Traffic policing in eBPF: applying token bucket algorithm

LPC 2018

Julia Kartseva, hex@fb.com
Traffic policing in eBPF: applying token bucket algorithm

<table>
<thead>
<tr>
<th>Shaping</th>
<th>Policing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffers exceeding packets</td>
<td>No buffering, instant action: drop or remark</td>
</tr>
<tr>
<td>Latency increase due to buffering</td>
<td>No latency increase, but drops may cause retransmits (e.g. in TCP)</td>
</tr>
<tr>
<td>Smooths traffic burstiness, output rate doesn’t deviate much</td>
<td>Bursts are propagated</td>
</tr>
<tr>
<td>Drops packets anyway when buffer capacity is reached. Buffer increasing causes higher latency</td>
<td></td>
</tr>
<tr>
<td>Linux Traffic Control: queuing disciplines, e.g. tc htb</td>
<td>Switch side policers, eBPF-based traffic policing</td>
</tr>
</tbody>
</table>
Traffic policing in eBPF: applying token bucket algorithm
RFC 2698, naive implementation

Two rate three color marker

```
_u64 delta_t = packet_ts - 2 * bucket->timestamp;
bucket->tokens += delta_t * rate_bps / NS_IN_SEC;
bucket->tokens = MIN(bucket->tokens, burst_size);
bucket->timestamp = packet_ts;
__u64 tokens_spent = 8 * skb->len;
__sync_fetch_and_add(bucket->tokens, (-1) * tokens_spent);
return TC_ACT_UNSPEC;
```

Does this code produce the desired rate?
Traffic policing in eBPF: applying token bucket algorithm
Advancing naive implementation

Problem:
Updates are in the kernel space. Data race with multi CPU. Getting and adding tokens into a bucket must be executed as an atomic action.

Solution or not?
• Critical section in eBPF program
• Per CPU eBPF data structures
• Update tokens from the user space
  What if burst duration is in microseconds?
• Data structures shared between CPUs: lru_hash, array
Traffic policing in eBPF: applying token bucket algorithm
Hackish working implementation: kernel space only, eBPF array

Key idea: refill tokens in a future bucket, take tokens from the current bucket.

```
__u64 packet_ts = bpf_ktime_get_ns();
__u64 burst_dur = burst_size * NS_IN_SEC / rate_bps;
__u32 refill_tbuck_idx = (packet_ts + (K >> 1) * burst_dur) / burst_dur % K;
__s64* refill_tokens = bpf_map_lookup_elem(tbuck_arr, &refill_tbuck_idx);
if (refill_tokens) *refill_tokens = burst_size;
__u32 curr_tbuck_idx = packet_ts / burst_dur % K;
/* Subtract tokens from curr_tbuck_idx bucket */
```
Traffic policing in eBPF: applying token bucket algorithm

Policers chaining

- The output rate must not depend on the order of policers
- If a packet is discarded, recredit the preceding policers
- Policers may not belong to the same logical hierarchy. No common root is required, unlike in qdisc HTB
Traffic policing in eBPF: applying token bucket algorithm

Limitations

• Heavy hammer
  • TCP congestion control + token bucket + DROP = capped max rate but underutilized average
  • No buffering: drops are inevitable

• Very thin per TCP flow fairness guarantees
  • No handy TCP session information in tc chain
  • N sub buckets and skb->hash % N

• Token bucket + DSCP remark
  • Only for multi-queue network devices
  • Packets may be received in disorder