Linux Traffic Control Classifier-Action Subsystem Architecture

Jamal Hadi Salim
Mojatatu Networks
Ottawa, Ont., Canada

hadi@mojatatu.com

Abstract

This paper describes the Linux Traffic Control (TC) Classifier-Action(CA) subsystem architecture. The subsystem has been around for over a decade (long before OF or P4) in the kernel (and longer in experimental patches before that) and we are finally motivated to do the boring unpleasant part of any open source project – documenting. We will describe the packet-processing-graph architecture and the underlying extensibility offered by the CA subsystem; we will further discuss the formal language that makes it an awesome packet processing architecture.

Keywords

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

Introduction

The Linux Kernel offers a very rich packet-processing framework.

Figure 1: Packet Processing Path

Figure 1 visualizes a simplified layout of a classical packet path. Packets come into the ingress of a port and go through a variety of processing components illustrated as the cloud above. Outgoing packets emanating from the illustrated cloud end up at an egress port to proceed with their journey.
As can be seen, the port is the anchor point of a packet processing path(or graph).

Ports

In the Linux world, ports are known as network devices or netdevs[1] in short. We are going to use those two terms inter-changeably.

Ports typically have a single datapath packet input and a single packet output in the east-west direction as shown in Figure 1. The datapath entry points are also split into an ingress as well as an egress. It is possible to have a port with only one of those entry points[2].

Ports[3] have north-south interfaces for control as well as eventing (mostly via the netlink[3][4] API). Control and configuration utilities such as *iproute2* or *ifconfig* make use of these north-south interfaces.

In the east-west direction a netdev implements some datapath activity.

The simplicity of the netdev abstraction has resulted in its wide adoption. Most port abstractions are ethernet-derived (meaning they will have L2 addresses that are ethernet MAC addresses). This makes it easy to generalize a lot of the tooling. For example, one can run *ip* command on any netdev regardless of what it implements; one could attach L3 addresses to a port, point a route at it, refer to neighbors (eg ARP/ND) on its egress, etc. In general it means netdevs can be composed in a traditional packet processing graph with L3 processing (attaching an IPv4/6 or even a DECNET address), L2 or other more exotic setups such as stacking netdevs (tunneling, bonding etc).

What netdevs do in their processing of received packets varies widely. They may be used for handling hardware or virtual abstractions. A few examples of netdev implementations are:

- lo. The host loopback. Takes a packet and loops it back. The processing scope is within the host
- Physical ports. Variety of physical controlling netdevs: wired ethernet chips, USB, wireless, CAN, etc. typically named *ethx*.
- Tuntap. Handles packets shunting between kernel and user space. A user application writes packets to a file descriptor which show up on the egress of a tuntap. Likewise reading the file descriptor, a user space application receives packets that show up on the ingress of the tuntap. Tuntap is popular as a virtual NIC for Virtual Machines as well as in user-space based packet-processing.

1 Not to be confused with netdev which stands for *network developer*. It is arguable which semantic the netdev mailing list is derived from.

2 The author is aware of the dummy netdev which takes in packets but does not egress them. Can not think of a good use case for the opposite.

3 In the SNMP world, Linux ports would be equivalent to the abstraction known as *interfaces*.

- A variety of tunnels. You name it, Linux has it: VLAN, GRE, VXLAN, IPIP, L2TP, Ipv6 over V4, etc
- MACVlan. Started as a way to represent individual MAC addresses on multi-mac physical ethernet ports but is getting feature-fat. It can act as a bridge or direct hardware offload into a container or host.
    - MacVTAP does the offloading of hardware directly onto VMs the same way MACVlan does it for containers
- veth. Implements a pipe pair of netdevs. Packets injected on the egress of one half show up on the ingress of the other half. Typically, one half of the pipe sits in a host and the other inside a container (although there are use cases where each half sits in a different container).
- Bonding/Team. Device aggregation, LACP etc. Has more features than a swiss-army knife.
- IP over Infiniband
- Bridge. Feature-rich IEEE bridge abstraction
- IFB. Intermediate device used to aggregate processing graphs, typically for flow QoS.
- Dummy. Takes a packet, accounts for it then drops it.
- Many, many others. Look at the code (or write one).

The Linux Datapath

Figure 2 expands on the general Linux datapath and exposes more details of the cloud illustrated in figure 1.

