Extending TC with 5-tuple hash offload

Roni Bar Yanai, Rony Efraim, Ariel Levkovich

Nvidia
roniba@nvidia.com,ronye@nvidia.com,lariel@nvidia.com

Abstract

Network hardware continues to evolve and provide new opportunities for offloading a complete stateful network pipelines. One of the obstacles of using such HW offload was a common API that is expressive, extendable, and vendor agnostic on one side, and a common API that is a part of the Linux network path on the other side. The TC was the natural choice for that, and already today, it supports many HW offload configurations. HW offload capabilities can be used directly by calling TC or by using applications such as OVS that are using TC to offload their networking [1].

Keywords

TC, Offload, hash, ECMP, dp_hash

Introduction

Packet processing abstractly includes a sequence of tables, where each table is consisted of matches and actions. Match can be defined on subset of packet header fields (l2/l3/l4), and in some cases might include also meta data that is generated during the packet processing, for example SKB mark field. Actions can include actions that are modifying the packet, such as strip tunneling layer or add tunneling layer, rewriting the MAC addresses…etc. Actions can be modifying the packet meta data, and actions can set the packet faith, such as set a drop, mirror or redirecting of the packet to a port. Table structure can be constant, for example when looking on Linux netfilter, there are 5 logical tables on packet path, and the user can configure rules that change the packet (NAT for example), or redirect it…etc. Nowadays, the use of SDN becomes common practice, and there is a need for a much more dynamic structure of tables. Use of Open-Flow for example requires that tables can be added and removed dynamically, table chaining and tables rules structure is unknown in advance. OVS is very popular Open-Flow virtual switch solution that translates Open-Flow into optimized data plan multi table processing chain. OVS table structure is very dynamic and logical tables, identified by their reicrc_id, are added and removed on demand.

TC flower classifier is expressive tool that can describe matches using flow key defines [1] and may include matching on meta data fields extracted from the packet, such as tunneling metadata. TC actions include among others drop/ modify/ redirect and jump to next chain, and has the required expressiveness required by Open-Flow [1].

TC is already used for HW offload, and it is being used as the OVS API for HW offload when using Kernel data path.

TC Flower HW offload overview

TC flower already supports offload capabilities, in figure 1, it is shown how VXLAN strip can be offloaded to HW by configuration of the following rule:

```
#tc filter add dev vxlan0 protocol ip parent ffff: piro 1 \
   flower \
   dst_mac e4:11:22:33:45:61 \
   src_mac e4:11:22:33:45:61 \
   enc_src_ip 1.1.1.1 \
   enc_dst_ip 2.2.2.2 \
   enc_dst_prt 4789 \
   enc_key_id 16 \
   action tunnel_key unset \
   action mirred egress redirect dev ens1f0_0
```

figure 1

The rule will be offloaded to HW uplink, and for packets of type VXLAN with the matching outer IP address, MAC addresses and VNI, the unset action and port forwarding will be executed in HW. HW saves the parsing, and stack processing of UDP packets (VXLAN packets are directed by UDP socket of VXLAN port), and the VXLAN strip in kernel, going directly to port ens1_f0_0 in the example.

TC allows the expression of multi-table as well, and there is a goto action with chain id indicating the next chain.

```
#tc filter add dev ens1f0_0 ingress prio 1 chain 0 proto ip \
    flower \
    src_mac 24:8a:07:a5:28:01 \
    ip_flags nofrag \
    action goto chain 9
```

figure 2

HASH in data path

ECMP (Equal Cost Multi Path) is used to better utilize the network. In many network topologies a packet can take different best paths to reach to the same destination.
Balancing between paths will usually generate a better utilization of the infrastructure and make it less sensitive for single point of traffic load. The problem is that different paths may have different local loads and therefor different paths can generate different latencies. When TCP session packets are split between several paths, packet might arrive out of order because of the different latencies on paths. There is an initiative for TCP with multipath [3], but currently TCP stack can treat those out of order packets as packet drop and as a result, TCP stack will generate packet retransmissions and the congestion avoidance algorithm will reduce window size dropping the TCP throughput significantly. The same scenario is also valid for UDP. Although UDP RFC doesn't define UDP retransmissions or congestion avoidance control, in some cases upper layers can generate it. A very popular example is Google QUIC protocol. QUIC protocol is built on top of UDP and implements congestion avoidance algorithm similar to TCP, generating similar behavior in the described scenario.

To avoid out of order, the packets should be balanced in consistent manner per session. One way of keeping such consistency is allocating next hop per 5-tuple and saving it in some data structure. However, saving a data per 5-tuple has a performance penalty both in memory and search time, and in addition in case of network update, such as a node is removed there is a need to locate all relevant entries and remove them.

