Improving TC Filters Insertion Rate

Guy Shattah, Rony Efraim
Mellanox, Ra'anana, Israel
[sguy | ronye] at mellanox.com

Abstract

Recent industry movement towards the use of Virtual Machines, OVS, and hardware offloading has led to a new requirements: the necessity for rapid update of TC filters. Until recently the existing TC code allowed for a poor rate[1] of merely 100 rules-updates/sec per priority (in average). Recent progress has improved the insertion rate to 50K/sec. However, this rate still does not satisfy the users who yearn for a rate of 1M/sec or better. In this paper we will discuss the existing situation, work done so far and ideas on adding batching operation to TC and more future architecture enhancements towards achieving this goal.

Keywords

Virtualization, switchdev, TC, OpenVSwitch, offload, flows, tunnels

Introduction

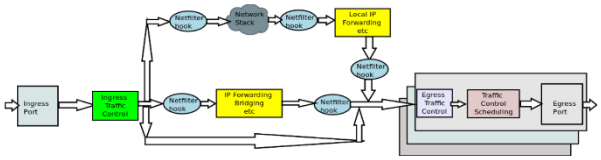
Introduction to TC

Internet traffic engineering deals with the issue of performance evaluation and performance optimization of operational IP networks. Traffic Engineering encompasses the application of technology and scientific principles to the measurement, characterization, modeling, and control of Internet traffic [1].

Enhancing the performance of an operational network, at both the traffic and resource levels, are major objectives of Internet traffic engineering. This is accomplished by addressing traffic oriented performance requirements, while utilizing network resources economically and reliably. Traffic oriented performance measures include delay, delay variation, packet loss, and throughput.

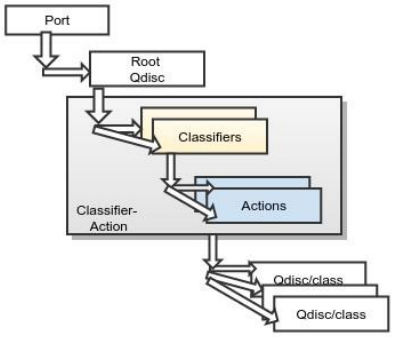
The optimization aspects of traffic engineering can be achieved through capacity management and traffic management. Capacity management includes capacity planning, routing control, and resource management. Network resources of particular interest include link bandwidth, buffer space, and computational resources. Likewise, traffic management includes (1) nodal traffic control functions such as traffic conditioning, queue management, scheduling, and (2) other functions that regulate traffic flow through the network or that arbitrate access to network resources between different packets or between different traffic streams. [2]

TC is Linux Traffic Control mechanism that includes the sets of queuing systems and mechanisms by which packets are received and transmitted on a router. This includes deciding which packets to accept at what rate on the input of an interface and determining which packets to transmit in what order at what rate on the output of an interface. [3]



When the kernel needs to make a routing decision, it finds out which table needs to be consulted.

Overview of TC



Queueing Disciplines (qdiscs) are scheduling objects which may be *classful* or *classless*. When classful, the qdisc has multiple classes which are selected by the classifier filters. A given classful qdiscs can contain other qdiscs, a hierarchy can be setup to allow differentiated treatment of packet groupings as defined by the policy. Each qdisc is identified via a 32-bit classid.

Classes are either queues or qdiscs. Qdiscs further allow for more hierarchies as illustrated. The parent (in the hierarchy) qdisc will schedule its inner qdiscs/queues using defined scheduling algorithm (refer to a sample space further down). Each class is identified via a 32-bit classid.

[1] Inserting 100,000 rules using E3120 Xeon processor

Classifiers/Filters are the selectors of the packets. They stare at either packet data or metadata and select an action to execute. Classifiers can be anchored on qdiscs or classes. Each classifier type implements its own algorithm and is specialized. A classifier contains filters which implement semantics applicable to the classifier algorithm. For each policy defined, there is a built in filter which matches first based on the layer 2 protocol type.

Actions are executed when a resulting classifier filter matches. The most basic action is the built-in classid/flowid selector action. Its role is to sort which class/flow a packet belongs to and where to multiplex to in the policy graph [4]

TC flower classifier

The flower filter matches flows to the set of keys specified and assigns an arbitrarily chosen class ID to packets belonging to them. Additionally (or alternatively) an action from the generic action framework may be called. The Flower classifier is one of the classifiers which supports hardware offloading.

Brief Introduction to OVS

In a virtual server environments, the most common way to provide Virtual Machine (VM) switching connectivity is via a virtual switch. The virtual switch is basically a software that acts similarly to a Layer 2 hardware switch providing inbound/outbound and inter-VM communication.