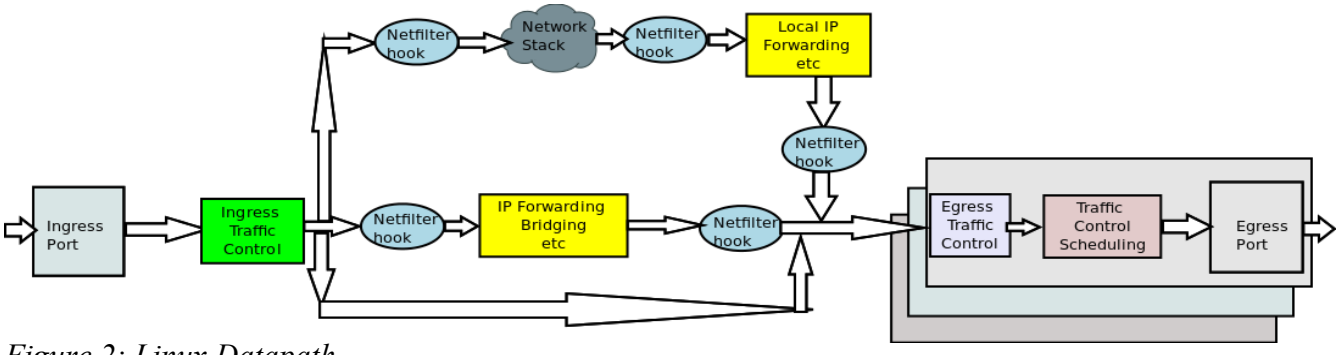

Figure 2: Linux Datapath

Several building blocks are shown. About all building blocks and their components as described in this document can be written as kernel modules. Control of all the datapath blocks is done from user-space (in the traditional unix sense of separating policy from mechanisms; the datapath being the mechanism implementation).

In this document we are going to focus on parts of the Ingress and egress Traffic Control, but we feel it is important to talk about the other blocks so as to provide context.

Netfilter

Netfilter[1] is a feature-rich, modular, extensible packet-processing framework. It provides a set of hooks at strategic Linux kernel packet-processing points that allows kernel modules to register callback functions. A registered callback function is then invoked by the kernel for every packet that traverses the respective hook within the network stack. The hooks (see figure 2) are labeled according to their position in the datapath

- *Pre-routing*,
- *Forwarding*,
- *Post-routing*,
- *Input* (to the local network stack),
- *Output* (from the local network stack)).

Netfilter[1] has too many powerful features that we would not be doing it justice in this short description, but we wanted to highlight a few features it possesses:

- Packet filtering
- All kinds of network address and port translations one can think of
- Multiple layers of user-visible control and datapath APIs for writing applications and datapath components.
- Stateful connection tracking capability known as *conntrack*[2].

Note: Figure 2 shows a bypass of the stack from ingress to egress. This is achieved by the CA subsystem with mirroring (to possibly many ports) or redirect (to one port).

Linux Traffic Control Overview

Figure 2 shows 3 blocks labeled with the term *Traffic Control*. Lets start with some concept definitions and use figure 3 which shows the egress path and combines the concept of *Egress Traffic Control* and *Traffic Control Scheduling* illustrated in figure 2.

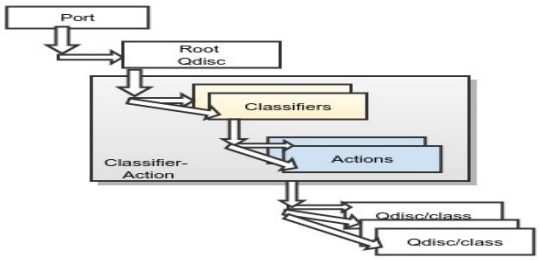

Figure 3: Egress Traffic Control Layout

- Queueing Disciplines (qdiscs) are scheduling objects which may be *classful* or *classless*. When classful, the qdisc has multiple classes which are selected by *class*ifier filters. Given classful qdiscs can contain other qdiscs, a hierarchy can be setup to allow differentiated treatment of packet groupings as defined by 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 some defined scheduling algorithm – refer to a sample space further down. Each class is identified via a 32-bit classid.
- Classifiers are selectors of packets. They stare at either packet data or meta data 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 primitive 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.

A port has two default qdisc anchor points attached to it. The *root qdisc* is anchored at the egress path as illustrated on figure 3. On the the ingress path the *ingress qdisc* as shown in figure 4 exists.

The ingress qdisc is a dummy queuing discipline whose role is to provide anchors to the CA subsystem.

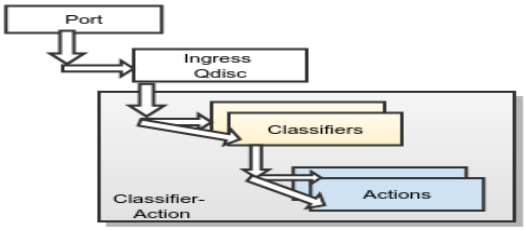

Figure 4: Ingress Traffic Control

A packet received by the ingress qdisc is essentially passed to the CA subsystem for processing based on the policy graph definition.

As mentioned, each qdisc/class is identified by a 32-bit id known as the *classid* or *flowid*. The *classid* is split into a 16 bit major id and 16 bit minor id. So one would see *x:y* used where *x* is the major number often referring to the hierarchy level and *y* is the minor number referring to entities within a hierarchy. Major number 0xffff is reserved for the ingress qdisc.

It is out of scope of this paper to describe qdiscs but we will mention a few to provide context and show variety:

- Prio. Flows selected internally based on TOS or skb->priority fields. Work conserving scheduling algorithm based on strict priority sorting (meaning low prio packets may be starved).
- Pfifo. Classless qdisc with a single FIFO queue. Packets are scheduled out based on arrival first in first out (FIFO).
- Red. Classless qdisc with scheduling based on the RED[7] algorithm.
- tbf. Classless qdisc, non-work conserving scheduling algorithm for shaping that is based on token bucket algorithm.
- htb. Hierarchical(classful) qdisc extension to TBF.
- Sfq. Stochastic fair queueing loosely based on [5].
- codel. Based on controlled delay algorithm[6].
- fq-codel. Extending codel with a sfq flavoring.
- Netem. Provides a variety of network emulation functionality for testing protocols by allowing mucking around with the protocol properties and semantics[14].
- Many others. The plugin definition is well defined. Write one today.

