The Datacenter Network Stack

Tom Herbert
Facebook
USA
therbert@fb.com

Lawrence Brakmo
Facebook
USA
brakmo@fb.com

Dave Watson
Facebook
USA
davejwatson@fb.com

ABSTRACT

Efficient, robust, and secure networking within the datacenter is achieved only through a mutually beneficial relationship amongst networking infrastructure, servers, and applications. At Facebook we are rethinking the server, or host networking, piece of this equation. Our vision is to create a *datacenter network stack* that provides unprecedented scalability, performance, and extensibility. The scope of this work encompasses all aspects of the stack including APIs, security, transport and network layer protocols, network virtualization, and network adapters. At the API layer, we are investigating alternatives to the venerable TCP sockets API. For security the intent is to provide a feasible solution to secure all data in flight. On the transport side, we are implementing Datacenter TCP optimizations and exploring ways to realize the full potential of IPv6 in the datacenter. The goal of network virtualization is to help create a completely malleable and resource efficient datacenter. We are leveraging advanced protocol features in networking adapters and are defining a framework to make the network stack easily programmable and extensible.

Categories and Subject Descriptors

• Networks → Network components → End nodes → Network servers; *Layering*, *Programming interfaces*

Keywords

Host networking; network stack; IPv6; datacenter

1. INTRODUCTION.

With more than one and a half billion active monthly users, Facebook is one of the most accessed sites on the Internet [11]. Providing services at planetary scale requires an extensive infrastructure. The Facebook infrastructure is composed of a number of geographically distributed networked datacenters, where each datacenter houses many thousands of interconnected servers. In order to continue to scale in terms of number of users and new services, both the datacenter and its networking infrastructure must continually evolve. In this paper we focus on efforts in evolving and revolutionizing server networking. Our goal is to define a *datacenter network stack* that provides performance, scalability, security, programmability, and extensibility to meet the requirements of future generations of applications and services.

Facebook datacenters adhere to the canonical datacenter model of servers housed in racks that are interconnected within the datacenter by a multi-tier network infrastructure. Servers provide the platform on which applications run. Each server connects to a Top of Rack switch (TOR) via Ethernet, and each TOR has uplinks connecting it to the datacenter networking fabric. Dedicated routers provide inter datacenter communication over Facebook's private backbone network, which in turn provides connectivity to the Internet at multiple Points of Presence (POPs). In this paper we focus on the network stack in the servers. The network stack is the server's point of connectivity to the rest of the world. It interfaces with the physical network, terminates transport layer protocols, and provides the user interface to networking.

The datacenter network stack can be thought of as a modification or extension to the traditional network stack. A network stack encompasses the application layer interfaces to the network, the implementation of various protocol layers and related plumbing, and the interfaces to the network adapter and, by extension, the network adapter itself. The APIs and protocol implementation are provided in a software stack as part of the host operating system, which in the case of Facebook servers is Linux. The network adapter, or Network Interface Card (NIC), is hardware that performs packet reception and transmission in a coordinated orchestration with the host software stack.

The remainder of this paper is organized as follows. In §2 we present the datacenter network stack developed for the Facebook datacenter. This stack includes the typical application, network, transport, and link layers; it also introduces security and virtualization as key layers. For each layer we are developing specific solutions. At the API layer we present Kernel Connection Multiplexer, which provides an efficient message-oriented interface on top of standard TCP sockets. For security we describe Transport Layer Security (TLS) in the kernel. At the transport protocol layer we present our efforts around Datacenter TCP (DCTCP). We describe the characteristics and benefits

of deploying IPv6 in the datacenter, and we present Identifier Locator Addressing (ILA) that leverages the large address space of IPv6 to provide an efficient solution for network virtualization. At the lowest level of the stack, we describe the interactions between the host network stack and the NIC in order to leverage common offload features, as well as a framework for "user programmability" of NICs and the network stack. We conclude the paper in §3.

2. DATACENTER NETWORK STACK

The abstraction for the datacenter network stack is shown in Figure 1. The foundation of the stack is the traditional network stack that has been augmented for the datacenter. Security and network virtualization are incorporated as relevant layers.

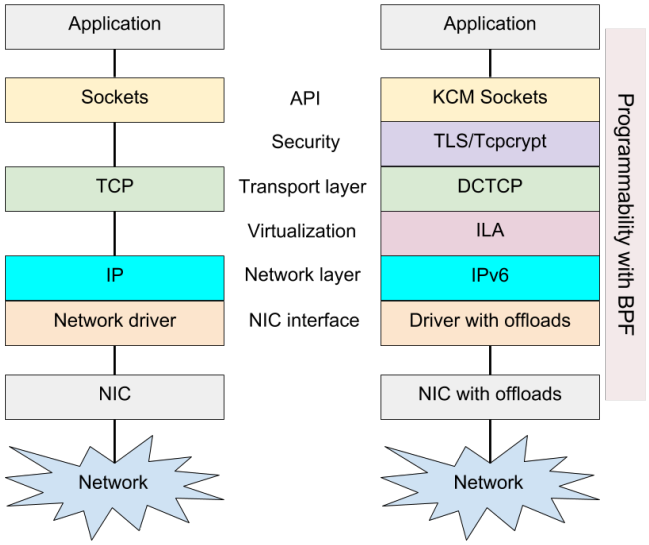

Figure 1. Traditional (left) and datacenter network stacks

In the remainder of the section we discuss the components of the datacenter network stack. The ordering of this dicussion is top down from the application to the network. This progression highlights the layering principle that higher layers in the stack set requirements for lower layers, but the converse is generally not true. Stack and NIC programmability has interactions across all the layers and is described last.

