VPP
The Universal Fast Dataplane
FD.io: The Universal Fast Dataplane

- **Project at Linux Foundation**
  - Multi-party
  - Multi-project

- **Software Dataplane**
  - High throughput
  - Low Latency
  - Feature Rich
  - Resource Efficient
  - Bare Metal/VM/Container
  - Multiplatform

- **FD.io Scope**
  - **Network IO** – NIC/vNIC <-> cores/threads
  - **Packet Processing** – Classify / Transform / Prioritize / Forward / Terminate
  - **Dataplane Management Agents** – Control Plane

![Bare Metal/VM/Container Diagram](image)
Fd.io in the overall stack
Multiparty: Broad Membership

Service Providers
- AT&T
- Comcast

Network Vendors
- Cisco
- Ericsson
- Huawei
- Metaswitch
- Brocade

Chip Vendors
- Intel
- Cavium Networks

Integrators
- Red Hat
- Inocybe
Multiparty: Broad Contribution
Code Activity

- In the period since its inception, fd.io has more commits than OVS and DPDK combined, and more contributors than OVS.

<table>
<thead>
<tr>
<th></th>
<th>2016-02-11 to 2017-04-03</th>
<th>Fd.io</th>
<th>OVS</th>
<th>DPDK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commits</td>
<td></td>
<td>6283</td>
<td>2395</td>
<td>3289</td>
</tr>
<tr>
<td>Contributors</td>
<td></td>
<td>163</td>
<td>146</td>
<td>245</td>
</tr>
<tr>
<td>Organizations</td>
<td></td>
<td>42</td>
<td>52</td>
<td>78</td>
</tr>
</tbody>
</table>

![Commit, Contributor, Organization Bar Charts](chart.png)
Multiproject: Fd.io Projects

Dataplane Management Agent
- Honeycomb
- hc2vpp

Packet Processing
- NSH_SFC
- ONE
- TLDK
- CICN
- odp4vpp
- VPP Sandbox
- VPP

Network IO
- deb_dpdk
- rpm_dpdk

Testing/Support
- CSIT
- puppet-fdio
- trex
Fd.io Integrations

Integration work done at OpenStack

- Openstack
  - Neutron
    - ODL Plugin
    - Fd.io Plugin

Control Plane

- Lispflowmapping app
- SFC
- VBD app

LISP Mapping Protocol

Data Plane

- Honeycomb
- Fd.io ML2 Agent

Netconf/Yang

REST

fd.io Foundation
Vector Packet Processor - VPP

- Packet Processing Platform
  - High performance
  - Linux User space
  - Run’s on commodity CPUs: Intel/ARM
  - Shipping at volume in server & embedded products since 2004.
VPP Architecture: Vector Packet Processing

Packet Processing Graph

Packet

Input Graph Node

Graph Node

Vector of n packets

dpdk-input → vhost-user-input → ... → af-packet-input

ethernet-input

ip6-input → ip4-input → mpls-input → ... → arp-input

ip6-lookup → ip4-lookup

ip6-rewrite → ip6-local → ip4-local → ip4-rewrite
VPP Architecture: Splitting the Vector

Vector of n packets

Packet Processing Graph

Packet

Input Graph Node

Graph Node

dpdk-input → vhost-user-input → ethernet-input → ip6-input → ip6-local → ip6-rewrite

dpdk-input → vhost-user-input → ethernet-input → ip4-input → ip4-local → ip4-rewrite

dpdk-input → vhost-user-input → ethernet-input → mpls-input

dpdk-input → vhost-user-input → ethernet-input → arp-input

af-packet-input
VPP Architecture: Plugins

Plugins are:
First class citizens
That can:
Add graph nodes
Add API
Rearrange the graph

Can be built independently of VPP source tree

Hardware Plugin

Skip software nodes where work is done by hardware already
Let’s look at performance data at scale
Packet throughput for
  - IPv4 routing,
  - IPv6 routing,
  - L2 switching,
  - L2 switching with VXLAN tunnelling.
VPP Universal Fast Dataplane: Performance at Scale [1/2]

Per CPU core throughput with linear multi-thread(-core) scaling

IPv4 Routing

IPv6 Routing

Hardware:
Cisco UCS C240 M4
Intel® C610 series chipset
2 x Intel® Xeon® Processor E5-2698 v3 (16 cores, 2.3GHz, 40MB Cache)
2133 MHz, 256 GB Total
6 x 2p40GE Intel XL710=12x40GE