Although Figure 3 shows no presence of CA anchoring on the lower hierarchy qdiscs, it should be noted that one could create policies anchored at any qdisc or class[4]. We are leaving out details because we feel that discussion is out of scope for this document.

Some History of Linux Traffic Control

This paper would be incomplete without a little reminder of the past.

4 Depends on the qdisc implementation. Most allow for this.

Alexey Kuznetsov pioneered the Linux Traffic Control[5] architecture. Alexey submitted the initial code patches which first showed up in kernel version 2.1.

Werner Almesberger[6] did a lot of work in the formative years[10][9][8] including authoring all the ATM related work.
The author contributed the action extension to the CA subsystem and is current maintainer of the CA subsystem.

Classifier-Action Subsystem Overview

The main role of the CA subsystem is to look at incoming packet data and/or metadata and to exercise a mix of classifier filters and action executions to achieve a defined policy.
There is a strong influence of the unix philosophy of composability in the approach to packet processing in the CA subsytem. Any classifier or action can be written as kernel module and a policy can be used to stitch together the different components for a defined outcome.

We are going to use the popular *tc* utility as a guide throughout the rest of this document to highlight the different architectural constructs; however, we want to point out that any application can use the same netlink API used by *tc* and therefore the features are not tied to *tc*.

```
tc  filter add dev $DEV parent 1:0 protocol ip prio 10 \
u32 match ip protocol 1 0xff \
classid 1:10 \
action ok
```

Listing 1: Simple Egress Classifier-Action Policy Addition

Listing 1 shows a simple egress policy addition using the *tc* BNF grammar. We use Listing 1 to describe some important attributes of the CA subsystem:

- Attachment point: to an arbitrary port *$DEV egress qdisc* with id *1:0*
- Built-in match: Matches on ipv4 packets (*protocol ip*)
- A filter *prio*rity of *10*
- A specified classifier type filter to use: the *u32* classifier to match icmp (*protocol 1 mask 0xff*) packets. Note each classifier will have attributes proprietary to itself as shown above.
- Upon filter match:
  - Built-in action: Selects a queue/qdisc with *classid 1:10*
  - Programmed action: Accepts the packet (*action ok*). Note each programmed action will have attributes proprietary to itself as shown above.

5 In addition to a *lot* of very creative things Alexey has done over the years.

6 Werner was hacking on everything from LILO (most popular Linux bootloader) to file systems in those days

Classifiers And Filters

As shown in figures 3 and 4, and illustrated in Listing 1, the qdisc is used as the policy filter attachment point.
The built-in filter is described in the policy using the *protocol ip* syntax. The built-in filter lookup is very efficient since the protocol type is already in the cpu cache at the point the lookup happens (*skb->protocol*). The built-in match very quickly discriminates what policy tree to use.

Illustrated in Listing 1 as well is a *priority* attribute tied to each filter. Filters are kept in a priority-ordered list for each protocol(i.e matching built-in classifier).
It is possible to configure two filter policies on the same protocol to match the same meta/data but have different action execution graphs[7] which then merge them to provide graph continuity as will be shown later. The purpose of the priority field is for ambiguity resolution in case of such filter conflicts. Lower priority values are more important.

Listing 1 also shows the *u32* classifier type used in the policy definition. The CA subsystem provides a plugin framework to define arbitrary classifiers.
There are many classification algorithms supported, (not exclusive) examples include:

- u32. Uses 32 bit value mask chunks on arbitrary packet offsets for filter matching. A very efficient protocol parse tree can be built with this low level classifier.
- fw. A very simple classifier that uses the skb mark metadata to match.
- route. Uses ip route attribute metadata (like route realms) to match.
- rsvp. Classification based on RSVP[13] filtering definition.
- basic. A collection of smaller classifiers that can be combined into a more complex match.
- BPF. Based on Berkley packet filter[17][21] as the match engine.
- Flow. A collection of various packet flow and common meta data selectors (including conntracking, user ids, group ids, etc) that can be combined into a complex policy.
- An Openflow classifier[14]. An N-tuple classifier that looks at all the packet fields defined by the OF specs (There were about 18 at the publication of this document.
- Many others. If you dont like any, write your own.

Note that each of these classifiers may have its own internal structuring with its own elements[8]. Figure 5 illustrates a parse tree for a layout of the u32 filter setup.

7 Sometimes this is for adding backup filter rules so when the higher priority filter is removed subsequent incoming packets are matched by the remaining lower priority filter.

8 Each element would have a 32 bit identifier. Refer to u32 classifier in figure 5 for an example of element ids.

Very efficient parse trees can be created for a specific setup but require attention to detail[9].

