Distributing Linux Traffic Control Classifier-Action Subsystem

Jamal Hadi Salim and Damascene M. Joachimpillai

Mojatatu Networks, Verizon
Ottawa, Ont., Canada and Waltham, Mass., USA

hadi@mojatatu.com, dj@verizon.com

Abstract

This paper will discuss distributing the Linux Traffic Control (TC) Classifier-Action(CA) subsystem packet processing across disparate nodes. The nodes could be a mix and match of containers, VMs, bare metal machines or ASICs.
A new tc Inter-Forwarding Engine (IFE) action is introduced based on IETF ForCES WG[1] Inter-FE LFB[2] work . The paper will go into both the implementation as well as the usage of the IFE tc action. Details on how to add new extensions to the IFE action will also be discussed.

Keywords

Linux, tc, filters, actions, qdisc, packet processing, ForCES, Software Defined Networking, iproute2, kernel

Introduction

The Linux Traffic Control Classifier-Action(CA) subsystem[3] provides a flexible way to compose packet treatment policies. A control application (such as the *iproute2 tc* utility) defines a policy in the form of an instance of a directed graph which constitutes a mix and match of filter and action instances through which selected packets traverse - with an end goal of achieving a specified packet service.

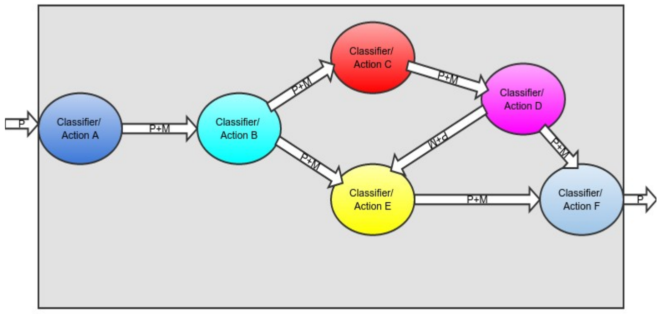

Figure 1: Packet Processing Policy Graph Instance

Figure 1 visualizes an arbitrary TC classifier-action policy graph which we will use for the purpose of providing context of our discussion.

Each graph vertex/node represents either a classifier or an action instance and the directed graph edges are labeled with a *P* to imply packet data and an *M* to imply metadata.
A policy graph is typically anchored at either the ingress or egress of a port[1]. Figure 1 ignores where the anchor point resides, again for the sake of illustration.
Packet data maybe edited (grown or shrunk) or consumed at any graph vertex. Packet metadata may be produced or consumed as well at the different vertices.
Figure 1 illustrates highlights a few architectural constructs of the CA subsystem[3]:

- Vertex B generates, alongside the packet data metadata providing additional details to the vertices downstream. Metadata may have local or global significance dictating its longevity. Locally significant metadata (such as the *skb cb[]* construct) may only last one vertex hop before being consumed or loosing its semantics. Globally significant metadata (eg the *skb mark* metadatum) could be modified at vertices along the way.
- The ability to branch is demonstrated at both vertices B and D in figure 1; at each graph vertex traversed that is capable of branching, a path decision is computed based on headers, metadata or stored state.

Note that a CA pipeline may be terminated at any point in the graph (example when a packet is queued or state dictates it is to be dropped etc). Also not illustrated is the fact a CA graph may contain loops.
A packet enters the graph in figure 1 at vertex *A*, which is always a classifier. The packet continues its journey from vertex A to vertex B. The packet exits, possibly modified or delayed at the policy graph terminal vertex F, without metadata (just as it came in).
For more details the reader is referred to [3].

In the next sections we will describe why one would want to distribute the graph such as the one in figure 1. We will then introduce the Inter-FE(IFE) action, the wire format used and the processing algorithms used. We will explore a few simple policies which make use of the IFE action and proceed to describe some of the challenges we had to overcome. Finally we are going to describe how to add extensions to the IFE action. In conclusion we will talk about future work we are intending to pursue.

1 Essentially a Linux netdev abstraction.

Distributing A Policy Graph

Often times there is need to take a specific instance of a packet policy and distribute it across processing node boundaries. The need for crossing node boundaries is often driven by a desire to scale a packet service horizontally, but sometimes by need to interact with specialized processing (look-aside offload engines such as crypto, TCAMs etc) or exception handling (packets requiring further inspection or treatment by specialized nodes such as controllers or service nodes).