2.1 KCM Sockets

Kernel Connection Multiplexor (KCM) [7] is a mechanism that provides a message-based interface over TCP for generic application protocols. With KCM an application can efficiently send and receive application protocol messages over TCP using datagram sockets.

2.1.1 Motivation

The motivation for KCM is based on the observation that most communications in the datacenter exhibit transactional and message based semantics. For example, memcache [39] is widely used as a distributed cache server. A memcache server receives requests for cached objects and replies with the corresponding object. Remote Procedure Call (RPC) frameworks, such as Thrift [43] or GRPC [17], also define transactional and message-based communications.

While the application level communications are message oriented, the underlying transport protocol, TCP, provides a stream and has no concept of message boundaries. The normal method is to define an application protocol with its own framing that can be mapped onto a TCP stream. The framing of application messages itself is not a problem, however efficiently multiplexing and load balancing messages amongst threads in a multi-threaded application is nontrivial. This is an API issue in that TCP sockets provide no atomicity guarantees for sending or receiving application layer messages on the stream. The result is that an application must implement the atomicity guarantees itself, which necessitates coarse-grained locking and other inter-process communication amongst its threads.

KCM allows an application to send messages over TCP with atomicity guarantees provided by the kernel. This permits I/O operations to be done in parallel in a multi-threaded application without needing locks or other synchronization primitives.

2.1.2 Implementation

KCM implements an NxM multiplexor as diagrammed in Figure 2. An instantiation of KCM is constructed from a plumbing of KCM sockets, a multiplexor, connection groups, and bound TCP sockets via stream parsers.

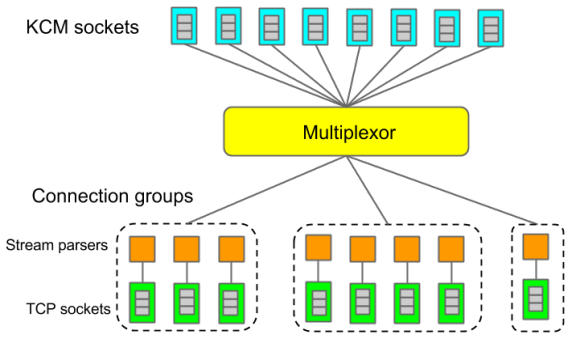

Figure 2. Kernel Connection Multiplexor plumbing

2.1.2.1 Stream parser

KCM employs the *stream parser* [19] (strparser) which is a utility that parses messages of an application layer protocol running over a TCP connection. The stream parser works in conjunction with an upper layer in the kernel to provide kernel support for application layer messages. The stream parser provides a generic mechanism for performing message delineation on a TCP stream for an application layer protocol.

In order to delineate messages in a TCP stream on receive, the kernel must parse the application layer framing. Berkeley Packet Filters (BPF) [31] is employed as a generic solution. When a TCP socket is attached to a

stream parser a BPF program loaded by the application is indicated. The program is able to parse the stream and return the length of the next message. This is sufficient to delineate messages for most message-based protocols so the stream parser can be used across a wide range of applications.

2.1.2.2 KCM multiplexor

TCP sockets are bound to the bottom of a KCM multiplexor using a stream parser instance for each connection. The bound connections are organized into *connection groups*. These are collections of TCP connections to the same destination and allow a user to send messages to different destinations on a single KCM socket by specifying the connection group to send on.

KCM sockets provide the user interface to the multiplexor. A new Linux socket family (AF_KCM) is defined that allows datagram and sequence packet types. All the KCM sockets bound to a multiplexor are considered to have equivalent function, and I/O operations on different sockets may be done in parallel without needing synchronization.

The *multiplexor* provides message steering. In the transmit path, messages written on a KCM socket are sent atomically on appropriate TCP sockets. Similarly, in the receive path, messages received over TCP are atomically steered to KCM sockets to achieve load balancing.

2.1.2.3 Performance

Thrift [43] has been modified to optionally use KCM. By default, Thrift ties a single connection to a single thread; KCM allows multiple connections to be load balanced across multiple threads. We tested Thrift/KCM on a production application. With KCM enabled, average latency is unchanged, but the 99th percentile latency is reduced 10%.

2.2 TLS in the kernel

Transport Layer Security (TLS) [9] and its predecessor Secure Sockets Layer (SSL) [14] provide privacy and data integrity between two communicating applications. The protocol is composed of two layers: the TLS Record Protocol and the TLS Handshake Protocol. The TLS Record protocol is layered on top of a reliable transport Protocol (e.g. TCP) and provides privacy and integrity checks for encapsulated messages.

TLS is well deployed on the Internet as the security protocol of HTTPS [38]. Typically, it is implemented as a user space library that is linked into applications. For servers, data being transmitted often originates in the kernel as opposed to user space. For example, in a video server the content may reside in files on a disk or at least in the page cache (or disk cache) of the kernel. Before security was a consideration, data could be sent directly from the page cache to the network using the *sendfile* system call obviating the data copies to and from user space. To retain this efficiency with security in effect we have implemented TLS in the Linux kernel (inspired by the design to implement TLS in the FreeBSD kernel [41]).

