Improving TC Filters insertion rate

Guy Shattah, Rony Efraim

NetDev 2.2 (2017) – The Technical Conference on Linux Networking

Mellanox® TECHNOLOGIES

Connect. Accelerate. Outperform.®

Contents

- Why are we here?
- Work done so far and what are the next steps
- TC Handle lookup flow
- The problem and progress so far
- Flow of a TC filter request
- The RTNL lock
- Suggestion 1: Breaking the Lock
- Suggestion 2: Multi-Threaded Batch under the Lock
- Executing accumulated work
- Comparison

Why are we here?

- SDN becomes more and more relevant.
- OVS is the most popular SDN switch.
- Growing number of concurrent connections leads to increased HW offload demand.
- Connections are offloaded by inserting filters rules.
- OVS datapath can be implemented using the Linux TC subsystem.

Brief: Work done so far and what are the next steps

- Until recently, existing TC code allowed for a poor rules/sec update rate.
- A work done recently has significantly improved insertion rate (~50,000 rules/sec)
- Despite recent improvements - users yearn for a rate of 1M/sec or better.
- Achieving a major improvement will demand a profound change.

TC Handle lookup flow

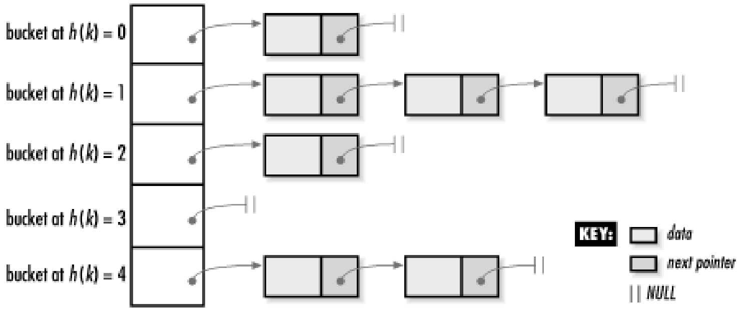
- Input is a TC filter request

1. TC flow:

1. Search for device in a linear list.
2. Lookup specified qdisc.
3. Find a class attached to the qdisc.
4. Find the classifier with the priority.

2. Inside the classifier:

6. Lookup rule handle – classifier (*get)() method searches in a linear list.
7. Set Action - classifier (*change)() method reads additional parameters from "struct * nlattr“, then sets a new matching rule and action to act upon matching. action resides in hash table with buckets of linked-lists