Software
Linux: Ubuntu 16.04.1 LTS
Kernel: ver. 4.4.0-45-generic
FD.io VPP: VPP v17.01-5~ge234726 (DPDK 16.11)

Resources
1 physical CPU core per 40GE port
Other CPU cores available for other services and other work
20 physical CPU cores available in 12x40GE setup
Lots of Headroom for much more throughput and features
VPP Universal Fast Dataplane: Performance at Scale [2/2]
Per CPU core throughput with linear multi-thread(-core) scaling

L2 Switching

L2 Switching with VXLAN Tunneling

Performance at Scale [2/2]

Hardware:
Cisco UCS C240 M4
Intel® C610 series chipset
2 x Intel® Xeon® Processor E5-2698 v3 (16 cores, 2.3GHz, 40MB Cache)
2133 MHz, 256 GB Total
6 x 2p40GE Intel XL710=12x40GE

Software
Linux: Ubuntu 16.04.1 LTS
Kernel: ver. 4.4.0-45-generic
FD.io VPP: VPP v17.01-5~ge234726 (DPDK 16.11)

Resources
1 physical CPU core per 40GE port
Other CPU cores available for other services and other work
20 physical CPU cores available in 12x40GE setup
Lots of headroom for much more throughput and features

Lots of Headroom for much more
12x40GE setup
20 physical CPU cores available in
12x40GE setup
Lots of headroom for much more throughput and features

MACThput [Mpps] 2x 40GE 4 core 6x 40GE 8 core
6x 40GE 10 core 12x 40GE 12 core
64B 20.0 38.4 55.9 73.5 91.0 108.6
128B 20.8 39.4 55.9 73.5 91.0 108.6
IMIX 15.0 30.0 45.0 60.0 75.0 90.0
1518B 3.8 7.6 11.4 15.2 19.0 22.8

MACThput [Mpps] 2x 40GE 4 core 6x 40GE 8 core
8x 40GE 10 core 12x 40GE 12 core
64B 11.6 25.1 38.5 52.2 65.7 79.2
128B 11.6 25.1 38.5 52.2 65.7 79.2
IMIX 10.5 21.0 31.5 42.0 52.5 63.0
1518B 3.8 7.6 11.4 15.2 19.0 22.8

MACThput [Mpps] 2x 40GE 4 core 6x 40GE 8 core
8x 40GE 10 core 12x 40GE 12 core
64B 35.8 71.6 107.4 143.2 179 214.8
128B 35.8 71.6 107.4 143.2 179 214.8
IMIX 10.5 21.0 31.5 42.0 52.5 63.0
1518B 3.8 7.6 11.4 15.2 19.0 22.8

MACThput [Mpps] 2x 40GE 4 core 6x 40GE 8 core
8x 40GE 10 core 12x 40GE 12 core
64B 46.8 93.5 140.3 187.0 233.8 280.5
128B 46.8 93.5 140.3 187.0 233.8 280.5
IMIX 10.5 21.0 31.5 42.0 52.5 63.0
1518B 3.8 7.6 11.4 15.2 19.0 22.8

Topologies:
Phy-VS-Phy
Scaling Up The Packet Throughput with FD.io VPP
Can we squeeze more from a single 2RU server?

1. Today's Intel® XEON® CPUs (E5 v3/v4):
   a. Per socket have 40 lanes of PCIe Gen3
   b. 2x 160Gbps of packet I/O per socket

2. Tomorrow's Intel® XEON® CPUs:
   a. Per socket support More lanes of PCIe Gen3
   b. 2x 280Gbps of packet I/O per socket

VPP enables linear multi-thread(-core) scaling up to the packet I/O limit per CPU => on a path to one terabit software router (1TFR).

Breaking the Barrier of Software Defined Network Services
1 Terabit Services on a Single Intel® Xeon® Server !!!
VPP Architecture: Programmability

Architecture

- Linux Hosts
  - Shared Memory
    - Request Queue
    - Response Queue

- Agent
- VPP

Request Message 900k request/s

Async Response Message

Example: Honeycomb

- Linux Hosts
  - Shared Memory
    - Request Queue
    - Response Queue

- Honeycomb Agent
- VPP

Netconf/Restconf/Yang

Request Message

Async Response Message

Can use C/Java/Python/or Lua Language bindings
Universal Fast Dataplane: Features

Hardware Platforms
- Pure Userspace - X86, ARM 32/64, Power, Raspberry Pi

Interfaces
- DPDK/Netmap/Af_Packet/TunTap
- Vhost-user - multi-queue, reconnect, Jumbo Frame Support