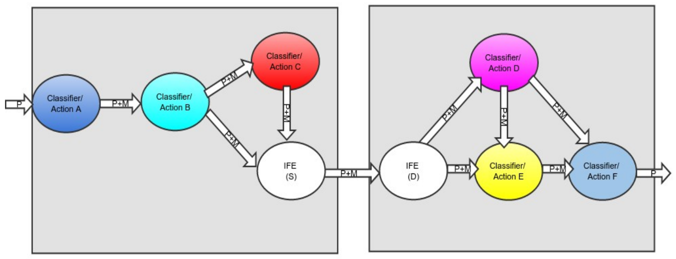

Figure 2: Distribute Policy Graph Across 2 Nodes

Figure 2 shows an equivalent graph to the one in figure 1 but split across two processing nodes.

In order for the two graphs to be equivalent, two requirements need to be met:

1. The distributed version needs to be able to propagate necessary metadata across processing nodes. This is achieved by introducing the IFE action which is the topic of this paper.
2. The number of input and output edges to each vertex stays unchanged (implying the implementation does not change).

Introducing The IFE action

The IFE action plays the role of a graph connector.

A source IFE instance(*S* in figure 2) is placed at the egress of a source processing node port and another at a downstream processing node's ingress port(*D* in figure 2) as illustrated in figure 2.

The egress IFE instance is instructed to take metadata as defined by policy and relay to the destination node. The ingress IFE is programmed to accept policy-specified metadata and pass it on the next vertex in the graph.

Figure 3 shows how the IFE data is encapsulated within ethernet frames.

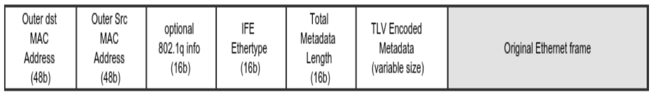
Outer dst MAC Address (48b)
Outer Src MAC Address (48b)
optional 802.1q info (16b)
IFE Ethertype (16b)
Total Metadata Length (16b)
TLV Encoded Metadata (variable size)
Original Ethernet frame

Figure 3: Inter-FE Ethernet Encapsulation

Egress processing

As illustrated in figure 3, the original ethernet frame is encapsulated inside the new ethernet frame, in addition:

- The original ethernet headers (source, destination MAC address and optional vlan information) may be used for the outer header or maybe over written with new programmed values.
  - The outer destination MAC address identifies the intended destination processing node.
  - The outer source MAC address provides the identity needed for the source processing node.
  - The optional VLAN tag information may be present if the original ethernet header had vlan information.
- Programmed metadatum are encapsulated, each in its own TLV. The 16 bit *type* field of the TLV uniquely identifies each transported metadatum.
- The total metadata size (including 16 bits for the length header) is included in the Metadata Length field.
- A standard 16 bit ether type field is set to a value recognized for IFE. The default value is 0xFEFE[2].

Allowed metadata (as specified by policy) to be encapsulated is selected in one of two ways:

1. Extracted from the skb or other kernel constructs (eg conntrack details) at runtime. The policy-data must specify that the specific metadatum is to be allowed.
2. Statically set at configuration time. Statically provisioned metadata values override runtime values. Refer to the egress processing algorithm below for details.

The egress processing algorithm is as follows:

- Increment action instance statistics for packet and byte count observed.
- Apply runtime metadata against the programmed metadata filter list. If no legitimate metadata is found that needs to be passed downstream, then increment stats (the *overlimits* stat is abused for this purpose) and the packet is allowed through as is.
- Create the outer ethernet header which is a duplicate of the incoming frame's ethernet header. The outer ethernet header may have an optional 802.1q header (if one was included in the original frame). Set the ether type to 0xFEFE.
- If the policy specifies the optional destination MAC address, use it.
- If the policy specifies the optional source MAC address, use it.

