VECTOR PACKET PROCESSING SIMPLIFIED

VPP Simplified OVERVIEW

For decades, the only way to speed up packet processing - central to all security and networking functions - was to add faster and faster hardware. When Moore's law couldn't address that need sufficiently on commodity silicon, the industry resorted to expensive, painfully slow time-to-market, inflexible, specialty silicon. But there is a new game in town: Vector Packet Processing (VPP).

Powered by the FD.io open source project, VPP delivers up to two orders of magnitude greater packet processing throughput, via software running on commodity processors. It is a core component in a much larger transformation of how information - the lifeblood of the digital age - will be moved and secured for years to come.

But what exactly is VPP? How does it perform this feat? How can it benefit organizations like yours? This primer addresses four key takeaways:

1. It makes packet processing really fast
2. It makes packet processing easily programmable and can be integrated into existing or new secure networking ecosystems
3. It can be deployed virtually anywhere
4. It is a project, so unless you are a DIY-type, you'll want to look for a product

VPP Simplified HIGHLIGHTS

KERNEL VS. VECTOR PACKET PROCESSING

Unlock two orders of magnitude speed gain from bypassing the kernel with vector packet processing.

PROGRAMMABILITY AND EXTENSIBILITY

Quickly and easily add, move, and change packet processing policies.

DEPLOYMENT

Agnostic to deployment need - appliance, virtual machine, virtual network function (VNF). Access an array of processor types.

PRODUCTIZATION

Projects are not products. TNSR is the answer to Vector Packet Processing productization.

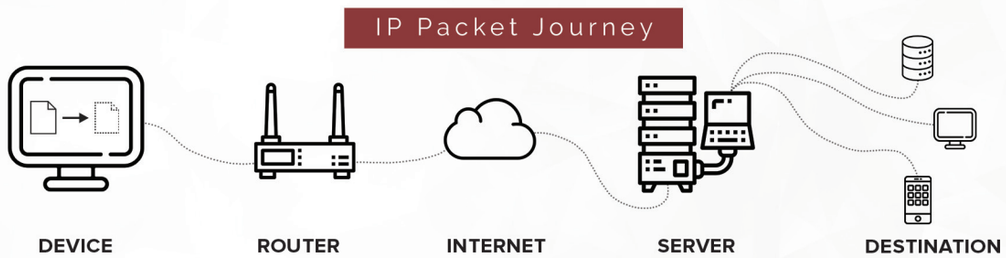
ESCAPE THE KERNEL.
FREE YOUR NETWORK.

tnsr™

PACKETING PROCESSING
EXPLAINED

First, a few basics. Internet traffic largely breaks down to either files or commands. Files could be a pdf, spreadsheet, image or movie, etc. Commands are instructions to do something - typically managed by interaction through a browser or client with an application somewhere. It's, of course, much more complicated than that, but for our purposes this will suffice.

Now commands and files have to be sent back and forth between your device and the far end server supporting your application with streams of Internet Protocol (IP) packets, each with its own header and payload. As you are likely aware, there are any number of network and/or security policies applied to each packet's header and/or payload along its journey to make sure it is 1) delivered along the most efficient path, 2) adherent to the rules of its transmission protocol, and 3) hopefully protected from harm or misuse.



Ok, probably for anyone reading this primer, that much is understood. But, repeating the basics serves a purpose. It gives us a baseline for discussing what is more important - and that is how IP packets are processed.

KERNEL-BASED PACKET PROCESSING

The prevailing packet processing model for decades has been 'kernel-based'. For every network device that receives, inspects, and subsequently sends the packet to its next hop (from your mobile device, desktop device, security camera, home theater system, etc. to the application server in a private data center or cloud somewhere, and back again) that packet is received on a network interface, and sent straight into the computer's operating system (OS) - in fact, all the way to the core of the OS (the kernel) - for determination on how it should be processed within that device.

Now, the kernel is the crown jewel of the OS. It manages the operation of the computer and its most important hardware - notably memory and CPU time. The kernel is also small, (relatively speaking). It is delicate. And, it is very, very busy when many computer processes request its attention.

So kernel processing of packets is designed around the principle of receiving one packet at a time, fetching an instruction from an instruction cache, performing that instruction on the packet, fetching the next instruction, performing that instruction, so on and so forth. Then that packet is sent on its merry way, and the second packet enters and goes through the same routine.

The FD.io analogy for explaining this is a good one, so we'll stick with it. Consider the problem of a stack of lumber where each piece of lumber needs to be cut, sanded, and have holes drilled in it. There are two ways of doing the job. Cut, sand, and drill each board one at a time. Or, cut all boards, then sand all boards, then drill all boards. The second approach will save loads of time as you'll avoid changing tools with each process step on each board.

Kernel-based processing is the former approach. On robust CPUs, e.g., Intel® Xeon® class processors, packet forwarding with stock Linux tops out at 2 million packets per second (Mpps) - and can easily be stymied by intracore locking and other effects. With experimental technologies, Linux has been shown to make some gains in artificial benchmarks, such as dropping all received packets, but a lot of work is still required, and VPP is available today.