2.2.1.1 Implementation

The initial implementation [10] supports the 128-bit advanced encryption standard (AES) using Galois Counter mode (also known as *gcm(aes)*). Symmetric encryption and decryption are performed in the kernel as well as necessary framing. The TLS handshake is performed in user space. The basic API is that a crypto socket (with Linux socket family *AF_ALG*) and a regular TCP socket are created. TLS handshake with the remote endpoint is performed on the TCP socket to establish keys, the ciphers, Initialization Vectors (IVs), and other security parameters. The keys and ciphers are set on the crypto socket with a *setsockopt* system call. The TCP socket is then bound to the crypto socket by another system call to plumb the data path of the TCP connection to go through the crypto socket. The application may then read or write plain text data on the crypto socket; the kernel performs TLS encryption and decryption transparently.

TLS in the kernel reduces CPU utilization by 2-7% compared to that of the OpenSSL user space library (due to the reduced number of data copies). We are investigating whether the TLS data path can be further offloaded to the NIC.

2.2.1.2 kTLS+KCM

TLS in the kernel is also a prerequisite for using KCM with secured connections. Decryption must occur in the receive data path before KCM parses for message delineation, and similarly encryption must occur after KCM transmit processing. Figure 3 depicts an application using the three methods of TLS: the traditional method of using a TLS library in the application, using a kTLS socket, and using KCM socket with a kTLS socket. Note that in kTLS+KCM case the KCM socket is stacked over a TLS socket so there are two stream parsers in the path—one that parses the TLS records on the TCP stream and the other that parses the unencrypted stream into application messages.

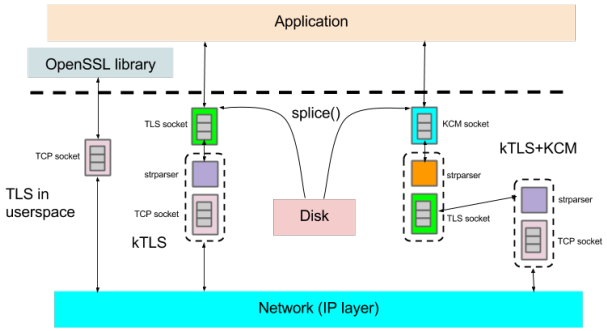

Figure 3. TLS in userspace, kTLS, KCM+kTLS

2.3 Datacenter TCP

Datacenters present unique opportunities for improving the performance of TCP [42,25]. We are in the process of

deploying various enhancements to TCP and in this section we discuss one of them: Data Center TCP (DCTCP) [1]. DCTCP uses Explicit Congestion Notification (ECN) [36] in the network to provide multi-bit feedback to the end hosts. Whereas the standard TCP response to ECN signals tends to underutilize the network links, DCTCP is able to achieve full link utilization while reducing latency as compared to Reno and Cubic [12] congestion control algorithms.

Figure 4 shows the average rates and latencies for Reno, Cubic, and DCTCP traffic for one experiment consisting of three hosts sending to one host, all within a rack, in a 10G environment. Each flow consists of back-to-back 1MB RPCs. For this experiment the number of flows per sending host varies between one and thirty-two. The vertical bars indicate the achieved rate (goodput). For one to four flows all TCP variants achieve similar rates (~3Gbps per host). At higher loads, Cubic's throughput decreases about 17%. The 99.9th percentile latencies are shown by the red diamonds and correspond to the rightmost vertical axis. Even at low loads Reno and Cubic have much worse 99.9th percentile latencies.

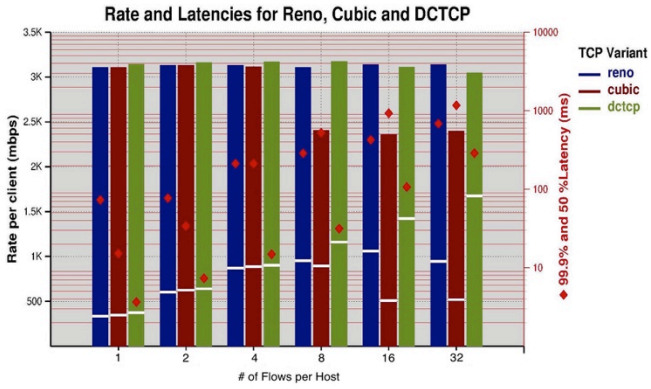

Figure 4. Comparison of Reno, Cubic, and DCTCP

The white lines indicate the 50th percentile latencies. For up to twenty-four flows (eight per sending host) 99.9th percentile latency tracks very close to the 50th percentile latency when using DCTCP. At higher loads, forty-eight and ninety-six flows, losses start to occur when using DCTCP (they occur starting with three flows with Reno and Cubic) resulting in RTOs (retransmission timeout) that increase the 99.9th percentile latencies. There are very few losses with DCTCP, but the congestion window (cwnd) is very small in these cases (six for forty-eight flows and three for ninety-six flows) resulting in more RTOs.

Decreasing the TCP minimum RTO from 200ms to 20ms keeps the 99.9th percentile latency close (within a factor of 1.6) to the 50th percentile latency even for ninety-six flows. Thus when using DCTCP, higher loads will not result in 99.9th percentile latencies that are significantly higher than 50th percentile latencies.