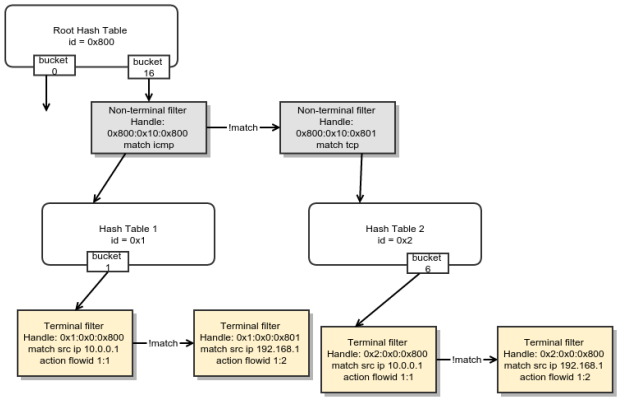

Figure 5: u32 Classifier Parse Tree

Classifier Design Principles

There are two guiding principle for the classifier architecture:

1. *It is not possible to have a universal classifier* because underlying technologies are constantly changing and there is always some new academic *flavor-of-the-day* classification algorithm. For example, the illustrated *u32* classifier matches on skb data using arbitrary 32-bit value/mask pairs, at arbitrary packet offsets; whereas the *fw* classifier matches on *skb->mark* metadata.
2. *Sometimes we need more than one classifier type*, each with different capabilities, to properly match a given signature. So an important CA design guideline is to allow for multiple classifier types to be used when needed by a policy.

Based on these principles, it is possible to match first based on one classification algorithm (example *u32* or *bpf*) then subsequently on a different algorithm (example *ematch* text classification with string matching via Boyer-Moore Algorithm[16] or via Knuth-Morris-Pratt algorithm[17] or whatever the latest text search algorithm is)[10]. The CA design choice has fostered innovation by not allowing monopolies of chosen classification algorithms.

Actions

Listing 1 illustrates the actions to be executed when the filter matches.
As mentioned earlier, there is a built-in action which selects the class/flow/queue. Listing 1 shows the built-in action selecting a *class id 1:10*.

9 Good fodder for automated policy generation and display.

10 Hardware commonly has a lookup using TCAM which may be followed up with lookup on RAM but we have more freedom in software therefore there is no need for a software architecture to be limited.

The policy of Listing 1 further describes that the packet be accepted and allowed to proceed in the processing pipeline. By design, actions adhere to the unix philosophy of *writing programs that do one thing and do it well*. One could for example compose a policy with a series of actions in a unix-like *A|B|C|D* pipe where each subsequent action refines further the previous action's work-result and just as in unix the pipeline could be terminated by any action in the pipeline. More on this when we talk about programmatic action pipeline.

Each action type instance maintains its own private state which is typically updated by arriving packets; but sometimes by system activities (timers etc).
To illustrate a few semantics of actions, lets show some of the attributes of the action programmed from Listing 1 retrieved using *tc* again. The output is captured after sending 10 pings to a remote location and is shown in Listing 2.

```
action order 0: gact action pass
 index 1 ref 1 bind 1 installed 32 sec used 15 sec
Action statistics:
Sent 980 bytes 10 pkt (dropped 0, overlimits 0 requeues 0)
backlog 0b 0p requeues 0
```

Listing 2: Action Runtime Policy Details

The action *pass* is subcomponent of the *gact* (generic action, more later). In this case several things are illustrated[11]:

- 32-bit system global per-action-type instance identifier of the action (*index 1*) which uniquely identifies the action type instance.
- how many policy graphs the action is bound into (*bind 1*); one in this case. Actions could be shared on multiple policies (more on this later).
- Age: How long ago the action was installed (*32 seconds*).
- Activity: When the last time the action was used (*15 seconds* ago).
- A bunch of standard statistics like how many bytes(380) and packets(10) were processed by the action. Other stats we can ignore for now for brevity; sufficient it is to say actions can add extended stats.

A few examples of actions include:

- nat. Does stateless NAT
- checksum. Recomputes IP and transport checksums and fixes them up.
- TBF policer. A token bucket as well as simple rate meter,

11 A lot of these attributes look very similar to what OpenFlow offers, but it should be noted that the Classifier-action subsystem existed many years before the birth of OpenFlow.

- Generic Action (*gact*). General actions that accept, drop packets or aid in the CA pipeline processing. More on gact later.
- Pedit. A generic packet editor. It uses value/mask pairs and can perform various algorithms on the packet (*xor*, *or*, *and,* etc).
- Mirred. Redirects or Mirrors packet to a port
- vlan. Encap or decap VLAN tags.
- Skbedit. Edits skb metadata
- connmark. Relates netfilter connection tracking details to skb marks.
- Etc, etc. Write a new one.

Classifier Action Programmatic Control

The CA subsystem provides a rich low level programmatic interface for composing policy to achieve a packet service.

At the core of the CA policy definition are two classes of pipeline controls. For lack of a better term, we will refer to them as *pipeline opcodes*. The first set of opcodes drives the classification block and the second set drives the action block behavior.

Each action is programmed with one or more opcodes to drive the policy flow. As an example a policer action could be programmed, when a rate is exceeded, to terminate the pipeline and indicate a packet is to be dropped or it may be programmed to take a different processing path to lower the quality of service. Essentially, the intent is programmed from the control plane and the kernel CA block implements the mechanisms.