The problem and the solution

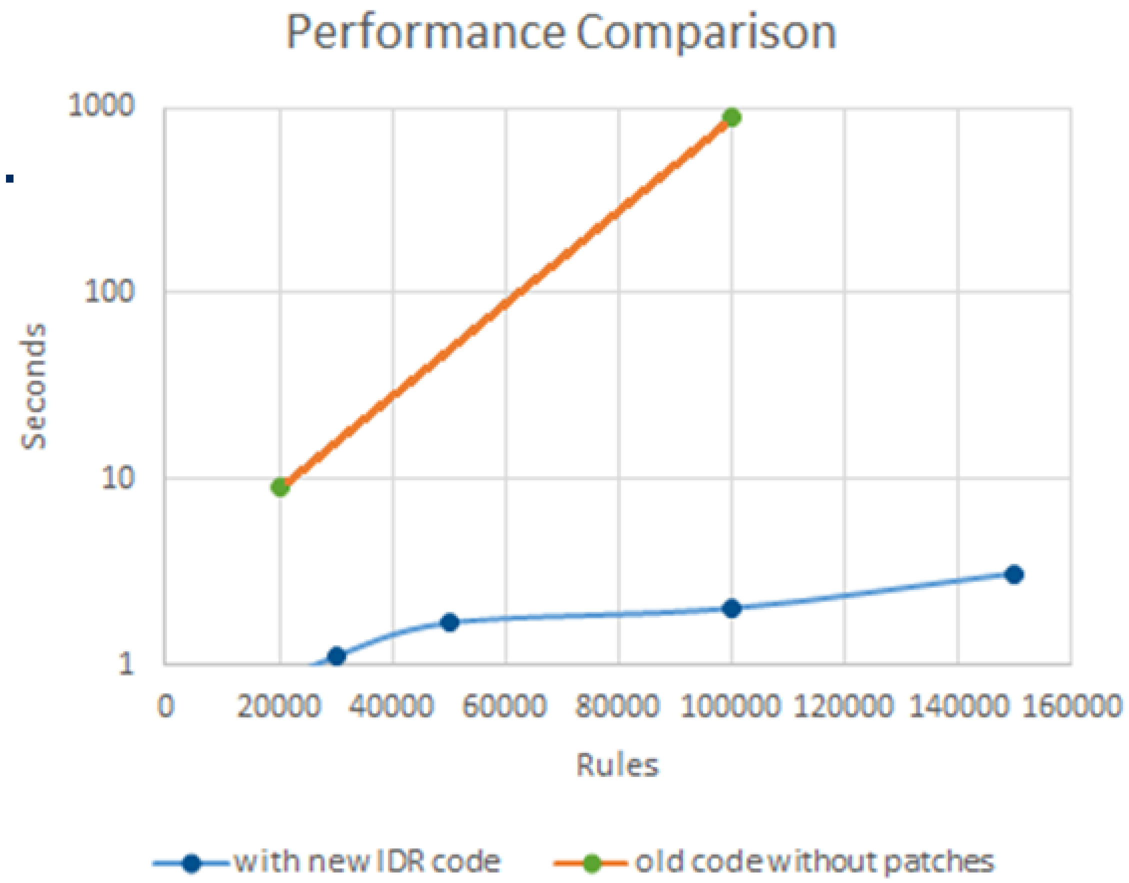
Inside the classifier:

6. Lookup rule handle – **classifier (*get)()** method searches in a linear list

7. Set Action: The **classifier (*change)()** method searches in **hash table with buckets of linked-lists**.

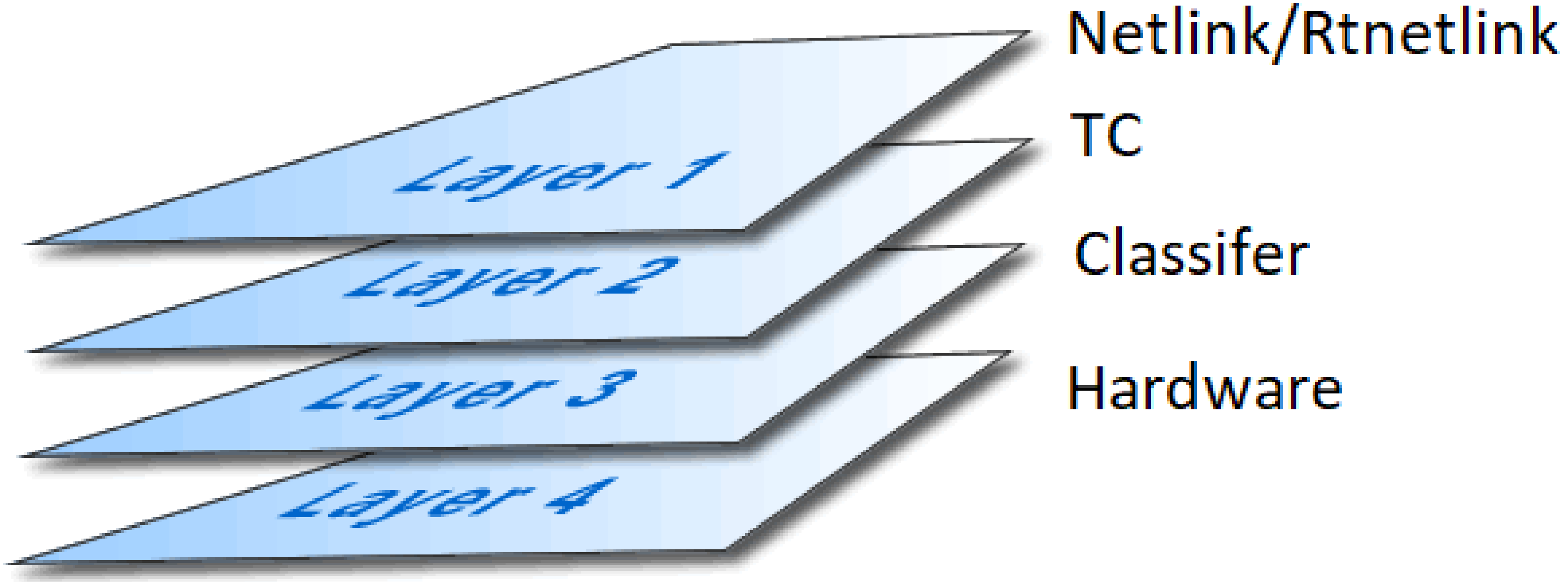
- **(*get)()** method's **linked-list** was replaced by IDR.
- **(*change)() -** method's which was using action **hash table with buckets of linked-lists** was replaced by IDR.

