Leveraging Kernel Tables with XDP
Is XDP Really Different than DPDK?
BPF programs are loaded, maps created
- Limited BPF helpers for XDP
- Maps provide data

Programs attached to device
- XDP hook in the driver runs programs for each packet
- Fast path packet processing

ACLs, firewall, load balancer, fast-path forwarding, …
Focus Discussion on
L2 / L3 Forwarding in XDP
Forwarding via Management by SDN Controller

Agent

- Receives updates from controller
- Updates maps

Little / no updates based on local events

All packet parsing, processing, rewriting done in BPF

- Packet never sees the Linux stack

Linux is a boot OS
Is XDP with Maps Really Different than DPDK?

Both bypass Linux networking stack and its networking APIs

Custom implementations of standard protocols
  ▪ e.g., Bridging and aging fdb entries

Custom tools to manage solution
  ▪ Configuration, monitoring and debugging

...  

Need better integration of XDP with Linux
  ▪ More complete and consistent Linux solution
More Linux: Snooping Notifications

Agent
- snoops RTM_*ROUTE and RTM_*NEIGH entries
- Updates lpm and neigh maps based on rtnetlink notifications

For simple setups only
- VLANs and bonds?
- Scalability with number of routes & nexthops? Duplicate data
- Multipath or encaps (e.g., MPLS)?
- MTU and fragmentation, changes in link state, …?

Full solution duplicates Linux!
Linux and Hardware Offload

Standard Linux APIs configure networking

Kernel modified so that ASIC driver gets necessary information

ASIC driver handles notifications and programs hardware
XDP as a Software “Offload”

Consider XDP as an offload rather than new framework

- Do not want to reinvent implementations of networking features

Allow BPF programs to access kernel tables

Consistency all the way around

- Existing protocols, APIs and processes to configure networking
- Slow path and fast path driven by same data
Options for Packet Pipeline in XDP

Don’t want to replicate Linux protocols and features in BPF form

Enable building blocks that BPF programs can invoke as desired

Networking management still done with standard tools
Key Elements for Packet Forwarding in XDP

Support for essential networking features
- vlans, bonds, bridges, macvlan

ACL and filters - ingress and egress
- Several attach points

Forwarding lookup
- FDB Lookup - L2 forwarding
- FIB Lookup - L3 forwarding

Packet Scheduling
- Crossing bandwidths, traffic shaping and priority
Linux Packet Processing (simplified)

per packet loop in __netif_receive_skb_core

Driver Rx allocate skb
  -> Handle vlan tag
     Set vlan_tci, Remove vlan header, Update skb protocol
  -> tc ingress
  -> netfilter ingress
     Ingress ACLs or filters
     vlan on skb dev?
        yes
        Get vlan dev
        Update skb dev
        yes
        Update skb dev to upper dev
        no
        upper dev? recirc?
           yes
           LAG, bridge, macvlan, …

Each pass handles features: vlans, bonds, bridges

Pass 1: skb->dev = net_dev for port
Pass 2: skb->dev could be net_dev for {port.vlan}
Pass 3: skb->dev could be upper dev (e.g., bond)
Pass 4: skb->dev could be net_dev for {upper_dev.vlan}
Linux Packet Processing (simplified)

- netfilter hooks
- forwarding lookup
- drop?
  - yes
    - drop pkt
  - no
    - forward?
      - yes
        - local delivery
      - no
        - egress netfilter hooks

- header rewrite
- packet queueing
- Tx

includes ACL via FIB rules
Device “Table” and Feature Stacking on NIC Port

Base is NIC Port
- Number of combinations of features stacked on top of port

<table>
<thead>
<tr>
<th>VLANs</th>
<th>LAG</th>
<th>VLAN</th>
<th>Bridge</th>
<th>VLANs</th>
<th>Bridge</th>
<th>LAG</th>
<th>VLAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Interface</td>
<td>Network Interface</td>
<td>Network Interface</td>
<td>Network Interface</td>
<td>Network Interface</td>
<td>Network Interface</td>
<td>Network Interface</td>
<td>Network Interface</td>
</tr>
</tbody>
</table>
Example: Port with multiple VLANs

NIC port as a trunk for multiple VLANs

VLAN subinterfaces exist and match destination mac

- “Router interfaces,” L3 Forwarding, potentially based on VRF
- Look up needs the subinterface index

VLANs directed at bridge?

- L2 forwarding … maybe
- VLAN aware bridge means no explicit subinterface devices
- Could be a VLAN on the bridge (SVI)!
Example: Port with Bonding

NIC ports are part of a LAG (bond)

Bond can:
- have a network address (L3 forwarding)
- be enslaved to a bridge (L2 forwarding)
- Bond can have multiple VLANS – previous slide

Egress port selection
- Forwarding lookup shows bond as the egress. Which port to use to Tx packet?
- Logic within the bond to select egress
Networking features typically implemented in modules and instantiated as virtual net_devices

- VLANs, LAG, bridging, VRF
- Manages upper/lower relationships
- Could be manipulating the packet (adding/removing vlan header) or dropping it (ingress on inactive slave)
- Could be implementing a protocol (bond and 802.3ad; bridge and stp)
- Selecting an egress device (bond, team)
- Influencing a lookup (bridge, vrf) or learning (bridge)

Do not want to have to reimplement networking features for XDP

Allow standard Linux interface managers to configure/manage networking features

- VLANs, bonds, macvlans, vlan-aware bridge, SVIs
Access to Device Table

Need bpf helpers to convert \{port index, vlan, dmac\} to:

- LAG id
- L2 device index – device enslaved to a bridge; L2 forwarding lookup
- L3 / RIF device index – input to L3 forwarding lookup
- Other intermediary devices? Features as a device is a Linux’ism

Device (or index) is key to other features

- ACL, filtering, scheduling, forwarding

Egress port index

- L2/L3 forwarding returns upper device
- Need egress port index for redirect
Prototype

One option is to use existing upper/lower list_heads

- Reduces code refactoring and APIs
- Does not scale (e.g., number of VLANs)

Better solution is to mimic logic of __netif_receive_skb_core

- Heavy refactoring, exporting APIs or creating new ones to move from one device type (feature) to another
  - Ingress port
    - If vlan tag, search for vlan device on port
    - Is net_device a LAG, bridge or macvlan slave?

Tested with VLANs and bonds – ingress and egress
Challenges

Core kernel code vs modules
- BPF helpers in filter.c – compiled into kernel
- Driver functionality often loaded as modules
- Common problem: vlan, bond, bridge, mpls, …

Code refactoring needed to not duplicate code
- Functions need to be invoked from 2 different contexts – XDP and skb

Number of use cases to cover to converge device lookup API
ACL, Filters and Packet Scheduling

ACL
- FIB rules supported via FIB lookup helper – L3 ACL
- tc vs netfilter

Packet Scheduling
- tc and qdiscs

Start point
- Converting tc code to handle frames from XDP context is the bigger ROI
L2 / L3 Forwarding

FIB lookup helper exists for L3 Forwarding
- Needs more work – e.g., handle lwtunnel encaps
- MPLS support (started)

Bridges and L2 forwarding
- FDB access per bridge
- MAC learning
- Flooding
Summary

Need better integration of XDP with Linux
- More complete and consistent Linux solution

Do not need BPF programs reinventing / re-implementing Linux
- Established APIs for configuring, monitoring, troubleshooting
- Established implementations of protocols with notifications and cleanups (e.g., aging entries)

Enables Linux to be a slow-path assist
- Same data for full-stack and fast-path in XDP
Thank you!

Visit us at cumulusnetworks.com or follow us @cumulusnetworks