Now, if one of the above-mentioned devices has a 10 Gbps interface, how will you process packets fast enough to fill that pipe? 10 Gbps line rate processing of the smallest (most CPU intensive) packets we have to deal with (64-byte packets, which is 84 bytes on the wire) is equivalent to 14.88 Mpps. Multiple Linux systems strapped together with a load balancer will consume a lot of cost, space, heat, etc. for a single 10 Gbps link - so you see where this is going. Extravagant CapEx and OpEx will be required.

Alternatively, you could opt for an expensive, vendor-proprietary application-specific integrated circuits (ASIC) or field programmable gate array (FPGA) solution. Well, that won't be cheap - and you'll also just have begun your subscription to 'vendor lock-in'.

VECTOR PACKET PROCESSING

What if packet processing could be freed from the constraints of the kernel? What if the instruction cache could be applied to an array of packets simultaneously, instead of one at a time?

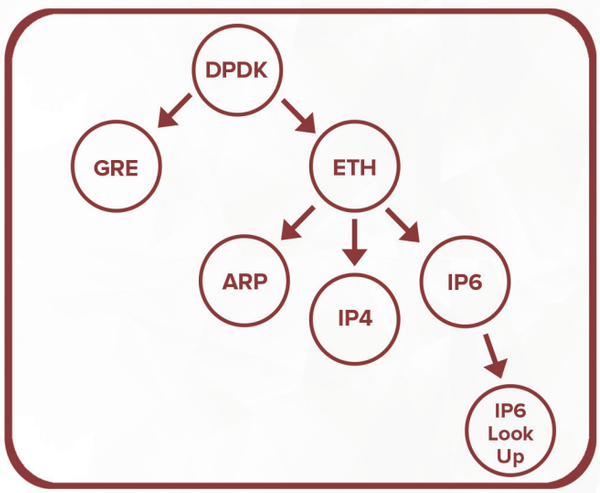
It's no longer a 'what if'. It's a reality. Introducing Vector Packet Processing (VPP). VPP is an open source version of Cisco's Vector Packet Processing technology - which is central to over $1 billion of Cisco ASA and CSR products. So, the core technology is proven. In essence, VPP is a modular packet processing node graph abstraction, where each node processes a vector of packets to reduce CPU I-cache thrashing, and is made extensible and dynamically reconfigurable via plugins. Let's break that down a bit.

VPP moves the packet processing workload out of kernel space and into user space. User space is where programs and libraries (that the OS uses to interact with the kernel, e.g., software that performs input/output, manipulates file system objects, application software, etc.) reside. As a result, there is much more 'elbow room' to manage cache-based instruction sets.

First, VPP gathers all packets from the device's network IO layer and groups them into a 'vector':

VECTOR

It then processes that vector of packets through a packet processing graph:



PACKET PROCESSING GRAPH

DPDK: Data Plane Development Kit

GRE: Generic Routing Encapsulation

ETH: Ethernet

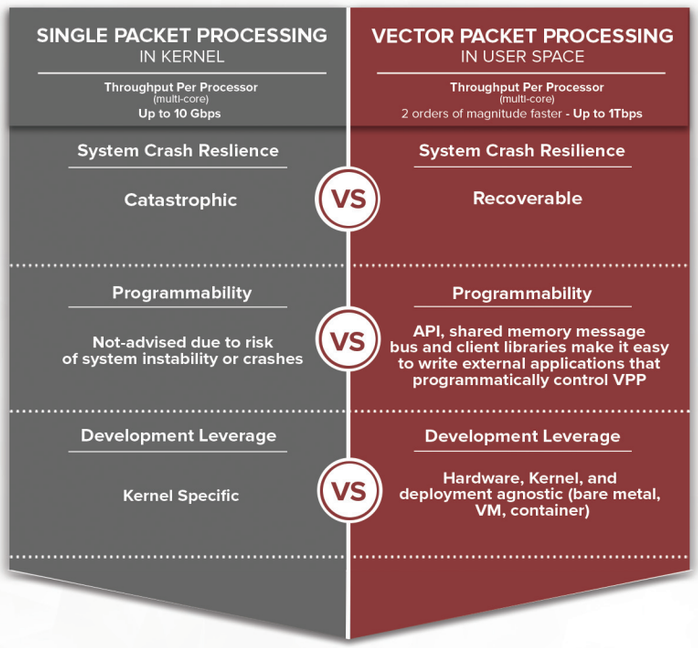
ARP: Address Resolution Protocol

IP4: Internet Protocol (Version) 4

IP6: Internet Protocol (Version) 6

Here is the key performance breakthrough. Rather than processing each packet through the entire processing graph, and then fetching the second packet and processing it through the entire graph, VPP fully processes the vector of packets through the first graph node before moving on to the next graph node. The first packet in the vector 'warms up' the instruction cache, so remaining packets can be processed extremely fast - sharply reducing the cost of processing each subsequent packet in the vector. This leads to 1) very high performance for processing a single packet, and 2) statistically reliable performance in processing a large number of packets over time. Additionally, VPP will often prefetch what it knows to be the next packet(s), ensuring that the CPU doesn't stall while the next packet is fetched from RAM. As a result, throughput is high and latency is consistently low.