- **Current stable insertion rate: ~50,000 rules/sec (Tested on E3120 Xeon)**



Flow of a TC filter request

1. Netlink layer
2. RTnetLink Layer
   - Accept message
   - Lock RTNL
   - Send to TC
3. TC layer
4. Classifier
5. HW (optional)



The RTNL lock

- The RTNL Lock is a mutex located inside rtnetlink.
- Used to make sure no two threads may enter the rtnetlink subsystem at the same time.

Florian Westphal - Yesterday: "*The widespread use of the RTNL lock in all major network configuration paths is a growing pain point, f.e. a task adding an IP address prevents another from seemingly unrelated tasks such as dumping TCclassifiers. Furthermore, some code paths can hold the rtnl mutex for very long times (in the order of several hundreds of milliseconds in some cases). Rtnetlink is a netlink subsystem used to inspect or change networking related configuration.*"

The RTNL lock (cont.)

RTNL lock effect on TC:

1. User process sends multiple TC filter requests in parallel (each one in separate thread) to the TC layer (via rtnetlink layer)

2. RTNL lock forces one message at a time!

3. tc_ctl_tfilter() method can’t run in parallel ☹

Suggestion 1: Breaking the Lock

- Breaking big lock into smaller locks.

- Difficult task: many kernel methods and drivers rely on the lock.

- In order to remove it, one (or many) would have to analyze very carefully all the code paths called after the lock. Find critical sections and implement smaller granularity locks.

- Florian W. Recently started working on this issue.
  - There is a long way to go before the work is completed.
  - Once the work is complete - vendors still have to make necessary adjustments to remove RTNL dependencies.

Suggestion 2: Multi-Threaded Batch under the Lock

- Don’t we already have netlink-batch support?
  - Existing netlink-batch is provided for convenience
  - Does not promote parallel execution.
- Suggestion 2.A: Multiple netlink messages.
- Suggestion 2.B: Compound netlink Message

- Issues:
  - Parallel processing implies all actions mustn’t have dependencies one on each other.
  - Parallel processing forces kernel to run a multi-threaded code even when the user application is single-threaded (and user possibly does not want to utilize additional CPUs).

Suggestion 2.A: Multiple netlink messages

- Extending netlink interface by introducing batch operations.
- Adding two new netlink flags: NLM_F_BEGIN and NLM_F_END.
  - NLM_F_BEGIN : start accumulating messages.
  - NLM_F_END : initiate parallel execution of accumulated messages.