It is important at this point to introduce the generic action, *gact,* since its sole reason for existence is to propagate the pipeline opcodes for the purpose of programmability. A gact instance is always programmed with a pipeline opcode (example in Listing 1, the *action ok* construct). When *gact* receives a packet, it accounts for it and then based on the programmed pipeline opcode helps define the flow control.

Classification Controls

Figure 6 shows some of the controls that provide the pipeline programmability that affect the packet service.

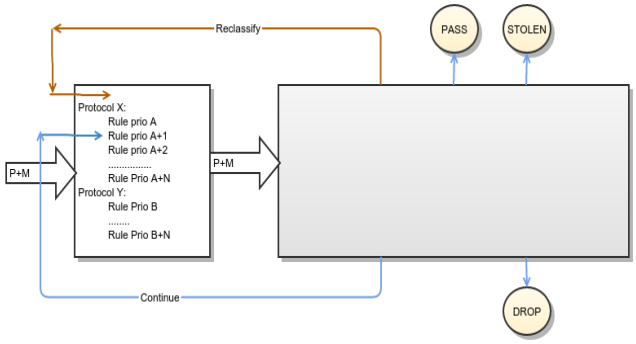

Figure 6: filter-action flow control

A packet entering the classifier block, upon matching a specific filter rule, gets exercised through the action block. Depending on the specific action's decision, the result could be:

- *Reclassify*. The packet is exercised through the whole set of anchored filter rules from the top again. This could be useful in the case of de-tunneling and the need to apply policy on the inner packet headers or tunneling and need to apply policy to the outer headers.
- *Continue*. The packet continues to be processed by the next lower priority filter rule.
- *Drop*. The pipeline is terminated and the packet is dropped.
- *Pass/OK*. The pipeline is terminated, however the packet is accepted and is allowed to proceed further to the stack on the ingress or go out a port on the egress.
- *Stolen*. The pipeline is terminated, the packet has been stolen by the action. This could mean it has been injected into a different pipeline (example redirected via mirred) or it could have been queued somewhere by the action so it could be re-injected later into some arbitrary pipeline.

Opcode: Reclassify

Lets take a look at how we could use the *reclassify* control opcode in a policy definition.

Listing 3 shows a set of rules that pop a vlan header, drop

```
#pop ingressing 802.1q vlan headers, then reclassify
tc filter add dev $ETH parent ffff: prio 1 protocol 802.1Q \
u32 match u32 0 0 \
flowid 1:1 \
action  vlan pop reclassify
#Drop any  packets from 10.0.0.21
tc filter add dev $ETH parent ffff: protocol ip prio 2 \
u32 match ip src 10.0.0.21/32 \
flowid 1:2 \
action drop
#Set the rest of the 10.x network packets to use skb queue map 3
tc filter add dev $ETH parent ffff: protocol ip prio 3 \
u32 match ip src 10.0.0.0/24 \
flowid 1:3 \
action skbedit queue_mapping 3
```

Listing 3: Reclassifying After De-tunelling

if the packet is from 10.0.0.21 otherwise if from any other host in 10.0.0.0/24 the skb metadata *queue_mapping* is set to 3 to be used further downstream.

To show the programmatic aspect, it is easier to visualize with a diagram. Listing 3 is visualized in Figure 7.

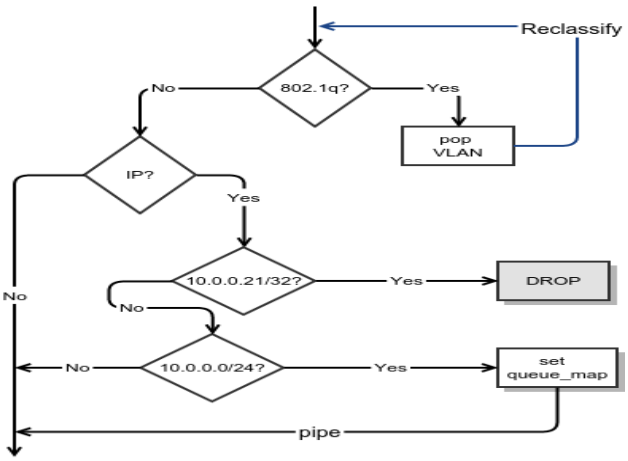

Figure 7: CA Control For Listing 3

As can be observed, we are able to build *if/else* constructs easily by taking advantage of filter rule priorities. The *reclassify* opcode gives us the opportunity to restart the policy graph. Essentially, it is a *loop* program control in that the parse operation is restarted. Also illustrated is a *drop* opcode.

Also shown in figure 7 is a *pipe* opcode control construct. While it is not shown in the policy of listing 3, the opcode is configured as the default by skbedit action[12]. Pipe is an instruction to the pipeline to continue processing the next action (in this case there was none).

