission and earliest possible reception node to node in a figure where transmissions take up space (time).

[23] As opposed to following the usual four epoch sequence, with potential reuse of an emptied *prior* queue's buffer' for the *last* queue in the following epoch.

A. Pseudo-code

The following ‘C’ code fragments illustrate the algorithm and highlight various points about its externally observable behavior—they are not intended to constrain real implementations in any other respect.

See Figure 16. Successive epochs and their *current* transmit queues are identified by the cyclically repeating series Zero, One, Two, … . The present epoch for each port and class of service can differ: epoch array elements identify their present *prior*, *current*, and *last* epochs[24] and currently selected tx (transmit) queue (*prior* or *current*)[25].

The reservations information for each port, class of service and packet stream or flow[26] comprises the number of transmitted octets (including the overhead attributable to each packet) permitted for that flow in an epoch, the epoch (queue_for: *current, next,* … *last*) for which that reservation’s packets are being queued at present, and the remaining octet allocation for the reservation in that epoch.

```
typedef int     Int;    // Types and constants case stropped by convention.
typedef Int     Port_no;
typedef Int     Class;          // Class of service

#define Epochs 4 // epochs and transmit queues for each port and class
#define Reservations // number (arbitrary) of reservations per port and class

typedef Int     Epoch;          // {Zero, One, Two,. Queues-1} repeating
typedef Int     Allocation;

typedef struct
{
   Epoch       prior;
   Epoch       current;
   Epoch       last;
   Epoch       tx;
} Port_class_epoch;

typedef struct
{
   Epoch          queue_for;
   Allocation     remaining;
   Allocation     permitted;
} Reservation;

Epoch                   following[Epochs];
Port_class_epoch        epoch[Ports][Classes];
Queue                   queue[Ports][Classes][Epochs];
Reservation             reservations[Ports][Classes][Reservations];
```

Figure 16—Data types and structures

See Figure 17. When a packet (for an egress port and class of service) is relayed, its transmit packet_allocation is subtracted from that remaining for its reservation’s present queue_for epoch. If the packet will fit it is enqueued, and if the remainder is not zero (indicating the possibility of queuing further packets for that epoch) the number remaining is updated and the procedure returns True (indicating success). If the packet was an exact fit, and the reservation had not yet begun queuing for the following epoch, queue_for is advanced to that epoch and the number remaining reinitialized to the permitted quota before the procedure returns. If the packet didn’t fit and the reservation has not yet advanced to the *next* epoch, the remainder is recalculated for that epoch with its updated allocation. This second attempt might succeed or fail (the total permitted allocation might be less than required for the packet’s size). If the packet is not enqueued the procedure will return False, with queue_for identifying the next_epoch and remaining the excess of the (possibly multiple) queuing attempts in excess of the permitted allocation.

```
Boolean relay(port_no, class, reservation, packet, packet_allocation)
Port_no         port_no;
Class           class;
Reservation *reservation;
Packet          packet;
Allocation      packet_allocation;
{
   Allocation      remainder;
   for(;;)
   {
      remainder   = reservation->remaining - packet_allocation;

      if ((remainder >= 0)
      {
         enqueue_packet(port_no, class, reservation->queue_for);
      }
      if ((remainder > 0) ||
         (reservation->queue_for == epoch[port_no][class].last))
      {
         reservation->remaining = remainder; return (remainder >= 0);
      }
      reservation->queue_for  = following[queue_for];
      reservation->remaining  = permitted;
      if (remainder == 0)
      {
         return (remainder >= 0);
} } }
```

Figure 17—Queuing a relayed packet for transmission

[24]The current value of all three epoch identifiers can be derived from any one: this structure avoids the need for modular arithmetic in the following code.
[25]The queue structures themselves are independent of this description and are not included in Figure 16.
[26]The procedures and criteria for associating any given packet with a particular reservation are independent of the present algorithm.

Note that if the relayed packet cannot be queued for the an epoch, no part of that epoch's allocation is carried forward to the following epoch. Nor is any subsequent smaller packet queued for any epoch once that epoch's permitted allocation has been exceeded.

See Figure 18. When an transmit opportunity for a port and service class arises, this procedure attempts to dequeue a packet from the epoch.tx queue. Initially, i.e. following the start of a fresh epoch, this may be the queue associated with *prior* epoch, as packets can be added to that queue (reservations permitting) right up to the end of the *prior* epoch (when it would have been *current*).

If the dequeue operation returns a packet, or the epoch.tx queue is already that for the current epoch, the procedure returns the packet (or a null pointer if no packet was available). Otherwise epoch.tx is updated to refer to the *current* epoch queue and the dequeing operation reattempted. Note that once the *current* epoch has started packets will no longer be added to the *prior* queue, so once the latter has been drained the transmit focus selection can shift to the *current* epoch's queue. Packets can be added to and removed from this queue throughout the *current* epoch, though packets for some reservations (in excess of their per epoch permitted limit) can be placed on the *next* epoch's queue, thus delaying their transmission (see Figure 17).

See Figure 19, which completes the model with the operations necessary when an epoch gives way to its successor. By this time the *prior* transmit queue should be empty (if the permitted total for all reservations has not erroneously exceeded the transmit capacity) — the queue is purged (emptied) to guard against persistent errors. The *prior*, *current*, and *next* epoch identifiers are updated (if they were Zero, One, and Two, they become One, Two, and Zero respectively). Then each reservation that is not already queuing to the (new) *current* epoch is updated to queue_for that epoch with its remaining allocation initialized to the permitted allocation for an epoch.

```
Packet tx_select(port_no, class)
Port_no     port_no;
Class       class;
{
   Packet      packet;

   for(;;)
   {
      packet = dequeue(port_no, class, epoch[port_no][class].tx);
      if ((packet != Ptr_to_null) ||
         (epoch[port_no][class].tx == epoch[port_no][class].current))
      {
         return(packet);
      }
      epoch[port_no][class].tx = epoch[port][class].current;
}  }  }
```

Figure 18—Transmit selection

```
epoch_tick(port_no, class)
Port_no            port_no;
Class              class;
{
   Epoch temp = epoch[port_no][class].prior;

   purge_queue(port_no, class, epoch[port_no][class].prior);

   epoch[port_no][class].prior       = following([port_no][class].prior);
   epoch[port_no][class].current     = following([port_no][class].current);
   epoch[port_no][class].last        = following([port_no][class].last);

   for(i = 0; i < Reservations; i++)
   {
      if (reservations[port_no][class][i].queue_for !=
          epoch[port_no][class].current)
      {
         reservations[port_no][class][i].queue_for =
            epoch[port_no][class].current;
         reservations[port_no][class][i].remaining   =
            reservations[port_no][class][i].permitted;
}  }  }
```

Figure 19—Epoch updating