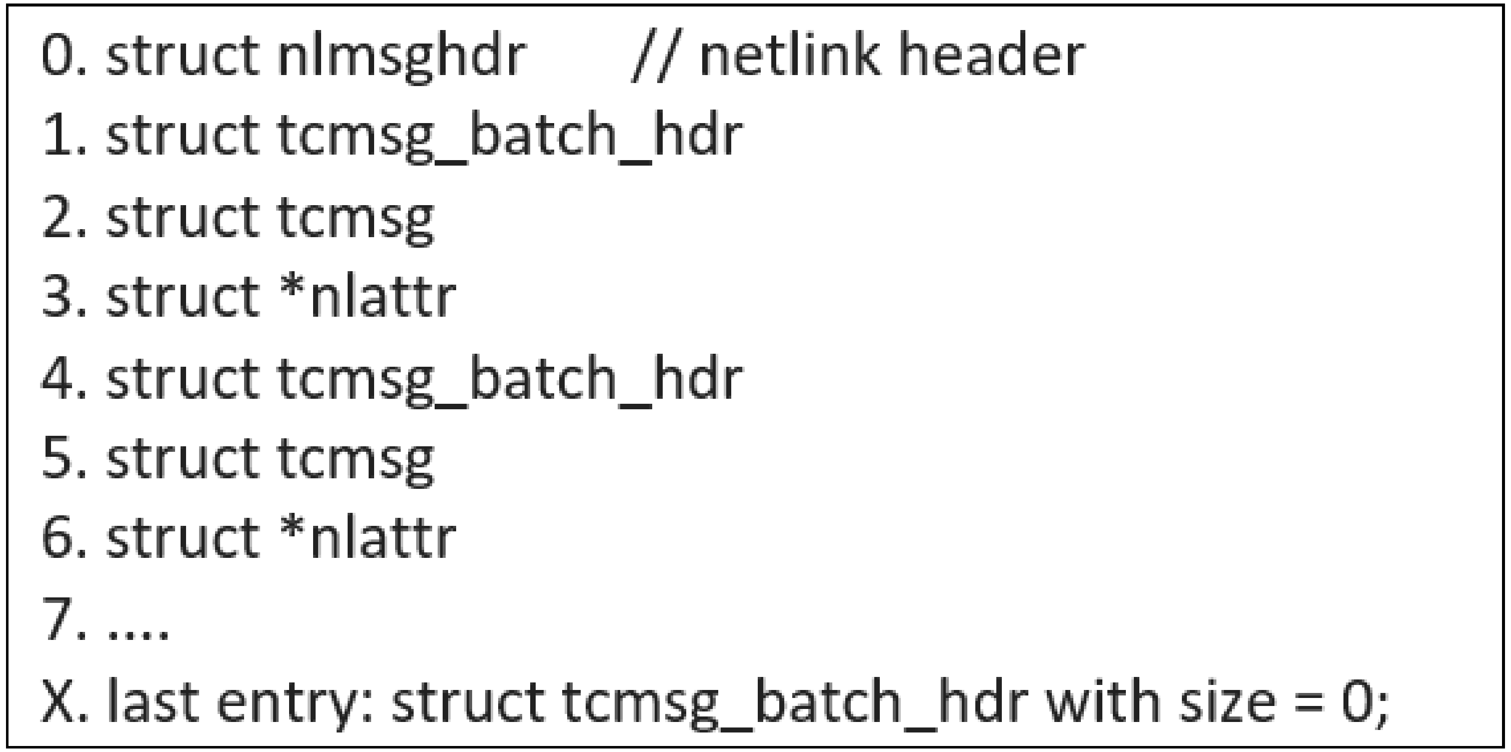
- Technical details :
  - Accumulated messages list has to be maintained per user, with pre-defined quota (to avoid overflow) and with some aging mechanism.
  - Suggestion differs from the existing solution by the use of ‘**begin**’ and ‘**end**’ flags to explicitly specify that all the actions included are to be executed in parallel, not one after another.

Suggestion 2.B: Compound netlink Message

- Introducing a compound TC message, RTM_BATCHTFILTER.
  - Message encapsulates multiple TC filter requests
  - Facilitating work by sending all messages to be executed in TC layer in parallel at once.

- Technical details :