Language Bindings
- C/Java/Python/Lua

Tunnels/Encaps
- GRE/VXLAN/VXLAN-GPE/LISP-GPE
- MPLS over Ethernet/GRE
- Deep label stacks supported

Routing
- IPv4/IPv6
- 14+ MPPS, single core
- Hierarchical FIBs
- Multimillion FIB entries
- Source RPF
- Thousands of VRFs
- Controlled cross-VRF lookups
- Multipath - ECMP and Unequal Cost
- IR Multicast

Segment Routing
- SR MPLS/IPv6
- Including Multicast

LISP
- LISP xTR/RTR
- L2 Overlays over LISP and GRE encaps
- Multitenancy
- Multihome
- Map/Resolver Failover
- Source/Dest control plane support
- Map-Register/Map-Notify/RLOC-probing
- IPSEC transport mode

MPLS
- MPLS over Ethernet/GRE
- Deep label stacks supported

Switching
- VLAN Support
  - Single/ Double tag
  - L2 forwrd w/EFP/BridgeDomain concepts
- VTR - push/poptranslate (1:1,1:2,2:1,2:2)
- Mac Learning - default limit of 50k addr
- Bridging
- Split-horizon group support/EFP Filtering
- Proxy Arp
- Arp termination
- IRB - BVI Support with RouterMac assignment
- Interface cross-connect
- L2 GRE over IPSEC tunnels

Security
- Mandatory Input Checks:
  - TTL expiration
  - header checksum
  - L2 length < IP length
  - ARP resolution/snooping
  - ARP proxy
- NAT
- Ingress Port Range Filtering
- Per interface whitelists
- Policy/Security Groups/GBP (Classifier)

Network Services
- DHCPv4 client/proxy
- DHCPv6 Proxy
- MAP/LW46 - IPv4aas
- CGNAT
- MagLev-like Load Balancer
- Identifier Locator Addressing
- NSH SFC SFF's & NSH Proxy
- LLDP
- BFD
- QoS Policer 1R2C, 2R3C
- Multiple million Classifiers - Arbitrary N-tuple

Inband iOAM
- Telemetry export infra (raw IPFIX)
- iOAM for VXLAN-GPE (NGENA)
- SRv6 and iOAM co-existence
- iOAM proxy mode / caching
- iOAM probe and responder

Monitoring
- Simple Port Analyzer (SPAN)
- IP Flow Export (IPFIX)
- Counters for everything
- Lawful Intercept
Rapid Release Cadence – ~3 months

16-02
Fd.io launch

16-06
Release: VPP

16-06 New Features
Enhanced Switching & Routing
IPv6 SR multicast support
LISP xTR support
VXLAN over IPv6 underlay
per interface whitelists
shared adjacencies in FIB
Improves interface support
vhost-user - jumbo frames
Netmap interface support
AF_Packet interface support
Improved programmability
Python API bindings
Enhanced JVPP Java API bindings
Enhanced debugging cli
Hardware and Software Support
Support for ARM 32 targets
Support for Raspberry Pi
Support for DPDK 16.04

16-09
Release: VPP, Honeycomb,
NSH_SFC, ONE

16-09 New Features
Enhanced LISP support for
L2 overlays
Multitenancy
Multihoming
Re-encapsulating Tunnel Routers
(RTR) support
Map-Resolver failover algorithm
New plugins for
SNAT
MagLev-like Load
Identifier Locator Addressing
NSH SFC SFF’s & NSH Proxy
Port range ingress filtering
Dynamically ordered subgraphs

17-01
Release: VPP, Honeycomb,
NSH_SFC, ONE

17-01 New Features
Hierarchical FIB
Performance Improvements
DPDK input and output nodes
L2 Path
IPv4 lookup node
IPSEC
Softwad HWCrypto Support
HQoS support
Simple Port Analyzer (SPAN)
BFD
IPFIX Improvements
L2 GRE over IPsec tunnels
LLDP
LISP Enhancements
Source/Dest control plane
L2 over LISP and GRE
Map-Register/Map-Notify
RLOC-probing
ACL
Flow Per Packet
SNAT - Multithread, Flow Export
LUA API Bindings
New in 17.04 – Released Apr 19

- **VPP Userspace Host Stack**
  - TCP stack
  - DHCPv4 relay multi-destination
  - DHCPv4 option 82
  - DHCPv6 relay multi-destination
  - DHCPv6 relay remote-id
  - ND Proxy

- **Stateful Security Groups**
  - Routed interface support
  - L4 filters with IPv6 Extension Headers

