Paternoster policing and scheduling

Mick Seaman

This note describes a simple real-time packet bandwidth reservation, policing, queuing, and transmission scheduling algorithm that provides bounded network delays without requiring clock synchronization between adjacent network nodes[1].

---

1. Summary

The paternoster real-time forwarding algorithm provides bounded delays across the network and lossless service for streams that conform to their reservations. The basic algorithm uses four output queues per class of service per port[2], and one counter for each egress port stream reservation. Best effort traffic can make use of any remaining bandwidth (either unreserved or not currently used), with a relaxed upper delay bound (before discard).

The algorithm can also be described (see 4. below) 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[3]; or indeed as an improved credit based shaper.

All the nodes (in any particular network) are configured to use a basic epoch duration[4] τ. Flow reservation i_p occupies a λ_{ip}[5] fraction of the transmission bandwidth ω_p of port p. p's epochs are not synchronized with those of other ports forwarding frames for eventual transmission by p.

At any given time the *prior*, *current*, *next,* and *last* epochs each have an associated transmit queue. Packets are transmitted from the *prior* queue (while any remain on that queue) and then from the *current* queue. Relayed packets associated with any given reservation are added to the *current* queue, until the addition of a packet would exceed that reservation's bandwidth allocation for an epoch, then to the *next* and finally to the *last* queues, discarding any additional packets.

When a new epoch begins, the *current* queue becomes the *prior* queue, the *next* and *last* queues (and their remaining allocation for each reservation) becomes *current*, and *next* respectively. The former *prior* queue should be empty (if it is not, there has been a reservation or transmission selection error, and the queue is purged), is given a fresh set of reservations, and becomes the new *last* queue.

Figure 1 shows the fortunate scenario in which *Alice*[6] transmits consecutive packets (1, 2, 3, …) for a given reservation at regular intervals, one in each of her epochs (delineated by |), with each packet experiencing a constant delay en route to *Bob* who queues each to the current queue (shown by *c* in the figure) in each of his epochs before transmitting it to *Charlie*, again with constant delay.

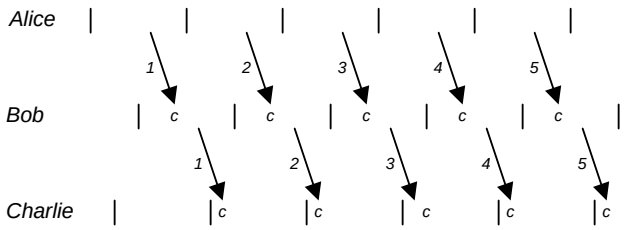

Figure 1—Best case forwarding (constant delays)

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

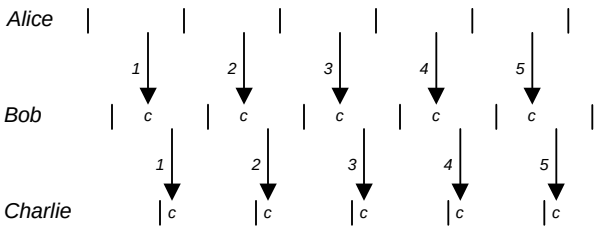

Figure 2—Best case, constant delay, time shifted

---

[1]The pasternoster algorithm originated in a network delay analysis. This note is a result of a request from John Messenger at the January 2017 interim for a protocol description that meets certain specific criteria, though we did not discuss these algorithms. Paternosters are described in https://en.wikipedia.org/wiki/Paternoster. I am not aware of any networks of paternosters, but the construction of a chain with load transfers at intermediate nodes should (for engineering reasons) be easier than the use of a single paternoster of the considerable extent required for many potential deployments.

[2]The use of the term queue here refers to a packet FIFO, though more sophisticated service algorithms are possible.

[3]Although these two scheduling approaches are probably thought of as being very different, there is a mapping between them. I have not had the time to participate in the P802.1Qcr discussion: everyone who has may already be familiar with these ideas—feedback appreciated.

[4]The wikipedia entry for the use of the following symbol, tau, in biology, seems appropriate. https://en.wikipedia.org/wiki/Tau

[5]Including the per frame overhead. A reservation has to be made with some awareness of packet size and per headers (including those added en route).