DCTCP is intended to be used only for intra datacenter traffic. What should one use for traffic arriving from outside the datacenter? Consider the case when two hosts send to a third host where one of the senders and the receiver are in the same datacenter while the second sender is 100ms RTT (Round-trip Time) away. Each of the senders has four flows to the receiver doing back-to-back 1MB RPCs. Table 1 shows the performance of local versus remote traffic. The first row provides the baseline, TCP Cubic local vs. TCP Cubic remote without ECN, and it shows the well-known RTT unfairness of TCP. For the following rows ECN has been enabled so we can use DCTCP for the local traffic. The rate of the remote traffic degrades even more if DCTCP is used for both local and remote traffic from 32Mbps to 6.5Mbps. The unfairness is worst when the local traffic is using DCTCP and the remote traffic is using Cubic; the rate for the remote traffic is now only 2.5Mbps.

Table 1. Local vs. remote traffic with DCTCP

ECN	Local ca	Remote ca	Local rate	Remote rate
No	Cubic	Cubic	2363 Mbps	32 Mbps
Yes	DCTCP	DCTCP	2375 Mbps	6.5 Mbps
Yes	DCTCP	Cubic	2375 Mbps	2.5 Mbps
Yes	DCTCP	Cubic-ECN	2448 Mbps	37 Mbps

To solve this problem, we use a modified version of Cubic (Cubic-ECN) that marks packets as ECN enabled but ignores ECN markings (i.e. it doesn't decrease its congestion window due to ECN markings). With this both the local and the remote traffic achieve better performance than the Cubic baseline. An alternative solution would be to use separate network queues for ECN tagged versus untagged traffic, but this involves higher operational complexities in that all network devices need to be reconfigured and reasonable queue weights need to be assigned.

2.4 Identifier Locator Addressing

Identifier Locator Addressing (ILA) [18] is a method to implement overlay networks for network virtualization. Unlike many other techniques for implementing overlay networks, ILA does not employ encapsulation allowing zero transport overhead and a very efficient solution.

2.4.1 Motivation

The goal of virtualization at Facebook is to transform the existing datacenter with rigid resource and job scheduling into a malleable one. This model of virtualization differs from that typically applied to cloud computing— jobs are run in containers as opposed to virtual machines, and there is only one network tenant so issues of multi-tenancy, such as network isolation, have lower priority. Ultimately, we expect to virtualize all resources so that jobs can be flexibly

scheduled, resources can be added to running jobs, and running tasks can be seamlessly migrated between servers per the discretion of the job scheduler.

Network virtualization is typically implemented as an overlay network where applications use virtual addresses to communicate with peers on the overlay network. A virtual address indicates no physical location so it must be mapped to the physical address where the addressed entity resides for transmission. To send a virtually addressed packet over the physical network it can either be encapsulated or its destination address can be translated to the physical address. Identifier Locator Addressing is a method for the latter, it is a means to implement network virtualization without encapsulation.

In addition to decoupling logical and physical addressing for virtualization, ILA allows for replacing some other use-cases of IP tunneling and eliminates much of the need for network layer encapsulation. For example, in Layer 4 load-balancers with direct server return (DSR) an IP tunnel is often used to deliver a packet to a machine in the load-balancing pool. With ILA, the tunnel could be replaced by address translation, solving the issues with Path MTU discovery in the underlay network path of the tunnel.

2.4.2 Details

The basic concept of ILA is that the 128-bit IPv6 address is split into a sixty-four-bit *locator* and a sixty-four-bit *identifier* (similar to ILNP [3]). The identifier expresses the identity of a communicating entity ("who") and the locator expresses the location of the entity ("where"). Each physical host is assigned a unique 64-bit address prefix that serves as its locator, and the network is configured to route locators accordingly. Each virtual entity (such a task or VM) is assigned a unique identifier.

ILA defines a checksum neutral translation to ensure that any encapsulated transport layer checksum that includes IP addresses in their pseudo header remains correct. The idea is that when one area of the packet is modified making the checksum invalid this can be corrected by making a complementary change to a different 16-bit field covered by the same checksum. For ILA the low order 16 bits of the identifier are used for this checksum adjustment value.

Applications use externally visible addresses that contain identifiers for initiating communications. When a packet is actually sent, a translation is done that overwrites the first sixty-four bits of the destination address with a locator. The packet is then forwarded over the network to the host addressed by the locator. At the receiver, the reverse translation is done so that the application sees the original address. An external control plane provides identifier to locator mappings.

ILA performs NAT translation in the data path on the upper sixty-four bits of the destination address (or upper sixty-four bits of the source address in the case of multicast). The steps of translating a packet are:

1) Lookup the sixty-four bit identifier (lower sixty-four bits of destination) in a table
2) If a match is found then:
    a. Overwrite locator (upper sixty-four bits of destination) with the new locator
    b. Set checksum neutral adjustment (low order 16 bits of identifier) to compensate for checksum being modified

An example of ILA translation is shown in Figure 5. In this scenario a source host is sending packets to a virtual host with address 3333::1. Initially, the host may not have the identifier to locator mapping so a packet is sent without translation and is forwarded to an ILA router that maintains a complete set of identifier to locator mappings (arrow 1). The ILA router performs translation replacing the 3333:: prefix with 2222:1:: (the locator for identifier ::1), resulting in a destination address of 2222:1::1. The packet is then sent on the network (arrow 2). The destination host receives the packet and rewrites the destination address back to the un-translated address of 3333::1. The packet is then delivered the application. The ILA router may send a special "redirect" message back to the source (arrow 3) informing it of the identifier to locator mapping so that future packets can be translated at the source host and sent directly to the destination (arrow 4).