Opcode: Continue

Lets take a look at how we could use the *continue* control opcode.

```
#pop ingressing IFE headers, set all skb metadata then reclassify
tc filter add dev $ETH parent ffff: prio 2 protocol 0xdead \
u32 match u32 0 0 flowid 1:1 \
action ife decode reclassify
#look further if the packet is from 192.168.100.159
tc filter add dev $ETH parent ffff: prio 4 protocol ip \
u32 match ip dst  192.168.100.159 flowid 1:2 \
action continue
#now classify based on mark
tc filter add dev $ETH parent ffff: prio 5 protocol ip \
handle 0x11 fw flowid 1:1 \
action ok
```

Listing 4: A more complex policy

Listing 4 shows a set of rules that pop an IFE header, dig deeper if the packet is from 192.168.100.159 to see if it has an skb mark of 0x11.

Figure 8 shows the visualization of Listing 4.
The important detail to observe here is that the *continue* control construct essentially gives us an *else if* extension to the existing *if/else* constructs. Also observe another exposed opcode *ok*.

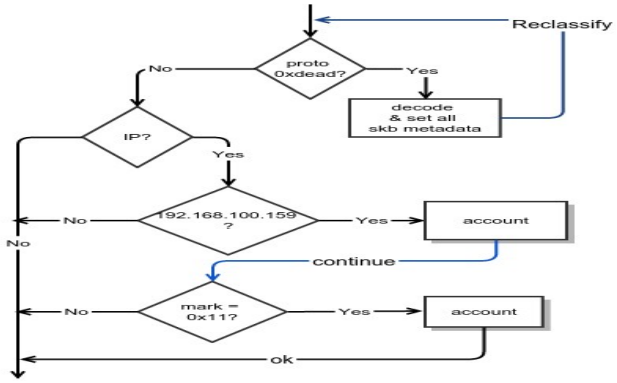

Figure 8: CA Control For Listing 4

Another way to visualize the policy is from a functional point of view. Figure 9 shows the functional view of Listing 4.

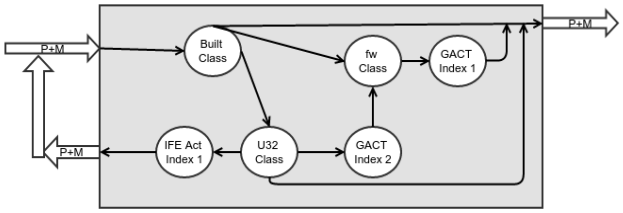

Figure 9: Functional View of Listing 4

Action Controls

Actions have opcodes that are specific within their pipeline scope. These are:

- *Pipe*. Equivalent to unix pipe construct
- *Repeat*. A loop construct
- *Jumpx*. Essentially a forward goto construct

Opcode: Pipe

Figure 10 shows a simple functional view of *pipe* control of actions.

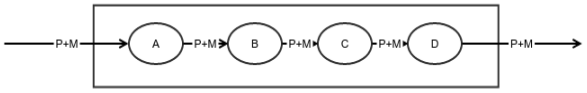

Figure 10: Action Control Policy Example

12 I looked at the source code ;->

In the simple example shown above the packet is piped from one action to the next. This is the unix equivalent of *A|B|C|D*. Figure 10's intent is achieved by programming each of the actions with the *pipe* opcode. As in unix, the action work pipeline can also be terminated by conditions programmed into actions; sometimes this is due to reaction to processing errors or other conditionals such as some threshold being exceeded.

Opcode: Repeat

Another opcode that is restricted to the action block is the REPEAT opcode. This is illustrated in Figure 11 where action C under some conditions may branch to action D or request the pipeline to re-invoke it.

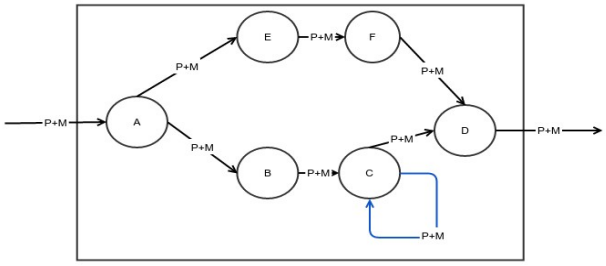

Figure 11: Action Functional Graph Showing a REPEAT Opcode

Essentially the REPEAT opcode is a *loop* construct. The scope of *REPEAT* opcode (as is the *PIPE* opcode) is only valid within the action block (as opposed to *reclassify* which has global effect).

An additional opcode, JUMPx, was envisioned back then but was never fully implemented because no strong use cases emerged[13]. The author still believes it is useful and will implement it when time allows. JUMPx was intended as a way to skip x actions in a pipeline.
As mentioned earlier the action block work-flow can be terminated by the opcodes *OK, DROP, STOLEN*.

Classifier Action Programmatic Control Summary

We have shown that it is possible to compose complex policy definitions that are made possible by the presence of CA opcodes. Program control constructs like if/else/else if/loop/(and possibly goto) are complemented with programmatic statements in the form of actions and filter rules.

Table 1 below provides the summary of all supported opcodes and how they are interpreted at either the classifier or action blocks.