[6]In the examples of present interest *Alice* always talks (and never listens), while *Bob* may relay the conversation to *Charlie*, who rarely says anything.

The algorithm does not dictate exactly when (in an epoch) frames for a given reservation are transmitted. The class of service transmit 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. If one or more packet can be transmitted, then one of them should be. *Alice* might 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 in a single epoch, adding the first to the *current* queue and the second to the *next* as shown in the Figure 3 scenarios (for a one epoch per frame reservation).

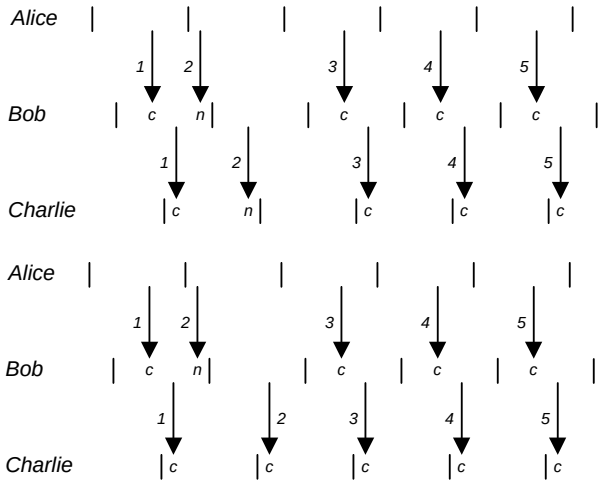

Figure 3—Variable transmission timing

In fact, because all the packets for a given reservation can arrive and be added to the *current* queue at the end of an epoch, up to two epoch's worth of traffic can be transmitted in one of the transmitter's epochs: from the prior epoch's *current* queue (now labelled *prior*) as well as that received during the present epoch (added to and transmitted from the *current* queue). However if that doubled up transmission occurs the transmitter will have transmitted at most one epoch's worth of traffic in the immediately prior epoch (from the *prior* queue for that epoch) and one epoch's worth of traffic in the immediately succeeding epoch (because there will be no traffic left on the present epoch's *current* queue)[7]. So a recipient can receive at most three epoch's worth of traffic in a single epoch, and add that to the *current*, *next*, and *last* queues. See Figure 4.

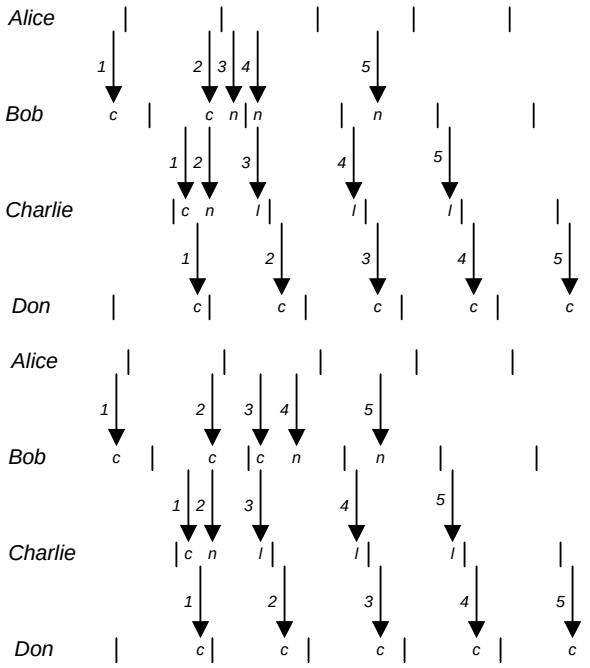

Figure 4—Worst case bunching

The bunching of two epoch's reservations into a single epoch on reception can propagate (e.g. to *Charlie* in Figure 3, first scenario), or might be defeated by transmission of the two (sets of) frames in different epoch's (as *Bob* does in the second). *Bob's* use of the *next* queue might end in the following epoch (first two scenarios) or persist (as in the third) until *Alice* reverts to her previous in-epoch transmission timing.

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 5.