- **NAT**
  - CGN: Configurable port allocation
  - CGN: Configurable Address pooling
  - CPE: External interface
  - DHCP support
  - NAT44, NAT64, LW46

- **Segment Routing v6**
  - SR policies with weighted SID lists
  - Binding SID
  - SR steering policies
  - SR LocalSIDs
  - Framework to expand local SIDs w/plugins

- **API**
  - Move to CFFI for Python binding
  - Python Packaging improvements
  - CLI over API
  - Improved C/C++ language binding

- **Segment Routing v6**
  - SR policies with weighted SID lists
  - Binding SID
  - SR steering policies
  - SR LocalSIDs
  - Framework to expand local SIDs w/plugins

- **iOAM**
  - UDP Pinger w/path fault isolation
  - IOAM as type 2 metadata in NSH
  - IOAM raw IPFIX collector and analyzer
  - Anycast active server selection

- **IPFIX**
  - Collect IPv6 information
  - Per flow state

Release notes: [https://docs.fd.io/vpp/17.04/release_notes_1704.html](https://docs.fd.io/vpp/17.04/release_notes_1704.html)
Images at: [https://nexus.fd.io/](https://nexus.fd.io/)
Universal Fast Dataplane: Infrastructure

- **Bare Metal**
  - Server
  - FD.io
  - Kernel/Hypervisor

- **Cloud/NFVi**
  - Server
  - VM
  - VM
  - VM
  - FD.io
  - Kernel/Hypervisor

- **Container Infra**
  - Server
  - Con
  - Con
  - Con
  - FD.io
  - Kernel
Universal Fast Dataplane: xNFs

FD.io based vNFs

Server

VM
FD.io

VM
FD.io

FD.io

Kernel/Hypervisor

FD.io based cNFs

Server

Con
FD.io

Con
FD.io

FD.io

Kernel/Hypervisor
Universal Fast Dataplane: Embedded

Embedded Device
- Device
  -FD.io
  -Kernel/Hypervisor
  -Hw Accel

SmartNic
- Server
  -Kernel/Hypervisor
  -SmartNic
  -FD.io
  -Hw Accel
Universal Fast Dataplane: CPE Example

- **Physical CPE**
  - Device
  - FD.io
  - Kernel/Hypervisor
  - Hw Accel

- **vCPE in a VM**
  - VM
  - FD.io
  - FD.io
  - VM
  - FD.io
  - FD.io
  - Kernel/Hypervisor

- **vCPE in a Container**
  - Con
  - FD.io
  - FD.io
  - Con
  - FD.io
  - Kernel/Hypervisor
Opportunities to Contribute

We invite you to Participate in fd.io

• Get the Code, Build the Code, Run the Code
• Try the vpp user demo
• Install vpp from binary packages (yum/apt)
• Install Honeycomb from binary packages
• Read/Watch the Tutorials
• Join the Mailing Lists
• Join the IRC Channels
• Explore the wiki
• Join fd.io as a member

FD.io git repos:
https://git.fd.io/
https://git.fd.io/vpp/
https://git.fd.io/csit/

FD.io project wiki pages:
https://wiki.fd.io/view/Main_Page
https://wiki.fd.io/view/VPP
https://wiki.fd.io/view/CSIT
Aside [1/4]: Computer Evolution For Packet Processing

*It started with ALU.* …

(1) It started simple …

(2) Became universal and faster …

(3) Then it got much faster …

(4) … but far from simple!

An Arithmetic Logic Unit (ALU)

(1) It started simple...

A simple generic purpose computer

(2) Became universal and faster...

A modern two CPU socket server

(3) Then it got much faster...

(4) ... but far from simple!

Aside [2/4]: Computer Evolution For Packet Processing...
... and we arrived to modern multi-socket COTS server ...

An Arithmetic Logic Unit (ALU)

(1) It started simple…


(2) Became universal and faster…

(3) Then it got much faster…

A modern two CPU socket server

(4) … but far from simple!

The pipeline of a modern high-performance XEON CPU

Four main functional dimensions important for processing packets:

A. CPUs executing the program(s):
   a) Minimize Instructions per Packet - Efficient software logic to perform needed packet operations.
   b) Maximize Instructions per CPU core clock cycle - Execution efficiency of an underlying CPU micro-architecture.

B. Memory bandwidth: Minimize memory bandwidth utilization - Memory access is slow.

C. Network I/O bandwidth: Make efficient use of PCIe I/O bandwidth - It is a limited resource.

D. Inter-socket transactions: Minimize cross-NUMA connection utilization - It slows things down.

Hint: Start with optimizing the use of CPU micro-architecture => Use vectors!