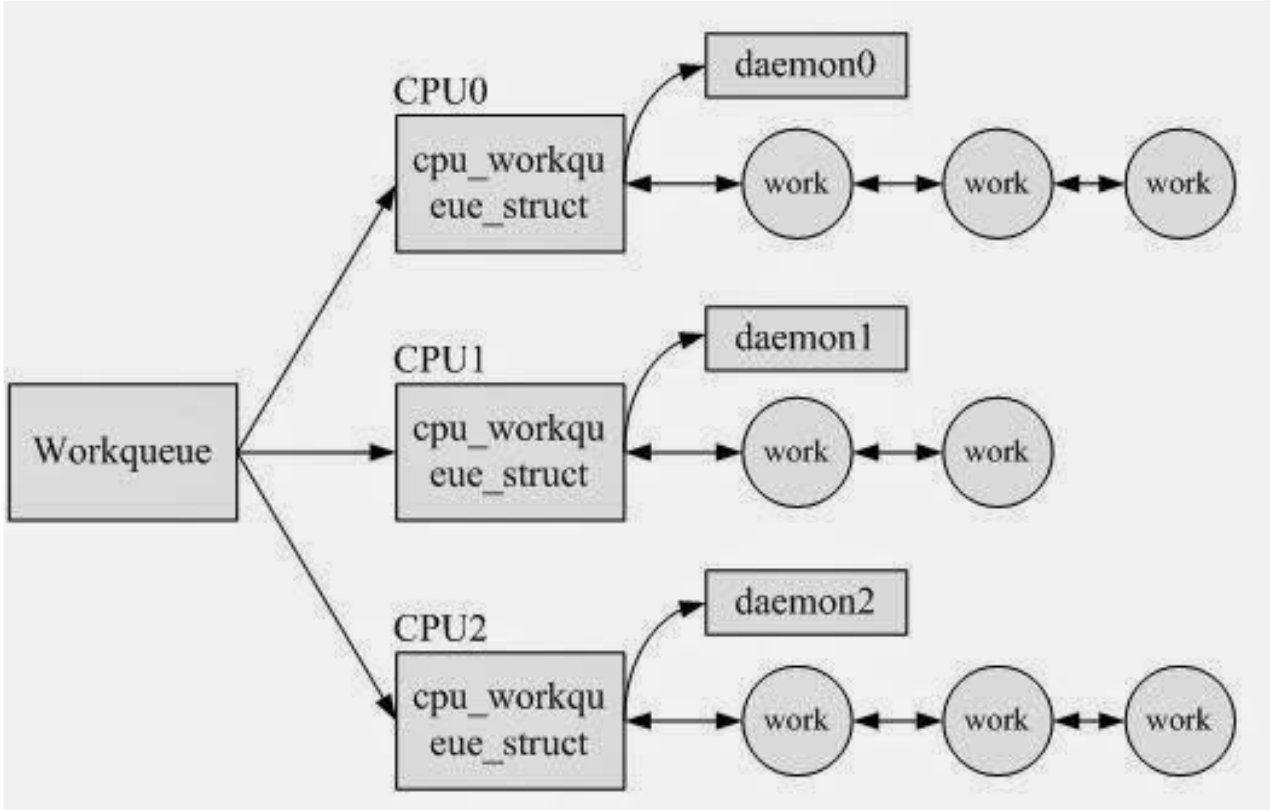
```
struct tcmsg_batch_hdr {
__u32       tcmsg_len;
__u16       tcmsg_type;
 __u16       tcmsg_flags;
 }
```



```
0. struct nlmsghdr      // netlink header
1. struct tcmsg_batch_hdr
2. struct tcmsg
3. struct *nlattr
4. struct tcmsg_batch_hdr
5. struct tcmsg
6. struct *nlattr
7. ....
X. last entry: struct tcmsg_batch_hdr with size = 0;
```

Executing accumulated work

- Accumulated work is executed in a workqueue
  - In suggestion 2.A (Multiple netlink messages):
    - a result is returned per netlink message.
  - In suggestion 2.B (Compound netlink Message):
    - On success: netlink success message.
    - On failures: netlink message contains list of pairs (msg index, error value).



Comparison

2.B Compound TC message	2.A Multiple netlink Messages
✓ Process first message immediately	× Requires NLM_F_END to start processing
✓ No slow-down memcpy()	× memcpy() each message
✓ No internal bookkeeping	× Keep a list of messages per process/user
✓ No internal list size limitation	× Each list mustn't exceed predefined size
✓ Always a single system-call	× Possibly requires more than one system-call
✓ RTNL lock is always taken once	× RTNL lock might be taken more than once
✓ Delivers better performance	✓ More generic

Thank You

Mellanox TECHNOLOGIES

Connect. Accelerate. Outperform.