Queues in LINCX

Maxim Kharchenko, Cloudozer LLP

03/05/2014

1 Overview

This document describes the details of the implementation of queues in LINCX and the rate limiting algorithm they use.

2 Implementation notes

Each queue is attached to a port – a queue port. Many queues can be attached to the same port. A queue buffers packets and spews them to the queue port if the constraints with respect to the data rate are met. In LINCX queues are represented as Erlang processes.

Queues are a part of the "fast path" of the switch and thus they should maintain minumum memory footprint and suppress garbage collection. A queue is yet another destination of a packets. Queues are created and controlled by the blaze process similarly to ports. The blaze process should maintain the queue map – the mapping between `queue_id` and `Pid`.

The flow table generator understands Set-Queue action already. If the action set returned after matching indicates a `queue_id`, then the corresponding `Pid` should be looked up in the queue map and the packet should be sent to that `Pid`.

A queue `Pid` changes as it restarts to reclaim memory. The protocol below ensures that there are no race conditions when packets are sent to a restarting queue.

The blaze process must trap exits and check for failing queues before processing data packets. Both ports and the queue map and transferred to the next reincarnation of the blaze process upon restart.

3 Restarting a queue

Each queue restarts after processing a fixed number (16384) of packets. The queue passess its state to its successor process including the queue port. The following message exchange happens next:

```
queue --> blaze:    {queue_restarting,OldPid,NewPid}
blaze --> queue:    queue_restarted
```

After receiving `queue_restarted` message the queue drains its mailbox, resends the messages to the successor queue, and exits. The blaze should discern and ignore exits signals coming from the restarting queues. The reason for such signals is `normal`.

4 Rate limiting algorithm

4.1 Variables

The queue keeps the data rate between R_{min} and R_{max}. Thus it transforms bursts of traffic into slower flows capped by R_{max}. It also may put packets closer together effectively creating a burst of traffic with the data rate higher than R_{min}.

The queue has a fixed time window (w) of 250ms. The queue always maintains two variable values: the current rate (R) and the timestamp (T) when the current rate was last updated. R is an estimate of the data rate during the $[T - w, T]$ time interval.

The queue has a packet buffer of a fixed length. The buffer length is configurable. The queue drops packets if the packet buffer overflows. The total size of buffered packets is B.

All calculations should use floating-point arithmetics to avoid bignums.

4.2 Procedure

Suppose a new packet arrives at the moment T_1. If the packet buffer is full then the packet is dropped. Otherwise, the current rate is recalculated as follows:

$$R = \begin{cases} 0, & \text{if } T_1 > T + w \\ R - R(T_1 - T)/w, & \text{otherwise} \end{cases}$$

The value of T_1 takes place of T. The queue then adds the received packed to the end of packet buffer. The total size of buffer packets increases. Note that a large packet larger than wR_{max} will clog the queue. This condition is not checked for each packet, rather setting R_{max} to a value lower than MTU/w is not allowed[1].

The queue now considers the packet of size P in the beginning of the packet buffer. If this packet is sent now, then the data rate will go up to $R_1 = R + P/w$. This rate must not exceed the maximum ($R_1 \leq R_{max}$). If all buffered packets are sent now the rate will shoot up to $R_2 = R + B/w$. This rate must be higher than the minumum ($R_2 \geq R_{min}$).

If both conditions hold than the packet is removed from the buffer and sent to the queue port. The current rate is updated to R_1 and the process continues with the next packet in the buffer.

When the packet buffer is empty or the packet cannot be sent due to rate limitations, the queue may set up a timeout. The timeout is set only if $R_1 > R_{max}$. The timestamp T_t for the timeout depends on the size of the next packet in the packet buffer – P_{next} – as follows:

$$T_t = w - wR_{max}/R + P_{next}/R + T$$

Upon arrival of the packet, a pending timeout, if any should be cancelled. The same procedure is repeated when either a packet arrives or a timeout expires.

[1] In case of Ethernet packets (MTU = 1500 bytes) this requires that R_{max} must be higher than 48kbps.

4.3 Limitations

1. The time window is fixed.
2. The packet buffer has a fixed length.
3. The arrival and send timestamps not kept.
4. If R_{min} is high, packets may wait in the buffer for a long time.

5 Queue configuration

The following specifications are copied from docs/queues.md. Note that parameter `rate` is not used by the proposed queue implementation.

5.1 Port configuration spec

```
[{PortId :: integer(),
  [{rate, {integer(), bps | kbps | kibps | mbps | mibps | gbps | gibps}} |
   {queues, [{QueueId :: integer(), [{min_rate, integer()} |
                                     {max_rate, integer()}]}]}]}]
```

5.2 Example setup

```
{queues,
 [
  {1, [
       {rate, {100, mbps}},
       {queues, []}
      ]},
  {2, [
       {rate, {1, gbps}},
       {queues, [{1, [{min_rate, 250},
                      {max_rate, 500}]}]}
      ]}
 ]}
```