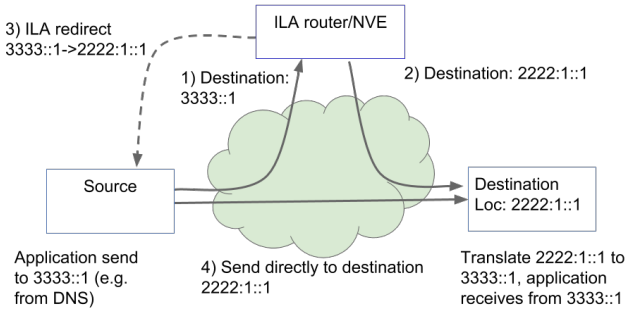

Figure 5. ILA translation example

ILA has less overhead than any encapsulation techniques. It does not increase the size of packets like encapsulation does, so there are no associated issues with Path MTU (PMTU) discovery or fragmentation. Since only addresses are being modified, stateless network offloads and accelerations (RSS, ECMP, TSO, etc.) work transparently. The downside of ILA is that it is less extensible than some encapsulations techniques (Generic UDP Encapsulation [22] for example) so it may not be appropriate for all cases. Also, this only makes sense to do in IPv6!

2.4.3 Performance

ILA is used in the datacenter to virtualize tasks and jobs. At its fullest incarnation, all intra datacenter communications might be targeted to virtual ILA addresses. This is basically adding a new virtualization capability to existing services

in a datacenter, so there is a strong requirement that this functionality is transparent and does not degrade performance. Table 2 provides some benchmark data for ILA. The benchmark is the netperf TCP_RR test which emulates a simple request/response communication. These results are from running 200 concurrent streams with a payload size of one byte. We compare ILA against native IPv4 and IPv6 performance, as well as against VXLAN [30], which is a popular encapsulation protocol for network virtualization. ILA performance is within 2% of native IPv4 and IPv6, which is better than VXLAN or other layer 3 encapsulation protocols. The overhead of ILA is a hash table lookup and checksum neutral handling; ILA does not incur overhead in additional bytes on the wire nor layered processing of multiple IP headers as in VXLAN.

Table 2. Request/response performance with ILA

Case	Throughput 10^6 tps	Latency 50/90/99%ile (usecs)	CPU utilization
IPv4	1.65(baseline)	108/194/381	82.71%
IPv6	1.65 (-0%)	109/195/374	83.11%
ILA	1.63 (-1.8%)	112/198/358	82.71%
VXLAN	1.23 (-28%)	142/249/552	86.04%

2.5 IPv6 in the datacenter

Facebook was one of the first of the major datacenter operators to switch core networking to use IPv6. The primary reason for switching to IPv6 is address space exhaustion. This applies to the datacenter in the sense that the private IPv4 address space (10/8 prefix) [37] can be exhausted. While the number of hosts in Facebook is well less than the theoretical maximum that can be addressed in a private 10/8 network (sixteen million hosts), due to sparseness in multi-level hierarchical allocation it is possible to exhaust the space with a much smaller number. Periodic renumbering of hosts across the datacenter to consolidate address assignments could be done to further extend the lifetime of IPv4, however this is quite invasive and not a sustainable long-term strategy. Moving to IPv6 incurs significant up front effort, but the benefits of nearly limitless scalability outweigh the costs.

In terms of the network stack there has been much effort in bringing IPv6 to feature and performance parity with IPv4. A particular area of weakness has been the routing table. Improvements in route lookup and route caching were addressed in Linux 4.2 [29]. Similarly, there are still disparities between IPv6 and IPv4 in hardware features and testing of IPv6 with different hardware. To a large extent, this is a non-technical issue as some hardware vendors still choose to prioritize their efforts for IPv4.

The forward-looking benefits of IPv6 in the datacenter are:

- Addressing flexibility
- IPv6 flow label
- IPv6 extension headers

2.5.1 Addressing flexibility

IPv6 addresses are 128-bits in size making them far larger than IPv4 addresses. The enormity of this space allows new uses of addressing beyond just simply assigning an address to each host. For instance, there are proposals (such as Identifier Locator Addressing) to assign each physical host in the network it's own 64-bit prefix. This means that 2^{64} objects could be addressable *within* each host.

One particular problem that IPv6 can help solve is dynamic port assignment and service lookup. With IPv4, applications running on the same host (using the same host address) share a common transport port number space. So if two instances of a service run on a host, they need to listen on different ports. This necessitates complex dynamic port assignment and service discovery methods. In IPv6, one can assign each instance of a service its own IP address. In this way instances of a service are distinguished solely by IP address and are free to use the any port numbers without possibility of conflict. This greatly simplifies service lookup to the point that simple DNS could be used.

2.5.2 Flow label

The IPv6 flow label is a 20-bit field in the IPv6 header [2]. Its purpose is to identify packets as belonging to the same flow. Intermediate devices may use the flow label to make flow aware forwarding decisions.

The IPv6 standard does not specify how the flow label is used nor what its structure must be; different protocol implementations may define the use of flow labels. There are two general possibilities: *stateless* and *stateful*. Stateless flow labels could be used in load balancing such as done in ECMP (see section 4.6.1). Stateful flow labels may be used to indicate a flow specific behavior such as network path selection or other flow specific filtering.

2.5.3 IPv6 extension headers

Extension headers are the method of extensibility for IPv6 [8]. They are similar to IPv4 options except that extension headers are not part of the IP header and thus are not subject to a maximum length (IPv4 options are limited to forty bytes). The flexibility of extension headers allows defining new, possibly custom extensions. One active area of interest for the use of extension headers in the datacenter is Segment Routing [35].