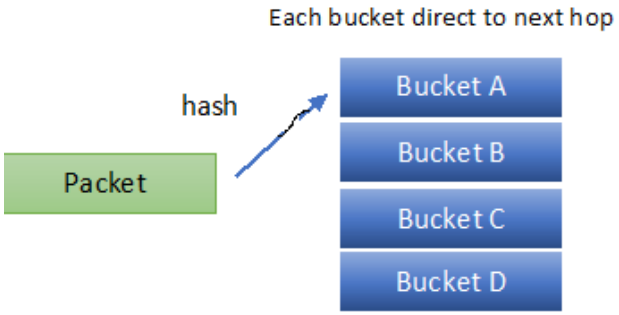

figure 3

A more efficient way is having two steps. The first step is calculating the hash on the 5-tuple. The second step is choosing one of the available buckets according to hash result as shown in figure 3. Each bucket contains the logic of sending the packet to the next node, which is usually setting the proper MAC address and send the packet to wire. The buckets and hash are functioning similar to hash table, where hash modulo table size, sets the bucket to use.
For such use cases it is better to use a fixed size table. Having a dynamic table size is a problem when update is required. For example, on network update such as when node is down, a bucket must be removed. The table size now changes, and it affects the entire traffic, namely changing all existing 5-tuple destination bucket. A more robust approach is having a predefined and a fixed number of buckets. On network update event, only an update of few specific buckets is required. For example, if node of bucket D should be removed, the bucket can be replaced by setting the same next node as in bucket B. In such update, only traffic that was destinated through bucket D would be affected. For having a good distribution, it is better to have a larger number of buckets and then replace each bucket of previous node D, with one of the other valid nodes. This will maintain the balancing, while affecting only the traffic that was previously sent to the down node D.

ECMP is a specific use case but hashing functionality can be used for other use case to achieve good load balancing. For example, it can be used in a Gateway that is load balancing between hosts. For each incoming packet the hash on the 5-tuple sets encap with the specific underlay and send to serving host. Redundancy in active-active mode is another good example, when hash can give a good load balancing, and steering can change with a small number of steps, namely updating the second table, in case of a fatal event.

Open-Flow generalize the problem and defines a concept of groups. Each group is a list of buckets, and for each bucket a list of actions is defined. There are different policies for choosing the bucket and hashing on 5-tuple is one of them. The user can achieve ECMP by setting the next hop or generate other load balancing using the concept of hash and buckets.

A high number of buckets in data path allows to generate weights (filling more buckets with next hop A in case of ECMP) and can support consistent hashing by moving only the relevant buckets.

TC hash HW offload

Extending TC: TC already has the expressiveness of matches, actions and chains. Adding such hashing capability will add a lot of flexibility, and will allow users to have their own type of service balancing/ECMP…etc. The policy, for example how to distribute weights, what happens when adding or removing nodes and how consistency is handled can be controlled by the user.

```
$ tc filter add dev ens1f0_0 ingress \
   prio 1 chain 0 proto ip \
  flower ip_proto tcp \
  action hash \
  action goto chain 2
$ tc filter add dev ens1f0_0 ingress \
prio 1 chain 2 proto ip \
flower hash 0x1/0xf  \
action mirred egress redirect dev ens1f0_1
```

figure 4

Figure 4 shows a match on TCP, action is hashing and goto chain 2. On chain 2, there is a match with a mask on the hash result, and an action. With the current example there can be up to 16 buckets (using 0xf as a mask). Each unique match is a bucket.

There is the question about where the hash result is kept and how it used. One option is to override the skb->hash which seems like a good option since value is kept beyond TC. Other option is using skb->cb[], that sets the hash boundaries to within the TC execution.

HW/SW hash consistency

Using the suggested expressiveness sets the ground for implementing ECMP, Service Balancing …etc using TC. However, when HW offload is added to the table, consistency becomes a problem.

The consistency problem raises because the exact way of hashing is not well defined, and it is very implementation dependent. When offloading to HW there is no way to communicate how hashing is done in SW, so HW path and SW path might generate different hash result.
For example, when executing hash, the Kernel usually uses the RSS value as the base value (if RSS exists) to save performance and on top of that there is an additional Jenkins hash calculation. In other cases, only Jenkins hash is executed, usually when the RSS value on SKB is not valid. The hashing function itself relies on where in the kernel path hash is calculated (after encap for example RSS is cleared), implementation also depends on the use case and in addition Kernel might choose a different hashing function or base in the future. From those reasons it will be hard to sync with HW. Moreover, HW vendors might want to implement hash function that are more efficient for HW and not to use SW efficient hash functions. In case that the HW and the SW don't use the same hash function we can get inconsistency as shown in figure 5. The inconsistency can happen in cases where first few packets of a 5-tuple session are handled in SW and only as a result, an HW rules is triggered.