Let's go back to our performance example above - where we explained a number of systems will be required to fill a 10 Gbps pipe with 64-byte packets using kernel processing. And since 10 Gbps is relatively pedestrian these days, let's kick it up a notch. Suppose you need 100 Gbps networking. VPP makes that achievable in software. 100 Gbps is a 10X jump over 10 Gbps, so we now need to process between 8 and 148 Mpps - depending on packet frame size. If the packet frames are large, we can forward 100Gbps - on a single core. If packets are tiny, we can process 100 Gbps on 10 cores. Normal internet traffic is a mix, so we'll practically land somewhere in between. Either way, this leads to a dramatic reduction in CapEx and OpEx relative to kernel processing.



While VPP makes software-based packet processing extremely attractive, there are still times when hardware acceleration is appropriate. Fortunately, the graph node architecture allows hardware acceleration to be easily inserted. As an example, computationally intensive traffic processing applications like hardware cryptographic acceleration - which helps offload the performance demands of securing and routing Internet traffic - can appear as just another graph node.

PROGRAMMABILITY / EXTENSIBILITY

By now you may be wondering how to put VPP to work for your specific purposes. Since a processing graph can be made to perform virtually any networking or security function at high speed, how do you bend it to your will? Good news. VPP easily lends itself to external programmability and extensibility.

VPP is equipped with a high-performance, low-level API. The API works via a shared memory message bus. Messages are passed along the bus as specified by a simple IDL (Interface Definition Language) used to create C and Java client libraries - making it straightforward to write external VPP control applications.

Further, FD.io is openly agnostic to any flavor of high-level API required for remote programmability. Apply any management agent you like, and then tap VPP functionality represented in YANG models via NETCONF and RESTCONF.

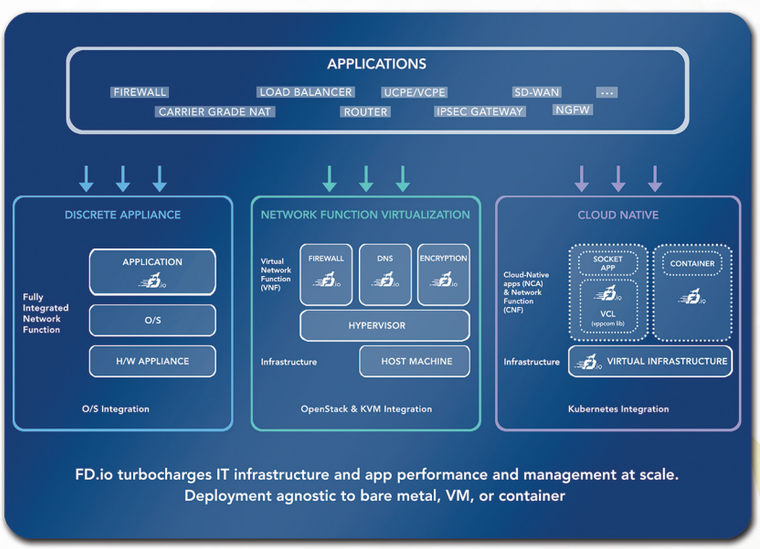
By extension, if VPP is remotely programmable via a RESTful API, it can certainly be integrated with 3rd party configuration management tools like Ansible, Chef, Puppet, or SaltStack.

Further, VPP is made extensible and dynamically reconfigurable via plugins.

DEPLOYMENT

FD.io is a collection of projects and libraries chartered to enable flexible, programmable and composable services on generic hardware platforms. VPP, of course, figures prominently into the mix. It runs in user space on multiple architectures including x86, ARM, and PowerPC architectures on both x86 servers and embedded devices.

As shown in the diagram below, VPP (represented in each of the three deployment models by the FD.io logo) is truly hardware, kernel, and deployment (bare metal, VM, container) agnostic:



PRODUCTIZATION

VPP is the wave of the future for all secure networking packet processing needs. Its packet processing power, programmability, manageability, and deployment flexibility make it essential for consideration by any organization looking for secure networking performance and scale. But VPP is an open source project, not a product. Open source projects have the advantage of community-driven development creativity and time-to-market. But development projects must still be productized in order to meet the demands of real-world deployment. That requires rigorous software integration, testing, packaging, distribution, and support. The expertise, investment, and effort required to effectively productize an open source project - let alone one experiencing rapid advancements - is not for the faint of heart.

Productizing open source secure networking projects into enterprise and service-provider ready solutions is Netgate's expertise. And TNSR is the answer to VPP productization.

To learn more about Open Source click here and TNSR click here.