2 Value selected to symbolize *FE* to *FE* node encapsulation.

- If the policy-data specifies the optional ether type, override the current (0xFEFE) value with the specified value.
- Walk the policy metadata filter list again. For each metadata specified, either encapsulate the runtime metadata or the specified static value.
- Note: At any steps above should an error be detected, increment the error stats and ask for the packet to be dropped

Listing 1 below shows a sample tc policy for encoding Iat an FE on a source processing node instance. The listing is intended to illustrate features as opposed to being practical.

```
tc filter add dev $ETH parent 1: protocol ip prio 10 \
u32 match ip protocol 1 0xff flowid 1:2 \
action ... action ... \
action ife encode type 0xDEAD \
allow mark use hash 10 use qmap 17 \
use mystring “foobar” \
dst 02:00:00:22:01:01 src 52:54:00:c3:4b:c5 \
action ...
```

Listing 1: IFE action egress sample policy

The policy-data specifies to:

- use an ether type of 0xDEAD to override the default 0xFEFE.
- Override the source(*src*) and destination(*dst*) MAC addresses.
- Use (i.e encode) the runtime *skb->mark* value.
- Ignore the *skb hash.* Instead send to the remote node an encapsulated value of 10. In other words the runtime value of the *skb hash* is over-ridden by the static policy defined value.
- Ignore the runtime *skb queue map* and instead always encapsulate a value of 17.
- Pass an arbitrary string metadata not related to any kernel structures a value of “foobar”. This is an example of how one would program an arbitrary metadata value.

When the IFE action completes, the next action or classifier in the graph is traversed (typically this would be a *mirred* action[3] to redirect a packet to another port or a vlan action to add a vlan tag to a packet etc).

Ingress processing

The target destination node receives IFE-encapsulated packets at its ingress as shown in figure 2.

The IFE ethernet frame is recognized by its ether type. Either the default 0xFEFE is observed or a different value that is agreed on is used(in which case an administrator or controller would program both ends).

The ingress processing algorithm is as follows:

- Increment statistics for packet and byte count observed.
- Parse the metadata list and for each of the metadata received compare against the allowed metadata list
    - If the metadatum is allowed, extract it and install it (typically on an skb, but could be on other structs etc).
    - If the metadatum is recognized but not allowed or content is malformed increment stats and ignore.
    - If the metadatum is not recognized ignore it but increment stats (overlimits stats are abused for this).
- Consume the outer header.
- Note: At any steps above should an error(such as malformed metadata) be detected increment the error stats and ask for the packet to be dropped.

When the ingress processing completes the packet and restored metadata are passed to the next action or classifier in the policy definition or simply up the network stack.

```
tc filter add dev $ETH parent ffff: prio 2 protocol 0xdead \
u32 match u32 0 0 flowid 1:1 \
action ife decode allow mark reclassify
tc filter add dev $ETH parent ffff: prio 5 protocol ip \
handle 0x11 fw flowid 1:1 action .....
```

Listing 2: IFE action ingress sample policy

Listing 2 shows a sample ingress policy. A higher priority filter rule using the u32 classifier is encountered first by the packet. The filter looks for the ether *protocol 0xDEAD* and ignores any other packet headers *(u32 0 0* construct*).* When it is matched, the ife action is invoked to decode the IFE data encapsulated.

The policy in this case is only interested in the *skb mark* field (*allow mark* construct). If the mark is observed within the IFE encapsulated data, it is extracted and its value set on the *skb mark* field. The packet is then run via the classifiers again (as per policy construct *reclassify*[3]).

The second classification is done by the *fw* classifier (first one to match ethernet *protocol ip*). This classifier upon matching the *skb mark* value 0x11, would pass control to the next action in the graph if indeed the IFE action had earlier extracted mark value of 0x11.

Overcoming Challenges

It would not be fun if we did not have challenges to overcome.

MTU Challenges

The diligent reader would immediately notice that the IFE action would enlarge the ethernet frame. This could cause MTU issues. To address this:

- Limit the amount of metadata that could be transmitted to fit within an MTU. We have a flexible implementation which allows filtering on what metadata goes on the wire[3].
- Use large MTUs when possible (example with jumbo frames).
    - Note: Given that the IFE action is expected to operate within the realm of Layer 2 and will deal with virtual environments, we expect large MTUs will be a common setup[4].