As is the case of IP options, many devices do not handle extension headers [13,16]. This is particularly true in devices that need to parse beyond the options or extension headers to provide a feature (such as DPI for flow based ECMP). Parsing to find a transport header in packets with IPv6 extension headers is conceptually harder than parsing packets with IPv4 options since the device must parse a list

of headers to locate the transport header. Use of the flow label as described above should greatly mitigate the need for deep packet inspection by devices with IPv6.

2.6 Networking offloads

Offload mechanisms are techniques that are implemented separately from the normal protocol implementation of the stack with the intent to optimize or speed up protocol processing. Hardware offload is performed within a NIC on behalf of a host. Software offload is implemented in a lower layer than protocol processing, typically near or in NIC drivers. Following the "less is more design" principle [34], there are a relatively small number of offload mechanisms that are essential for deployment [20]:

- Receive load balancing
- Checksum offload (transmit and receive)
- Segmentation offload (transmit and receive)

For each of these offloads, there are both protocol-specific techniques that involve the device parsing transport layer protocols, and protocol-generic techniques that don't require parsing the transport layer and work with arbitrary protocols. In light of protocol ossification, we advocate that new implementation and features be protocol-generic.

2.6.1 Receive Load balancing

Networking hardware and software stacks implement a variety of mechanisms to perform load balancing (statistical multiplexing) of packets across a set of resources. Switches implement Equal Cost Multipath routing (ECMP) to distribute packets over multiple network paths [27]. Network Interface Cards (NICs) implement Receive Side Scaling (RSS) to distribute packets over a number of host receive queues [32]. The Linux stack implements Receive Packet Steering (RPS) as a software analogue for RSS, and Receive Flow Steering (RFS) that steers packets to the CPU where the receiving process runs [21]. Preferably, load balancing is done at the granularity of packet flows (sequences of packets that belong to the same communication or connection).

Devices perform hash computations on packet headers to classify packets into flows or flow buckets. Hashes are either over a three-tuple consisting of the source address, destination address, and protocol; or over a five-tuple that also includes the source and destination ports. The five-tuple hash provides more granularity and better load balancing, but often is only supported in hardware for TCP or UDP packets. This has motivated the use of UDP encapsulation protocols, where the source port is set to act as a flow label for encapsulated packets [20].

Our preferred direction is to use IPv6 flow labels. The flow label in an IPv6 header can be set to refer to the encapsulated flow [5]. Hardware can compute a flow hash over a three-tuple of source address, destination address, and flow label providing a suitable approximation of a five-tuple hash even for packets with non-TCP and non-UDP protocols.

2.6.2 Checksum offload

The Internet checksum is a one's complement checksum that is used in TCP, UDP, and other Internet protocols. Checksum calculation is known to be an expensive operation to perform by a host CPU [4]. Most NICs have capabilities to offload checksum calculations for both transmit and receive.

In the protocol-specific variants of transmit checksum offload, the NIC parses each packet to determine the offset of a transport checksum field, performs the checksum calculation including the checksum over any pseudo header, and writes the result in the checksum field. In the receive path, a NIC performs checksum validation on its own and returns an indication to the host stack that one or more checksums in the packet are verified to be correct.

In protocol-generic transmit checksum offload, the host provides the NIC with the start offset in a packet for checksum calculation and the offset of the transport layer protocol checksum field (`NETIF_HW_CSUM` feature in Linux). The checksum field is initialized to the complement (bitwise not) of the pseudo header checksum. The device performs the checksum calculation from the start offset to the end of the packet and writes the result at the offset of the checksum field.

In protocol-generic receive checksum offload, a NIC computes the one's complement checksum over all (or some predefined portion) of a packet and returns the result to the host (`CHECKSUM_COMPLETE` feature in Linux). The host stack uses this checksum to verify any transport checksums in the packet (both for inner and outer headers).

2.6.3 Segmentation Offload

Segmentation offload refers to techniques that reduce CPU utilization on hosts by having the layers of the stack operate on large packets.

In Large Segmentation Offload (LSO), a transport layer creates large packets greater than MTU size (Maximum Transmission Unit). It is only at much lower point in the stack, or possibly the NIC, that these large packets are broken up into MTU sized packets for transmission on the wire. Similarly, in Large Receive Offload (LRO), small packets are coalesced into large, greater than MTU size packets at a point low in the software receive path or in a device. The effect of segmentation offload is that the number of packets that must be processed by layers of the stack is reduced, and hence CPU utilization is reduced.

Since segmentation operates at the transport layer (TCP for instance), it is difficult to create a truly protocol-generic implementation; this is particularly true in LRO where a device must parse transport headers of receive packets. The software analogues, Generic Segmentation Offload (GSO) and Generic Receive Offload (GRO), provide most of the

benefit of equivalent hardware offloads and allow the generality of software implementation for supporting different protocols.

2.7 Programmable network stack

It is desirable that applications can program some features of the network stack per their own processing or protocols. For instance, consider how one might implement GRO for an application layer transport protocol such as QUIC [6]. Without a programmable stack one would need to implement awareness of each application protocol in the kernel as specialized GRO handlers. With a programmable stack, a program can be injected at runtime that parses an application protocol and interfaces with a generic GRO engine to achieve the offload.

