Paternoster[1] 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[2].

1. Summary

The paternoster real-time forwarding algorithm provides bounded delays across the network and lossless service for streams that conform to their reservations. It uses three output queues per class of service per port[3], 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) if required.

The algorithm can also be described as an improvement on the peristaltic shaper (CQF), with the node to node synchronization requirement removed (see 5. below); or as deadline oriented, with not before/not after attributes for forwarded packets[4].

All the nodes (in any particular network) are configured to use a basic epoch duration[5] τ. Flow reservation i_p occupies a λ_{ip} fraction of the transmission bandwidth ω_p of port p, i.e. p is allowed to send up to $\lambda_{ip}\omega_p\tau$ octets/bytes[6] for frames associated with i_p in each epoch. That epoch is not synchronized with the epochs of the ports that are forwarding frames from other nodes for eventual forwarding through p, so p can receive (and have to buffer) up to $2\lambda_{ip}\omega_p\tau$ bytes: the worst case assumption being that the upstream node(s) transmit(s) all of one epoch's allocation at the end of that epoch, all of the next epoch's allocation at the beginning of that epoch, and that p receives[7] both bursts in a single epoch[8]. Equivalently p can receive traffic towards the end of an epoch that fits within that epoch's allocation, but whose transmission will naturally extend into the next epoch. Fortunately these effects are not additive.

At any given time the *prior*, *current*, and *next* epochs each have an associated transmit queue. Packets are transmitted from the *prior* queue, while any remain, 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, and then to the *next* queue, again until a further addition would exceed that epoch's allocation. Any additional packets for the reservation are also discarded, until it is time to begin a new epoch.

When a new epoch begins, 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). The *current* queue becomes the *prior* queue, the *next* queue (and any remaining allocation for each reservation) becomes *current*, and the (now empty) *prior* queue becomes the *next* queue.

For algorithm pseudo-code see 6. below.

The worst case per forwarding node delay for reservation conforming traffic is $2\ \tau$, and could occur if there is sufficient network fan-in to realize the worst case scenario described above. The traffic immediately following that experiencing the $2\ \tau$ delay will suffer, at worst, a single τ delay as the worst case condition is not immediately repeatable. Detailed analysis of modest over provisioning, fan-in, and packet sizes can reduce these bounds, but part of the

[1] Picking a name for the purposes of this note.

[2] 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.

[3] The use of the term queue here refers to a packet FIFO, though more sophisticated service algorithms are possible (see later). A more complex control structure requires only two FIFO queues, but the three queue version is easier to describe.

[4] Although these two scheduling approaches are probably thought of as being very different, there is a (contorted) mapping between the two. 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.

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

[6] Including the overhead associated with each frame, so a reservation has to be made with some awareness of packet size and per frame overhead (including an allowance for headers added en route).

[7] Assuming an output buffered model in this description.

[8] If there is a single egress port, of equal bandwidth, for all flows at p's node the total buffering requirement is clearly reduced,. A sophisticated analysis would take into account the fan in, the total reserved bandwidth faction, and any other algorithm interleaving frames on output. It is probably not worth doing that analysis. One approach is to allocate $\lambda_{ip}\omega_p\tau$ bytes per queue, so the total is three times (rather than twice) that—which may simplify error handling.

attractiveness of a good simple scheme is not having to do that analysis.

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 packets eligible for transmission in an epoch. If one or more packet can be transmitted, then one of them should be. If other algorithms share the bandwidth this condition may be met by providing the usable bandwidth at the end of the epoch.

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 have to be 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. Multi-epoch extensions

The algorithm could use additional epochs, e.g. *next*, *next_but_one*, and *next_but_two*, to distribute forwarded frames for each reservation more evenly over time. The difficulty is in the granularity of reservations, which must be large enough to accommodate the near simultaneous forwarding, from all potential fan-in ports, of frames for that reservation.

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, so if the transmission of such a frame extends from the end of one epoch into the start of another the reserved bandwidth commitment can still be met 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.

4. Pre-emption and paternoster

Pre-emption can be used, though each reservation has to be increased (by the difference between the smallest eligible and the maximum tolerated pre-empting frame sizes): to avoid bumping a further frame, received while pre-emption is in progress, from *current* to *next*.

5. 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 the same (when measured in epoch durations) as that of the paternoster algorithm, 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 d 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 $2\ h\ \tau$ (ignoring 'time on the wire' in both cases), 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—making the upper delay bounds the same for both algorithms. Against this has to be set the possible difficulty of making small reservations when using short epochs.

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 1. 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 *next* 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, either *current* or *next*) for which that reservation’s packets are being queued at present, and the remaining octet allocation for the reservation in that epoch.

See Figure 2. 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 *next* 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.

```
typedef int     Int;    // Types and constants case stropped by convention.
typedef Int     Port_no;
typedef Int     Class;          // Class of service
typedef Int     Epoch;          // {Zero, One, Two, ..} repeating
typedef Int     Allocation;

#define Queues 3 // transmit queues for each port and class
#define Reservations // number (arbitrary) of reservations per port and class

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

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

Port_class_epoch    epoch[Ports][Classes];
Queue               queue[Ports][Classes][3];
Reservation         reservations[Ports][Classes][Reservations];
```

Figure 1—Data types and structures

```
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].next))
      {
         reservation->remaining = remainder; return (remainder >= 0);
      }
      reservation->queue_for  = epoch[port_no][class].next;
      reservation->remaining  = permitted;
      if (remainder == 0)
      {
         return (remainder >= 0);
} } }
```

Figure 2—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 mod 3 arithmetic in the following code.
[10]The queue structures themselves are independent of this description and are not included in Figure 1.
[11]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 *current* epoch, no part of that epoch's allocation is carried forward to the *next* epoch. Nor is any subsequent smaller packet queued for the any epoch once that epoch's permitted allocation has been exceeded.

See Figure 3. 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 2).

```
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 3—Transmit selection

See Figure 4, 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);

   epoch[port_no][class].prior               = epoch[port_no][class].current;
   epoch[port_no][class].current_epoch = epoch[port_no][class].next;
   epoch[port_no][class].next                = temp;

   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 4—Transmit selection