Make DPDK's software traffic manager a deployable solution for vBNG

CSABA KESZEI, ERICSSON

DPDK Summit - San Jose – 2017
Agenda

- The TM problem in access and aggregation networks
- Limitations of DPDK software TM in the light of real deployments
- Other performance and usability tunings
Why do we need Traffic Management?

- Physical access network topology might be radically different
- Intermediate nodes typically lack per subscriber information
- Shape traffic in (v)BNG not to cause any congestion in the access network
Model Access Network transport capabilities in TM tree

Layer 2 connectivity model

Shape traffic in (v)BNG not to cause any congestion in the access network.

Traffic management model (mirror)

- Token bucket
Number of children should be dynamic
- Topology change without traffic disturbance on the rest of the tree is a requirement

Number of levels should also be dynamic
- SVLAN+CVLAN is not supported by DPDK at the moment
- Tunneling cases (like L2TP) could require more levels
Store/traverse subport hierarchy as a linked list

struct rte_sched_pipe {
    [...]  
    uint16_t pipe_subport_id;
};

struct rte_sched_subport {
    [...]  
    uint16_t subport_parent;
};

- Refill subport credits in connection with pipe credit update
- Deduct/verify chain of subport credits upon pipe dequeue
- Fits into our processing budget in case of
  - ‘moderate’ number of subports
  - 3 levels
- No contradictions with new rte_tm_node_add() API
On-demand queue allocations

- Static allocation of queues wastes memory
  - $16 \times 8 \times 256 = 32$K/pipe for 256 long queues
  - 2GB for 64K subscriber slots

- Real topology is more diverse and dynamic, preallocating worst case is not feasible

- Low hanging fruit: allocate queues dynamically
  - Fits into prefetch pipeline
  - Allows for per pipe queue sizes

```c
struct rte_sched_pipe {
    [...]  
    struct rte_mbuf **qbase;
}
```

Configuration example:
```
port ethernet 1/1
no shutdown
encapsulation dot1q
dot1q pvc 3026 encapsulation 1q:tunnel
dot1q pvc on-demand 3026:1 through 4000
gos rate max 100000
idle-down 60
startup-timer 600
service clips dual-stack source-mac
service clips dhcp max 100 context CLIPS_12
service clips dhcpv6 max 100 context CLIPS_12
```
On-demand queue allocations

Layer 2 connectivity model

Pipe (L2, subscriber) (L1)

Traffic class (L0)

Queue (L0)

"SubSubPort" (L3, CVLAN)

SubPort (L3, SVLAN)

Port (L4)

(v)BNG
Use case: re-distribute remaining bandwidth in a subtree to users without configured TM

- Not feasible with static configuration
- Algorithmic change is needed at (sub)port level

Use RFC2697 color-aware srTCM

- TM enabled use conform (green) bucket
- Rest use excess (yellow) bucket
- Red means skip to next pipe

```c
struct rte_sched_subport {
    ...
    uint32_t tb_credits[2];
}
// We do not use subport level TCs
```
- Fixed pipe traversal order
- First-come first served on subport level
- Nothing guarantees
  - Fairness
  - Configurable resource share
Idea: dynamically mark green the fair share

- Inspired by ‘TC3 over-subscription’ but more generic
- Use RFC2698 trTCM on pipe level (PIR = tb_rate)
- Scale all CIRs in the subtree to match configured subport rate
- Configured CIRs become weights
- Users without configured TM get PIR = port_rate, CIR = 0

Control loop

- Pipes visited in a fixed order, to make it fair, make changes once per full round
- Bottleneck: subport where we are out of conform credits
  - Theoretically one per path
- Adjust subport associated scale
  - Overshoot is the bigger problem
  - Unused bandwidth is re-distributed in an unfair way
Speed up credit updates

- *idiv* instruction is also expensive
  - FPU operation is removed via commit:
    - ‘sched: eliminate floating point in calculating byte clock’
  - Few integer divisions are still visible hot-spots

- After simplifications: shift + multiply
  - Granularity is impacted
  - Actual *rate* is the fraction of port rate

```c
grinder_credits_update()
{
    [...]
    uint64_t n_periods;
    /* Subport TB */
    n_periods =
        (port->time - subport->tb_time) /
        subport->tb_period;
    [...]
    /* Pipe TB */
    n_periods =
        (port->time - pipe->tb_time) /
        params->tb_period;

    uint64_t period = (time - tb_time) >>
        tb_period_bits;
    tb_time += period << tb_period_bits;
    tokens = tb_credits_per_period* period;

    tb_period_bits = log2(512.0 / rate);
    tb_credits_per_period = rate *
        (1 << tb_period_bits);
```
- *tc_period* is not intuitive
  - Example for 40ms:
    - Minimal rate is 300kbps to pass a 1500 bytes packet
    - At least 5M buffer per queue (7B125 64 bytes packets) is needed to avoid buffer under-run for 1G rate, unrealistic
  - No intuitive burst size

- Store TC rates, CIR as a fraction of TB rate
  - Cost is granularity, simplifications possible by handling CIR as % of TB rate
  - Fits into the processing chain of division-less credit updates
  - Opens the possibility of real TC level burst size (+CBS)

- Saves few bytes in the structures
  - Especially when profiles need to be embedded