2.7.1 Programmability using BPF

Berkeley Packet Filters (BPF) [31,40] has long been the interface for packet filtering in the Linux kernel, extending this model to support general user programmability of the network stack is relatively straightforward (BPF has also been proposed as a solution for dataplane programmability in SDN [24]). This model can be further extended to allow programming network devices that implement appropriate support.

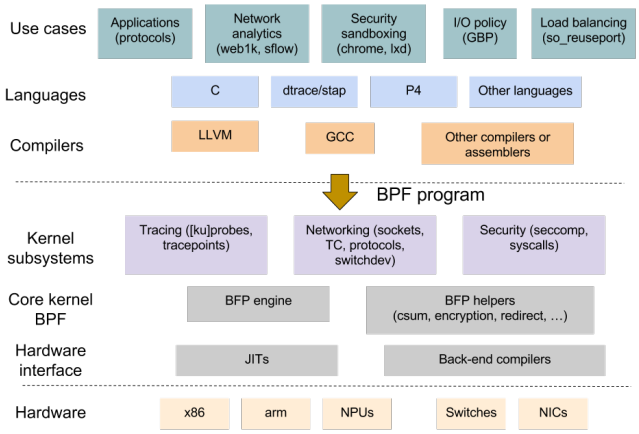

Figure 6. Programmability with BPF architecture

Figure 6 shows a layered architecture for a programmable network stack with BPF. At the top level are the "use cases". These include applications that offload protocol processing to the kernel (e.g. GRO for QUIC), network analytics, security provisioning and sandboxing, I/O policy, and load balancing. A "BPF program" can be written in one of several high level languages. Programs are compiled into BPF instructions that can be executed by the kernel with associated data consisting of an indication of attachment point, (e.g. tracing, networking), ancillary data structure configurations (e.g. hash table, array), and other type information. A BPF backend has been merged into LLVM that allows a simple means to build BPF programs [28].

A compiled BPF program is loaded into the kernel in the context of one of the subsystems. For example, a BPF program to parse application messages for KCM is registered in the core networking subsystem. Once the program is loaded it can be enabled at a number of hooks in the kernel. For instance, one hook allows a BPF program to be enabled on a UDP socket that is configured with *SO_REUSEPORT* [26,15] in order to provide fine grained control over how received packets are distributed to equivalent sockets. The BPF engine drives the execution of a BPF program; BPF helpers provide high performance library functions (such as a function to calculate an Internet checksum). A set of run-time APIs and related data structures (such as sk_buff [33] which is the meta data structure for a packet) are defined to allow programs to interact with the host network stack. Prior to execution, the kernel statically verifies BPF programs for safety.

A BPF program can be offloaded into a supporting networking device using one of two methods. An in-kernel Just-In-Time (JIT) compiler translates BPF instructions into a hardware specific representation that the kernel can load into the hardware. Alternatively, a back-end compiler can be used. In this case, a callout is made from the kernel back to user space providing the BPF program (instructions, configuration, and types) previously loaded in the kernel. A user space daemon executes a compiler to translate the program into a hardware specific representation and then arranges for the kernel to load the program into hardware. In this context, BPF serves as a portable intermediate code representation. A BPF program could be compiled to specific hardware targets: an FPGA, ASIC, or NPU for example. When programs are offloaded into hardware, the kernel arbitrates access to the hardware via the BPF API to provide resource isolation and safety.

2.7.2 eXpress Data Path

eXpress Data Path is being developed as a programmable, high performance, specialized application packet processor in the networking data path. Figure 7 illustrates the plumbing of XDP.

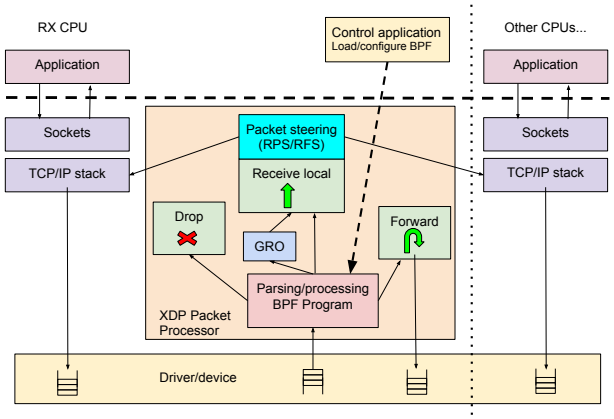

Figure 7. eXpress Data Path architecture

XDP defines a hook that is placed in the receive function of NIC device drivers. The hook calls a backend BPF program

passing only pointers to the raw packet. The program may perform parsing and modification of the packet. The function then returns an action code back to the driver. These are one of:

- **Pass** – pass packet for normal stack processing.
- **Transmit** – transmit packet on same port it was received. Presumably the program has set IP and Ethernet addresses appropropriately.
- **Drop** – drop the packet. This is an early drop which is useful to help in denial of service (DDOS) mitigation

XDP provides a programmable "bare metal" networking path. We have shown 20Mpps drop rate on a single CPU and 14Mpps forwarding performance. These numbers are faster than going through the full stack and are competitive with kernel bypass solutions such as DPDK [23].

3. CONCLUSION

The server network stack is a major component in datacenter networking. In this paper we show that this component is replete with opportunities for enhancement. Performance, security, and scalability are key objectives in these enhancements, and the datacenter network stack is expressly designed to achieve these objectives. This is a work in progress, and in some sense will always be, as we continue to evolve and occasionally revolutionize the network stack. We believe that the future direction is to continue to create more synergy with applications and networking infrastructure by programmability of the stack, opportunistically leveraging new hardware features, and development of new features in the network stack based on an application centric viewpoint.