Aside [2/4]: Computer Evolution For Packet Processing

… and we arrived to modern multi-socket COTS server …
Aside [3/4]: Computer Evolution For Packet Processing

… we then optimize software for network workloads ...

(2) Became universal and faster ...

A simple generic purpose computer

(1) It started simple ...

An Arithmetic Logic Unit (ALU)

(3) Then it got much faster ...

A modern two CPU socket server

(4) ... but far from simple!

The pipeline of a modern high-performance XEON CPU

Aside [3/4]: Computer Evolution For Packet Processing

… we then optimize software for network workloads ...

• Network workloads are very different from compute ones
  • They are all about processing packets, at rate.
  • At 10GE, 64B packets can arrive at 14.88Mpps => 67 nsec per packet.
  • With 2GHz CPU core clock cycle is 0.5nsec => 134 clock cycles per packet.
  • To access memory it takes ~70nsec => too slow to do it per packet!

• Packet processing efficiency is essential
  • Moving packets
    • Packets arrive on physical interfaces (NICs) and virtual interfaces (VNFs) - need CPU optimized drivers for both.
    • Drivers and buffer management software must not rely on memory access – see time budget above, MUST use CPU core caching hierarchy well.

  • Processing packets
    • Need packet processing optimized for CPU platforms.
    • Header manipulation, encap/decaps, lookups, classifiers, counters.

Aside [3/4]: Computer Evolution For Packet Processing
... we then optimize software for network workloads ...

(1) It started simple ...

- Network workloads are very different from compute ones
  - They are all about processing packets, at rate.
  - At 10GE, 64B packets can arrive at 14.88Mpps => 67 nsec per packet.
  - With 2GHz CPU core clock cycle is 0.5nsec => 134 clock cycles per packet.
  - To access memory it takes ~70nsec => too slow to do it per packet !

(2) Became universal and faster ...

- Packet processing efficiency is essential
  - Moving packets
    - Packets arrive on physical interfaces (NICs) and virtual interfaces (VNFs) - need CPU optimized drivers for both.
    - Drivers and buffer management software must not rely on memory access – see time budget above, MUST use CPU core caching hierarchy well.
  - Processing packets
    - Need packet processing optimized for CPU platforms.
    - Header manipulation, encap/decaps, lookups, classifiers, counters.

(3) Then it got much faster ...

- A modern two CPU socket server

(4a) ... let’s make it Simple

AND => (4a) ... let’s make it Simple

Aside [3/4]: Computer Evolution For Packet Processing
... we then optimize software for network workloads ...
An Arithmetic Logic Unit (ALU)

(1) It started simple
(2) Became universal and faster...
(3) Then it got much faster...
(4a) ...let’s make it Simple AGAIN!

Aside [4/4]: Computer Evolution For Packet Processing
...and use FD.io VPP to make them fast for packets.

Vector Packet Processing
software worker thread

dpdk-input  vhost-user-input  ...  af-packet-input
ethernet-input
ip6-input  ip4-input  mpls-input  ...  arp-input
ip6-lookup  ip4-lookup
ip6-rewrite  ip6-local  ip4-local  ip4-rewrite

(1) Core writes Rx descriptor in preparation for receiving a packet.
(2) NIC reads Rx descriptor to get ctrl flags and buffer address.
(3) NIC writes the packet.
(4) NIC writes Rx descriptor.
(5) Core reads Rx descriptor (polling or irq or coalesced irq).
(6) Core reads packet header to determine action.
(7) Core performs action on packet header.
(8) Core writes packet header (MAC swap, TTL, tunnel, foobar..)
(9) Core reads Tx descriptor.
(10) Core writes Tx descriptor and writes Tx tail pointer.
(11) NIC reads Tx descriptor.
(12) NIC reads the packet.
(13) NIC writes Tx descriptor.

AND => (4a) ... let’s make it Simple

A modern two CPU socket server

A modern two CPU socket server

CPU Socket Again!

CPU Cores

NIC packet operations
NIC descriptor operations

Asynchronous Logic Unit (ALU)
Steps 1-to-13 in a Nutshell:

- VPP software worker threads run on CPU cores.
- Use local caching with No-to-Minimal memory bandwidth per packet.
- Get speed with predictive prefetching and smart algos.
- And make CPU cache hierarchy always “Hot” => Packet processing at rate.

Making VPP simply tick the A-B-C-D server optimization points!