One of the dominant virtual switches is OVS (Open Virtual Switch) which switches frames between local VMs on the host (sometimes called east-west traffic) and between local VMs and remote VMs (sometimes called north-south traffic). One major difference between OVS and a "regular" IEEE Ethernet bridge is that OVS switch "flows" as oppose to a regular Ethernet bridge which provides frame delivery between VMs based on MAC/VLAN. [5]

The OVS DP (Open vSwitch kernel datapath) is a match-action forwarding datapath. The information about the match and actions to be taken on the packet which matches are in a form of "flows". These flows are inserted, modified or removed by userspace.
OVS DP can be implemented using the Linux TC (Traffic Control) subsystem. The TC subsystem existed long before OVS DP and offers more flexibility. [6]

Motivation for Improving TC Filters Insertion Rate

Recent industry movement towards the use of VMs and OVS along with the accelerating speeds of IP networks led to growing number of new OVS connections. One of the ways to implement the OVS DP is by using the TC subsystem, each new connection entering the OVS possibly creates a new TC rule, hence improving TC rules insertion helps to improving the OVS performance. Until recently, existing TC code allowed for a poor rules-updates/sec rate. This does not satisfy the existing requirements. A work done recently has significantly improved the insertion rate, up to x500 faster than the older rate.
However, this recent improvement still does not satisfy the users who yearn for a rate of 1M/sec or better. In this paper we will discuss the existing situation, work done so far and ideas on adding batching operation to TC and more future architecture enhancements towards achieving this goal.

Review of the TC filter Insertion Flow[2]

To create a new TC filter rule, a user constructs a struct tcmsg message, wraps it inside a netlink[3] message, sets netlink message type to RTM_NEWTFILTER and sends it to the kernel. tcmsg message is composed of two parts: struct tc_msg (which contains instructions to the generic TC layer) and struct nlattr (which contains a list of attributes). Once the message reaches the netlink layer in the kernel, netlink calls a callback, which is actually tc_ctl_tfilter() method, which applies the following procedure:

1. Searches for the device
2. Looks for the qdisc specified in the TC message.
3. Tries to find a class attached to the qdisc
4. Within that class looks for proto-tcf (transport classifier) with the input priority.
5. If proto-tcf with this priority does not exist, creates a new one, according to the information provided in struct nlattr.
6. The classifier tries to look up the Qdisc handle by using the classifier (*get)() method and handles the success/failure for the lookup based on the flags specified in the netlink message.
7. If all is good and we have a valid handle, the classifier (*change)() method handles the new request. It reads additional parameters from "struct * nlattr": sets a new matching rule and action to act upon matching.

Current work

Analyzing the code revealed two major bottlenecks. The first bottleneck was found in step 6.

[2] In the following paragraph, the words filter and classifier are used interchangeably but refer to the same object.

[3] Netlink is used to transfer information between kernel and userspace processes.

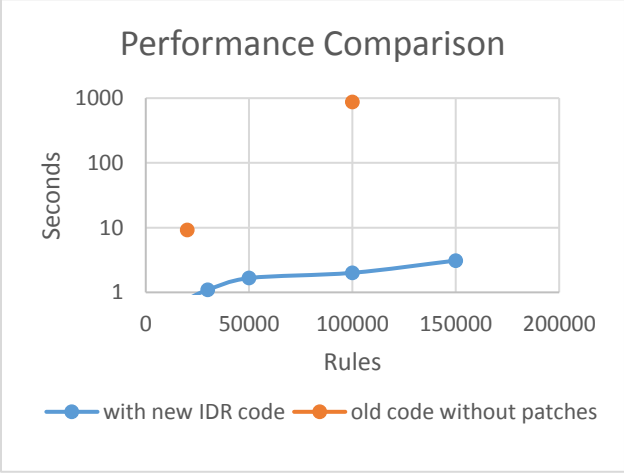
Several classifiers[4] were storing the existing handles inside a linear list which resulted in linear growth of any added/looked-for/removed handle from this list. In accordance to the past requirements this presented no problem. Since storing 100 or 1,000 rules and looking up can be done relatively sufficiently. However, moving to 1M entries means that adding the 1,000,000 rule would take 1,000,000 iterations (!).

The second bottleneck was found in step 7. Each action was stored inside a relatively small sized hash table (8 or 16 buckets), with each bucket points (once again) to a linear list.

The solution we suggested was to replace the two structures found in the bottlenecks into a radix tree. Looking further we ended up choosing IDR as the most suitable solution.