OPCODE	Classifier Program Control	Action Program Control
RECLASSIFY	Start classification from the top	End Action Pipeline
CONTINUE	Iterate next rule in the same protocol	End Action Pipeline
STOLEN/QUEUED	Terminate Pipeline. Stop processing	End Action Pipeline
DROP	Drop packet. Terminate pipeline. Stop processing	End Action Pipeline
ACCEPT/OK/PASS	Terminate Pipeline. Allow packet to proceed	End Action Pipeline
PIPE	Terminate Pipeline. Allow packet to proceed	Iterate Next Action
REPEAT	Terminate Pipeline. Allow packet to proceed	Repeat Action
JUMPx	NOT YET IMPLEMENTED	NOT YET IMPLEMENTED

Table 1: Classifier-Action Opcodes

Action Instance Sharing

Action instances can be shared and bound by multiple policy graphs. This could be useful in complex policy graphs where joint accounting or sharing of resources amongst flows is needed.
We use Figure 12 to demonstrate a use case where several action instances are shared (conntrack instance 1, policer instance 1 and dropper instance 1). Figure 11 shows a setup where someone with access to the internet is sharing their access with a next door neighbor. The neighbor is connected via the wireless port, wlan0. The owner's home network is connected via the wired port, eth0. The goal is to restrict the neighbor's download rates to no more than 256Kbps and to have the shared 256Kbps contended for with the owner's downloads.
Two policy rules are provisioned – one for the home network and another for the neighbor's network (illustrated with red and blue colors).

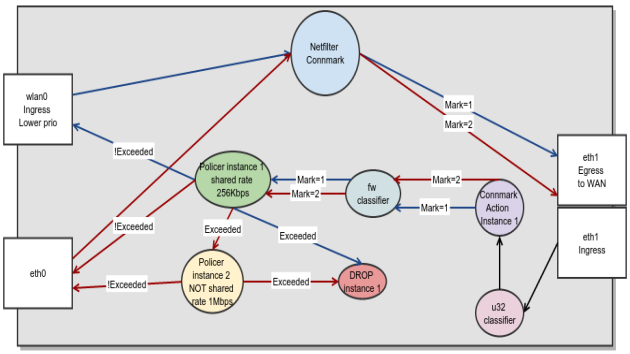

Figure 12: Neighbor Sharing

All of the neighbor's incoming packets are tagged with a connection tracking mark of 1 using nftables. The home network traffic is marked with value 2. The netfilter subsystem stores the connection state along with the mark. Download requests go out the wan port eth1. Response packets coming in (downloads etc) arrive from the internet on the ingress of eth1; they are sent to connmark action instance 1 by the u32 classifier. Connmark instance 1 consults the netfilter connection tracking state and maps all flows to the (original) connection mark. Packets are then re-classified via the fw classifier based on the restored mark. All packets with mark 1 and 2 are then subjected to a *shared* policer instance 1. If the aggregate rate exceeds

13 The lack of a use cases could be due to ignorance of the feature.

256Kbps, the policer will pass a verdict on the neighbor's packets to be dropped[14]. The home network users packets on the other hand do not suffer the same fate; when the policer instance 1 aggregate rate is exceeded, they are passed to another policer (instance 2) which further allows them the luxury of using an extra 1Mbps before they are dropped.

Flow Aging

All actions keep track of when they were installed and when they were last used. A simple way to age flows would be to add a *pipe* action to a filter. A control space application can then be used to retire idle flow entries by periodically looking at the age fields.

Late binding

It is possible to create action instances, give them identifiers and then later bind them to one or more policies.

```
tc actions add  action connmark zone 3 index 10
tc filter add dev $ETH parent ffff: pref 11 protocol ip \
u32 match ip protocol 17 0xff  flowid 1:3 action connmark index 10
```

Listing 5: Action Late Binding

Listing 5 shows a connmark action being instantiated and later bound to a flow. Note: That same action instance could have been bound to multiple flows.

Extending the Classifier-Action Across Nodes

[15] Describes the InterFE(IFE) action that is capable of

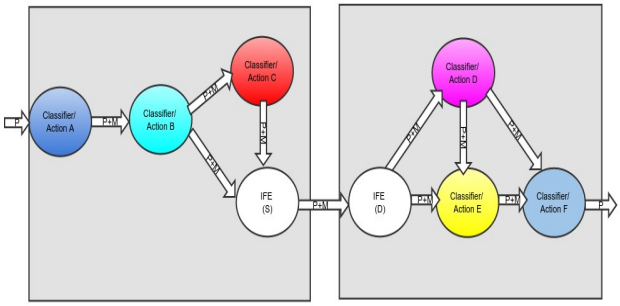

Figure 13: Distribute Policy Graph Across 2 Processing Nodes via IFE action

extending a policy graph and spreading it across multiple nodes. This is typically done to scale instantiated policy graphs or access specialized processing. One can think of IFE as a patch extending the policy graph across nodes (physical, VMs, containers, etc)

Future Work

There are several opportune activities that the community could undertake to get us to the next level.

Improving Usability

One of the main challenges of the TC architecture (i.e. not limited to the CA subsystem) in general has been usability. Some of the constructs of the *tc* utility BNF grammar, although precise and complete, are very low level and could be overwhelming to the lay person. Werner Almesberger tried to improve usability with tcng[11] but lost interest in the project at some point. That work could be taken over by anyone interested.

TC and the CA subsystem in general lend themselves well to a programming language. As we have shown a lot of the necessary programmatic controls (loops, branches, statement executions) are already in place. It is possible to extend the tc BNF grammar into a higher level language binding. The author is very interested in this work and welcomes a discussion in this area.