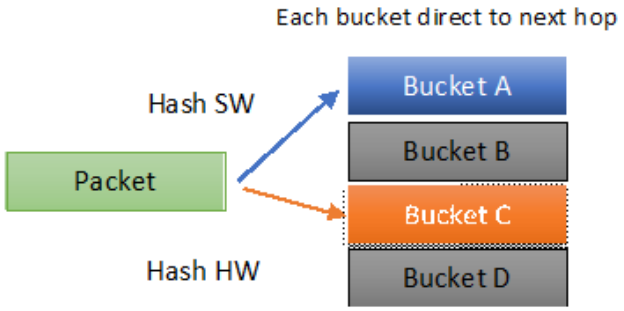

figure 5

As shown in figure 5, SW 5-tuple X is directed to bucket A after hash. HW 5-tuple X is directed to bucket C. In cases where everything known in advance and all options can be offloaded to HW before first packet is seen, the problem doesn't exist. However, for SDN controllers this is usually not the case. The data path is generated on demand and data path rules are not predefined.

The result of the described inconsistency depends on the use case. In a simple scenario it can cause packet re-order. In a more complex scenarios of service load-balancing it might cause a fatal failure of the connection, as HW will direct the packets to different host, e.g. SW path establish a TCP connection on host A, and HW will direct packets to host B, where the TCP connection doesn't exists.

In order to have consistency, there must be a way to synchronize or communicate the hash function. Since TC in HW offload case defines a table in the pipeline the hash on that point should be calculated the same in HW and in TC

Use of eBPF to communicate hash function

A way to communicate such hash calculation is to use eBPF. eBPF is generic enough for hashing calculation and already has access to both packet data and SKB. eBPF is secured, allowing user to define eBPF to be execute in the Kernel.

For every HW, the driver can supply an eBPF code version of the hashing used. On the SW path, TC will use the eBPF code provided to execute the hash, and by that it is guaranteed that HW and SW will have the same hash. Since TC configure HW tables (or chains), and also provide SW path with the same pipeline, having the eBPF code will generate the same hash on same table, either in SW or HW, and will synchronize the execution path.

The suggestion is to add eBPF object file that implements the hash to the TC rule as show in figure 6.

```
$ tc filter add dev ens1f0_0 ingress \
  prio 1 chain 0 proto ip \
  flower ip_proto tcp \
  action hash bpf object-file <file> \
  action goto chain 2
```

figure 6

TC now has hash action and also has the exact calculation on how to do it.

The driver will provide the hash eBPF object on the device /sys/ and in this way users can use the TC command and provide the right eBPF.

Criticism

The main argument is that TC already has a generic eBPF action that basically could generate the same result, namely having the right hash calculation and put it on skb->hash. Therefor there is no need in adding new hash action.

From HW offload perspective using eBPF action is not expressive enough, since the driver has no way of knowing that this is a hashing action. It is not expected that driver will analyze the eBPF code and "understand" this is hashing action. Adding some kind of enum on eBPF type is not different then stating hash action explicitly.

From user perspective, it is less straight forward to define actions by eBPF code, when maybe the name of the file itself is the only indication about what exactly is the expected action.

Generally speaking, we could say that many other explicit actions such as header re-write could be expressed using eBPF action, but the fact is that it has a dedicated action. The reason for that is that there is a need for tradeoff between generality, expressiveness and usability, and hash probably needs its own unique action.

Summary

As Networks become more and more dynamic there is a need for a more flexible way to balance between paths/ports/services…etc.

Use of TC transfers more control to the users in how such use cases are implemented and opens the door for many scenarios that are not supported currently or having some limitations.

Having HW offload on top of that will accelerate the performance significantly but also adds some challenges that are specific for the hash use case.

A way for extending TC with hash action was suggested with the use of eBPF code for solving HW/SW hash communication.

References

1.Simon Horman, "TC flower Offload", netconf 2.2, https://netdevconf.info/2.2/papers/horman-tcflower-talk.pdf
2. Cumulus, "Celebrating ECMP in Linux:", https://cumulusnetworks.com/blog/celebrating-ecmp-part-two/
3. "TCP Extensions for Multipath Operation with Multiple Address, " https://tools.ietf.org/html/rfc6824