Ethernet Type Challenges

While we expect to use a unique IEEE-issued ether type for the inter-FE traffic, we use lessons learnt from VXLAN deployment to be more flexible on the settings of the ether type value used. Linux VXLAN implementation uses UDP port 8472 because the deployment happened much earlier than the point of RFC publication which prompted IANA to assign udp port 4789. For this reason we make it possible to define, at control time, what ether type to use and default to the IEEE issued ether type. We justify this by assuming that a given setup is likely to be owned by a single organization and that the organization's administrator or controller would be responsible to program all participating processing nodes.

Metadata IDs

While the ForCES approach is to standardize the metaids (Including leaving some space for private use), in an organization under the same administrator it is possible to just standardize on a private space.

Metadata Propagated

A few obvious skb metadata are currently supported. These are:

- 32-bit skb *mark* (optionally with a 32-bit mask). Metadata id 1
- 32-bit skb *prio.* Metadata id 3
- 16-bit skb *queue mapping.* Metadata id 4
- 32-bit skb *hash.* Metadata id 2

The IFE action is designed to offer a simple interface to add more types of metadata that can be transmitted across inter-forwarding boundaries as demonstrated in the next section.

Extending The IFE action

A core feature of the IFE action is to allow easy addition of metadata handling in the kernel. To this end we have provided a simplified kernel module API.

The module api provides methods for:

- checking presence of the metadata via *check_presence()* method. This method would be the one that decides where to retrieve the metadata value from a runtime value or to use a statically defined policy value.
- Metadata encoding on egress via *encode()* method.
- Metadata decoding on ingress via *decode()* method.
- Encoding the metadata when the control side requests for it via the *get()* method.
- Allocating space for the metadata via the *alloc()* method.
- Freeing of the metadata space via the *release()* method.

The module author is expected to be able to present implementations for the above methods. For basic metadata like 32 or 16 bit definitions, we provide some of the basic utility functions.

Listing 3 shows a sample use of the metadata methods for the skb hash metadata.

The development of a metadata extension involves:

- The user specifies a *struct tcf_meta_ops* and datafills it with all the required details (ops etc).
- The user implements the methods/ops required.
- At module initialization, register the *struct tcf_meta_ops* using *register_ife_op()* API call.

3 Administrators for ethernet-extending protocols commonly set the egress MTUs to be just enough to allow for allowed maximum wire size minus extra space needed. Care needs to be taken to not go too low (leave about 576B for IPV4 and 1260B for IPV6)

4 The MTU for loopback device on my laptop is 64K. And just as large for veth.

```
static struct tcf_meta_ops ife_hash_ops = {
        .metaid = IFE_META_HASHID,
        .name = "skbhash",
        .check_presence = skbhash_check,
        .encode = skbhash_encode,
        .decode = skbhash_decode,
        .get = get_meta_u32,
        .alloc = alloc_meta_u32,
        .release = release_meta_uxx,
        .owner = THIS_MODULE,
};
static int __init ifeprio_init_module(void)
{
return register_ife_op(&ife_hash_ops);
}
```

Listing 3: skb hash metadata operations

At runtime on an egress node, when the metadata filter list indicates a specific metadata is allowed, then its encode method is invoked. Listing 4 shows the encode method for the skb hash metadatum.

```
int skbhash_encode(struct sk_buff *skb, void *skbdata,
struct tcf_meta_info *e)
{
        u32 skbhash = skb->hash;
        if (e->metaval) { /* use static value */
                skbhash = *(u32 *)e->metaval;
        }
        if (!skbhash)
                return 0;
        skbhash = htonl(skbhash);
        return tlv_encode(skbdata, e->metaid, 4, &skbhash);
}
```

Listing 4: The encode method for skb hash

At runtime on an ingress node, when a policy metadata filter list indicates a specific metadata(found encapsulated in the received packet) is allowed, then its decode method is invoked.
Listing 5 shows the decode method for the skb hash metadatum.

```
int skbhash_decode(struct sk_buff *skb, void *data,  u16 len)
{
         u32 shash = *(u32 *) data;
        shash = ntohl(shash);
        skb->hash = shash;
        return 0;
}
```

Listing 5: skb hash decode method

The reader is referred to the kernel code for the IFE action for more samples and fine-grained details.

Offloading IFE

It is possible to write extensions that take existing hardware metadata carrying approaches such as Broadcom Higig[5] and map them to IFE metadata thus extending the policy graph across ASICs.