The IDR library is used in the kernel to manage assigning integer IDs to objects and looking up objects by their ID. It is essentially and almost O(1) operations library implemented on top of radix tree. Supporting add, remove, find and additional operations. [8]

This has significantly improved the insertion rate. Before this work: insertion of 100,000 rules took approximately 1,000 seconds. (Effective rate of 100 insertions/sec) insertion of 1,000,000 rules did not end within a reasonable time frame (several hours). After this work: Insertion of 100,000 rules took approximately 2 seconds while insertion of 1,000,000 rules took about 20 seconds. [9] [10]
This work has effectively improved the rate to a stable rate of 50,000 insertions/sec[5]. The following graph compares rate of rules insertion to the time it took. Since the improvement was exponential, the graph has logarithmic scale.



[4] As of Kernel 4.13 - Linear handle lookups were found in the following classifiers: basic, bpf, flower, flow. Other classifiers were either returning 0L or using hash table.

[5] Benchmarked on Intel Xeon E5645@ 2.40GHz+128Gb

Suggested Improvements

Analyzing the code further revealed that achieving minor improvement is possible with minor algorithm tweaks, but achieving a major improvement will demand more than a mere replacement of data structures. It will require profound change of the algorithm, or a parallel execution.

The RTNL Lock is a mutex located inside rtnetlink[6]. It is used to serialize rtnetlink requests by making sure no two threads may enter the rtnetlink subsystem at the same time.

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 TC classifiers. Furthermore, some code paths can hold the rtnl mutex for very long times (in the order of several hundreds of milliseconds in some cases). [7]

In TC, the RTNL Lock presents an interesting challenge. When a user sends multiple filter messages down the netlink messaging system and towards the TC layer, tries to insert or delete rules then only one message at a time may enter the tc_ctl_tfilter() method. Forcing any multi-threaded code to act as if it were composed of a single thread within a single process. Thus, multi-threaded rules insertion/deletion offers no benefit.

While this paper deals with ideas on how to improve rules insertion rate, the RTNL Lock is the foundation of the issues this paper aims to solve. In this paper we review several proposals towards solving this issue. We compare the advantages and disadvantages of the different approaches and finally recommend the solution which provides the best performance.

Solution 1: Removing the Lock.

Removing locks is always the best solution but tend to be the most difficult to achieve. As mentioned earlier, many kernel methods and drivers rely heavily 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. [7] However, there is a long way to go before this challenging work is completed and a solution for the problem can be reached as the required effort is enormous.

Even once Florian completes this work in the kernel, in order to take full advantage of the patches, hardware

[6] Rtnetlink is a netlink subsystem used to inspect or change networking related configuration. Rtnetlink stands for "Routing netlink"

vendors will have to make the necessary adjustments in order to make sure their propriety driver code runs properly without the RTNL Lock.

Solution 2: Multi-Threaded Batch under the Lock

The second approach is based on the pragmatic view that we have to live with the lock and the suggestion is to try to amortize the cost of the lock by implementing both batching and multi-threading.

The kernel already allows user space to batch multiple netlink datagrams in one message. However, once in the kernel receives the batched netlink messages, they are still processed independently and in series. So the only value brought by batching in this case is amortization of the cost of grabbing the lock, copying from kernel to user space etc. More granular serialization of individual netlink messages is achieved by other TC subsystems such as Actions; however even in that case the actions are added in series.

Our suggestion, which we discuss here, aims at improving things by replacing the serial ordering with parallel processing of the messages once in the kernel.

We do want to caution the reader that this approach will only work well if the individual batch entries are not dependent on each other.

Another issue with this approach is forcing the kernel to run a multi-threaded code and on several CPU cores even when the user application is single-threaded and he does not want to utilize additional CPUs.

This approach is divided into two smaller tasks: the first is accumulating the work under the lock and the second is executing the accumulated work, while under the lock.

Implementing Solution 2

Accumulating work

Suggestion 1: Multiple netlink messages.
Extending netlink interface by introducing batch operations. Batch operations are series of actions which are meant to be executed together. Two new netlink flags should be added: NLM_F_BEGIN and NLM_F_END.

Upon receiving NLM_F_BEGIN, the underlying rtnetlink subsystem will accumulate all incoming messages until a message with NLM_F_END arrives. Once a NLM_F_END flagged message was received a parallel execution is initiated as described in later section of this paper. Messages are accumulated in one list by copying (memcpy) the incoming messages to an internal buffer.

Accumulated messages list has to be maintained per user, with pre-defined quota (to avoid overflow) and with some aging mechanism.

This 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.

Suggestion 2: Compound netlink Message
Extend TC interface by introducing a compound TC message, RTM_BATCHTFILTER. This message will encapsulate multiple TC messages. Facilitating the work by sending all messages to be executed in TC layer in parallel at once.