Functional Discovery

Earlier versions of the CA subsystem(up to until a year ago) had some rudimentary capability discovery built in. No user space code was added to take advantage of the feature, so a decade later it sounded fair to remove it. Over the years the author feels he has learnt lessons in particular in the involvement with ForCES[22] and working with variety of hardware with equivalent functions but differing capabilities that he feels it is time to for a fresh perspective; however, that level of discussion is out of topic for this paper.

Hardware Offloading

Given the architecture of the CA subsystem (the control constructs and functional statement expression), it would be very fitting to implement the whole architecture in hardware. Infact on reviewing a lot of hardware layout one would observe there is already a lot of synergy with the TC CA architectural definitions.

Figure 13 shows a Realtek RTL8366xx datapath and mapping to TC.

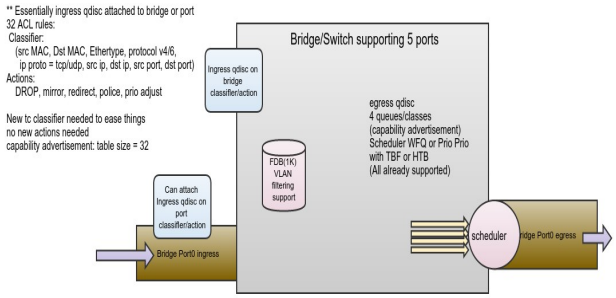

Figure 13: Realtek RTL8366xx Datapath

While the RTL8366xx is a tiny chip, a lot of other ASICs have similar architectural layout but different capabilities and capacities. There are challenges that need to be addressed with handling the capacitance mismatch between what hardware offers and how policy maps it. We have a

14 For TCP this helps slow down the neighbor's downloads.

lot of community experience in dealing with quarks (from hardware bugs) and feature differences that can be leveraged in getting this part right. As can be observed from figure 13, there is a very close resemblance between the hardware datapath and the Linux datapath.

The author believes there is strong need for a tight coupling between the Linux control interfaces, which are already widely deployed, and hardware offload. The requirement is necessary to provide transparency of the tooling. There is a huge ecosystem around linux tooling in place and while it is possible to have new tools, it should only be done if the current toolsets cannot be re-factored in order to provide smooth continuity. In other words the hardware ASICs provide drivers that are exercised transparently by the current Linux tools and APIs.

Acknowledgements

The preparation of these instructions and the LaTeX and LibreOffice files was facilitated by borrowing from similar documents used for ISEA2015 proceedings.
The author thanks Werner Almesberger, Alexey Kuznetsov and David Miller for their patience and many words of wisdom when he was working on tc action.

References

1. Netfilter home, http://www.netfilter.org/
2. Netfilter's Connection Tracking System, Pablo Neira Ayuso, *people.netfilter.org/pablo/docs/login.pdf*
3. Pablo Neira Ayuso,Rafael M. Gasca, Laurent Lefevre. Communicating between the kernel and user-space Linux using Netlink sockets. Software: Practice and Experience, 2010
4. J. Hadi Salim, H. Khosravi, A. Kleen, A. Kuznetsov, Linux Netlink as an IP Services Protocol, RFC 3549, July 2003
5. Paul E. McKenney "Stochastic Fairness Queuing", IEEE INFOCOMM'90 Proceedings, San Francisco, 1990.
6. Kathleen Nichols, Van Jacobson. "Controlling Queue Delay" ACM Queue. ACM Publishing. 6 May 2012
7. Sally Floyd, Van Jacobson; "Random Early Detection (RED) gateways for Congestion Avoidance". IEEE/ACM Transactions on Networking (August 1993)
8. Werner Almesberger, “Linux Network Traffic Control – Implementation Overview”, April, 1999
9. Werner Almesberger, "Linux Traffic Control - Next Generation", 18 October 2002
10. Werner Almesberger, Jamal Hadi Salim, Alexey Kuznetsov “Differentiated Services on Linux”, Proceedings of Globecom '99, vol. 1, pp. 831--836, December 1999
11. Bob Braden (editor), “Resource ReSerVation Protocol (RSVP) version 1”, RFC 2205
12. Jiří Pírko, “Implementing Open vSwitch datapath using TC”, Proceedings of Netdev 0.1, Feb 2015
13. Jamal Hadi Salim, Damascene M. Joachimpillai, “Distributing TC Classifier-Action Packet Processing”, Proceedings of Netdev 0.1, Feb 2015
14. Stephen Hemminger, “Network Emulation with NetEm”, Linux Conf Australia, 2005
15. Steven McCanne, Van Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture", Dec 1992
16. Robert S. Boyer, J Strother Moore, "A Fast String Searching Algorithm.". *Comm.* Association for Computing Machinery, 1977
17. Donald Knuth, James H. Morris, James , Vaughan Pratt, "Fast pattern matching in strings". *SIAM Journal on Computing, 1977*
18. OpenFlow Spec, https://www.opennetworking.org/
19. P4 Spec, http://p4.org/
20. ForCES Spec, https://datatracker.ietf.org/wg/forces/charter/
21. Daniel Borkman, “BPF classifier”, net/sched/cls_bpf.c

Author Biography

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.