We believe that the IFE metadata sourcing and termination is easy to implement in hardware. A smart NIC at a receiving path, essentially parses the metadata and makes it available via dma descriptors ready for consumption via the stack.

Sample use Cases

There are many possible use cases for distributing a policy graph as described earlier.

Pipeline-stage Indexing

An earlier motivation for us was to scale packet processing. So we use metadata to carry pipeline-stage indexing information for systolic packet processing reasons.

You start with a single processing node and then as your performance needs grow you split the functionality into multiple machines thus horizontally scaling. In such a case the processing pipeline is built such that the more expensive functionality is parallelized.

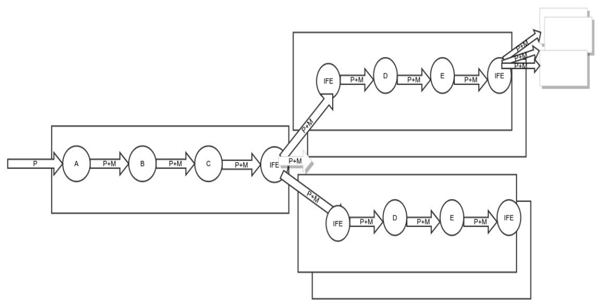

Figure 4: Scaling by Splitting

Figure 3 shows how to scale a policy graph into pipelines across additional processing using the IFE action as the *split* point. Each east-bound node is identified via its MAC address of the receiving port.

The reverse direction also uses the IFE action as the *merge* point.

Other use Cases

The flexibility provided by the IFE action offers more opportunities. A few examples of metadata that could be attached for a variety of processing:

- OAM information – example turn on some packet debug information on a need basis.

- Exception handling information – example VXLAN service handling.
- Authentication and authorization information.
- Versioning information.
- Compliance information.
- Service Identifiers.

Integrating In A Controller Environment

We have illustrated how one would control the IFE policies over standard *tc* cli tooling. However, this could get cumbersome as the number of nodes grows (very extreme when you start using containers). For this reason it would make sense to automate the process with the use of a centralized controller.

To that end we have implemented policy control involving the IFE and other graph nodes via the ForCES architecture. The IFE action is modeled as a ForCES LFB(Logical Functional Block).

Future Work

We have not yet done good performance measurements. We do expect to see slight increases in latency when a processing graph is split across nodes, but believe the overhead will be small due to the fact we are running directly over ethernet. We will be publishing performance numbers in the future.

We plan to prototype hardware offloading ideas via the rocker[4] device or an offloaded network processor.

We are also exploring ways to extend usability at user space tc level of the metadata control such that simple new metadata extensions do not require any code changes in the tc utility.

Clearly, for sanity of inter operation, standardization is needed. Our intention is to be able to, in the future, discover these metaid values by querying the kernel; for now we have specified our own Linux values as shown in the next section.

Acknowledgements

The preparation of these instructions and the LaTeX and LibreOffice files was facilitated by borrowing from similar documents used for ISEA2015 proceedings.
The authors would like to thank Evangelos Haleplidis for providing feedback on this paper.

References

1. https://datatracker.ietf.org/wg/forces/documents/
2. https://datatracker.ietf.org/doc/draft-ietf-forces-interfelfb/
3. Jamal Hadi Salim, “TC Classifier-Action Architecture”, Proceedings of Netdev 0.1, Feb 2015
4. Scott Feldman, “Rocker: switchdev prototyping vehicle”,Proceedings of Netdev 0.1, Feb 2015
5. https://en.wikipedia.org/wiki/Higig

Authors Biographies

Jamal Hadi Salim has been dabbling on Linux and open source since the early to mid 90s. He has contributed many things both in the Linux kernel and user-space with a focus in the networking subsystem. Occasionally he has been known to stray and write non-networking related code or even documentation. Jamal has also been involved in what kids these days call SDN for about 15 years and co-chairs the IETF ForCES Working Group.

DJ is responsible for Software and Systems Architecture of Verizon’s Network Evolution program. He has paid for his sins already with many years of Network Development and is now focused on large scale network deployment. He has a keen interest in parallelism of the Network Processing applications and is a big fan of open networking in which he believes Linux plays an important role.