The new TC message header is used to replicate attributes in nlmsg struct which should have been part of the tc_msg.

```
struct tcmsg_batch_hdr {
__u32     tcmsg_len;
__u16     tcmsg_type;        /* Message contents */
__u16     tcmsg_flags;

}
```

The compound message looks like a series of trio:

tcmsg_batch_hdr + tcmsg + nlattr and ends with a tcmsg_batch_hdr with len 0.

As for the attributes of struct tcmsg_batch_hdr:

tcmsg_len is the size of the trio. **tcmsg_type** contains same value as nlmsg_type would contain, i.e. a netlink message type, for example: RTM_NEWTFILTER. **tcmsg_flags** contains same value as nlmsg_flag would contain, for example: NLM_F_CREATE.

```
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 and runs in parallel until completion. In suggestion 1, a result of the running action is returned per netlink message. Same as a series of message without the new introduced batch flags. In suggestion 2, if all operations completed successfully then the netlink message return value is success. Otherwise the returning netlink error message contains a list of pairs

(msg index, error value). This message is returned per a compound TC message.

Comparison

Suggestion 1: Multiple netlink Messages.

1. Need to wait for NLM_F_END in order to start processing.
2. Need to memcpy() each message
3. Need to keep a list of messages per process/user.
4. Need to make sure each list doesn't exceed predefined size limit.
5. Possibly requires more than one system-call.
6. RTNL lock is might be taken more than once, for long batches.

Suggestion 2: Compound TC message

1. Can process first message in batch immediately.
2. No slow-down memcpy().
3. No internal bookkeeping.
4. No internal list size limitation.
5. Always a single system-call.
6. RTNL lock is always taken once. Max size of netlink message will be increased to include larger batch.

To conclude, Suggestion 1 is more generic, but suggestion 2 will undoubtedly deliver better performance.

Discussion

Both suggestions have parallel execution in common, possibly running requests out of order. Out of order execution can be an issue if the batch has inter-message dependencies.
There are two ways to handle this issue: forcing ordered execution and ignoring it.

When forcing ordered execution additional step of ordering the requests has to be supported by the classifiers. Adding a 'comparison' method to the classifier which allows TC to run topological sort to create an order. Another issue is failure handling - Once order is imposed and execution of requests runs in parallel a single request might fail. The ways to handle the failure is: full rollback, continue non-dependent requests execution and ignoring the failure.

Ignoring the order is easier and transfers the responsibility to the user. Therefore, to maximize utilization, we suggest to avoid inter-message dependency. The kernel will not have any mechanism to reinforce the order.

Additional measures have to be taken when supporting parallel execution: the classifiers code should be modified to support multi-threaded code and dependency on RTNL-Lock should be removed. Same applies to the driver's layer.

Since some of the classifiers and some of the drivers are still not fully 'multi-threaded compatible' we make another suggestion to add a 'capabilities' flag per classifier and per driver. TC will not allow any non- 'multi-threaded compatible' classifier or driver to run in parallel to a compatible one.

Conclusion

Improving TC filter rules insertion rate is vital for supporting contemporary virtual switches. The current TC rules rate is not sufficient. In this paper we presented several solutions to the problem. Removing the lock and 2 ways to do under-the-lock multi-threading. We strongly recommend on implementing the new TC compound message, which has the best performance by far, in order to support faster insertions.

References

[1] D. Awduche et al. – "Requirements for Traffic Engineering Over MPLS", Network Working Group - RFC 2702.

[2] D. Awduche et al. – "Overview and Principles of Internet Traffic Engineering", Network Working Group - RFC 3272.

[3] Martin A. Brown – "Traffic Control HOWTO". http://tldp.org/HOWTO/Traffic-Control-HOWTO/

[4] Jamal Hadi Salim – "Linux Traffic Control Classifier-Action Subsystem Architecture", Proceedings of Netdev 0.1, Feb 2015

[5] Rony Efraim, Or Gerlitz – "Using SR-IOV offloads with Open-vSwitch and similar applications", Proceedings of Netdev 1.2, Feb 2017

[6] Jiří Pírko, "Implementing Open vSwitch datapath using TC", Proceedings of Netdev 0.1, Feb 2015

[7] Florian Westphal, "RTNL mutex, the network stack big kernel lock", Proceedings of Netdev 2.2, Nov 2017

[8] Linux Inside, GitBook, https://0xax.gitbooks.io/linux-insides/content/Initialization/linux-initialization-9.html

[9] Chris Mi Commit, net/sched: Change cls_flower to use IDR, https://github.com/torvalds/linux/commit/c15ab236d69dd6dad24541400f460c47853803f8

[10] Chris Mi Commit, net/sched: Change act_api and act_xxx modules to use IDR, https://github.com/torvalds/linux/commit/65a206c01e8e7ffe971477a36419422099216eff