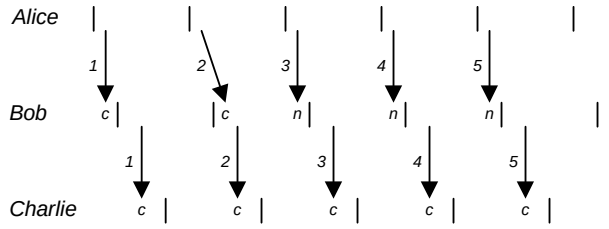

Figure 5—Variable transit delay

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[8], then variations in packet size imply variations in transit

[7]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.

[8]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.

delay. Larger effects result from the use of preemption (which can significantly increase the transit time of the preempted frame). However, if the time to transmit a full *prior* queue (starting at the beginning of an epoch), plus the variation in transit delay, plus any differences between participants' epoch duration does not exceed τ, then we do not require any extra queues. This is sufficient to ensure that four epoch's worth of reservations are not received in a single epoch: in Figure 4 frames 1, 2, and 3 from *Bob* are not delayed in transit so much that *Charlie* receives them in the same epoch as frame 4.

This delay restriction can be met by ensuring that one epoch's worth of reservation leaves adequate time for best effort traffic, while still allowing for the delay variation experienced across a bridge from input port to output port. *Alice*, *Bob*, and *Charlie* are associated with ports that are transmitting frames for a given reservation, and there are sufficient *Alices* etc. to allow all the frames for a reservation to be received at the same time—if that is necessary to make a worst case argument. See Figure 5

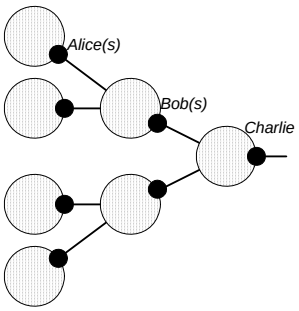

Figure 6—Algorithm participants in network

For algorithm pseudo-code see 6. below.

The basic algorithm's worst case forwarding delay through a single bridge for reservation conforming traffic is 3 τ, and could occur if there is sufficient network fan-in to realize the worst case scenario described above (e.g. frame 3 at *Charlie* in Figure 5). Detailed analysis of modest over provisioning, fan-in, and packet sizes can reduce this bound, but part of the attractiveness of a good simple scheme is not having to do that analysis.

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

2. Using additional epochs

The basic paternoster algorithm can be extended to use additional epochs/queues, adding *next_but_one*, and *next_but_two*, … before *last* to accommodate network links with greater transit delay variation. PBNs can be used (for example) to interconnect local sites without requiring the frames traversing the PBNs to be identified as having a separate class of service or increasing τ for all traffic. Only the bridges that interface directly with the PBNs need use the additional epochs.

3. Best effort traffic

Best effort traffic can be simply transmitted at a strictly lower priority, filling in the transmit opportunities left by reserved traffic in any given epoch. If this is done, the bandwidth remaining after fixed reservations should allow for at least one maximum sized best effort frame per epoch (as well as meeting the frame delay criterion described above), 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 also be used to bound transmission delays without restricting the amount of bandwidth used in an epoch.

4. Comparison with other algorithms

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 the 2 τ. 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 $3\ h\ \tau$, 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.

5. Additional remarks

It may be apparent from the figures above that, once the *last* queue is being used (with the delay that implies) it will continue to be used until the inbound traffic reverts to its previous timing (with the epoch gap in reception that implies). I would recommend that the τ configured for the bridges within the network be slightly shorter (as part of not attempting 100% network utilization) than that used by the attached sources of traffic, so the queues will definitely clear.

6. 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 7. 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 (see 1. above): epoch array elements identify their present *prior*, *current*, and *last* epochs[9] and currently selected tx (transmit) queue (*prior* or *current*)[10].

The reservations information for each port, class of service and packet stream or flow[11] 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 7—Data types and structures

See Figure 8. 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

```
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 8—Queuing a relayed packet for transmission

[9]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.
[10]The queue structures themselves are independent of this description and are not included in Figure 7.
[11]The procedures and criteria for associating any given packet with a particular reservation are independent of the present algorithm.

next_epoch and remaining the excess of the (possibly multiple) queuing attempts in excess of the permitted allocation.

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 9. 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 8).

```
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 9—Transmit selection

See Figure 10, 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.

```
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 10—Epoch updating