4. ACKNOWLEDGMENTS

The authors would like to than Blake Matheny, Alexei Starovoitov, and Petr Lapukhov for there insightful review and suggestions.

5. REFERENCES

[1] Alizadeh, M., Greenberg, A., Maltz, D., Padhye, J., Patel, P., Prabhakar. B., Sengupta, S., and Sridharan, M., "Data Center TCP (DCTCP)" in *SIGCOMM '10*, 2010.

[2] Amante, S., Carpenter, B., Jiang, S., and Rajahalme, J., "IPv6 Flow Label Specification", *IETF* RFC6437, November 2011.

[3] Atkinson, R. and Bhatti, S., "Identifier-Locator Network Protocol (ILNP) Architectural Description", *IETF* RFC6740, November 2011.

[4] Braden, R., Borman, D., and Partridge, C., "Computing the Internet Checksum", *IETF* RFC1071. September 1988.

[5] Carpenter, B. and Amante, S., "Using the IPv6 flow label for equal cost multipath routing and Link Aggregation in Tunnels", *IETF* RFC6438, November 2011.

[6] Chromium, "QUIC, a multiplexed stream transport over UDP", Retrieved January 2016, https://www.chromium.org/quic.

[7] Corbet, J., "The kernel connection multiplexer", *LWN*, September 2015, https://lwn.net/Articles/657999/.

[8] Deering, S. and Hinden, R., "Internet Protocol, Version 6 (IPv6) Specification", *IETF* RFC2460, December 1998.

[9] Dierks, T. and Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.2", *IETF* RFC4246, August 2008.

[10] Edge, J., "TLS in the Kernel", *LWN*, Decemeber 2015, https://lwn.net/Articles/666509/.

[11] Facebook, "Number of monthly active Facebook users worldwide as of 3rd quarter 2015 (in millions)", December 2015, http://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/.

[12] Esterhuizen, A. and Krzwsinski, A., "TCP Congestion Control Comparison", Retrieved January 2016, http://www.satnac.org.za/proceedings/2012/papers/2.Core_Network_Technologies/15.pdf.

[13] Fonseca, R., Porter, G., Katz, R., Shenker, S., and Stoica, I., "IP Options are not an Option", *University of California at Berkeley*, Technical Report No. UCB/EECS-2005-24, December 2005.

[14] Freier, A., Karlton, P., and Kocher, P., "The Secure Sockets Layer (SSL) Protocol Version 3.0", *IETF* RFC6101, August 2011.

[15] Gallek, C., "Faster SO_REUSEPORT", *LWN*, December 2015, https://lwn.net/Articles/670020/.

[16] Gont, F., Linkova, J., Chown, T., and Liu, W., "Obserservations on the Dropping of Packets with IPv6 Extension Headers in the Real World", *IETF* InternetDraft draft-ietf-v6ops-ipv6-ehs-in-real-world, December 2015.

[17] Google, "GRPC", Retrieved January 2016, http://www.grpc.io/.

[18] Herbert, T., "Identifier-locator addressing for network virtualization", *IETF* Internet-Draft draft-herbert-nvo3-ila, April 2015. http://www.spinics.net/lists/netdev/msg388053.html

[19] Herbert, T., "Stream Parser", August 2016, https://www.kernel.org/doc/Documentation/networking/strparser.txt

[20] Herbert, T., "UDP Encapsulation in Linux", February 2015,

http://people.netfilter.org/pablo/netdev0.1/papers/UDP-Encapsulation-in-Linux.pdf.

[21] Herbert. T. and de Bruijn, W., "Scaling in the Linux Networking Stack", 2011, https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[22] Herbert, T., Yong, L., and Zia, O., "Generic UDP Encapsulation", *IETF* Internet-Draft draft-ietf-nvo3-gue, December 2015.

[23] Intel, "Intel® Data Plane Development Kit (Intel® DPDK)", December 2012, http://www.intel.com/content/dam/www/public/us/en/documents/presentation/dpdk-packet-processing-ia-overview-presentation.pdf.

[24] Jouet, S., "Programmable Dataplane", *University of Glasgow*, Retrieved January 2015, https://netlab.dcs.gla.ac.uk/uploads/files/a2f58b9e7814a64e77598c1b25028ba3.pdf.

[25] Judd, G., "Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter", *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implemenation*, May 2015.

[26] Kerrisk, M., "The SO_REUSEPORT socket option". March 2013, https://lwn.net/Articles/542629/.

[27] Lappetelainen, A., "Equal Cost Multipath Routing in IP Networks", *Aalto University School of Science and Technology*, March 2011.

[28] Larabel, M., "BPF Backend Merged Into LLVM To Make Use Of New Kernel Functionality", January 2015, https://www.phoronix.com/scan.php?page=news_item&px=LLVM-BPF-VM-Backend-Lands.

[29] Lau, M., "Linux IPv6 improvement: Routing cache on demand", July 2015, https://code.facebook.com/posts/1123882380960538/linux-ipv6-improvement-routing-cache-on-demand/.

[30] Mahalingam, M. *et al*., "Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks", *IETF* RFC7348, August 2014.

[31] McCanne, S. and Jacobson, V., "The BSD Packet Filter: A New Architecture for User-level Packet Capture", *Usenix93*, December 1992, http://www.tcpdump.org/papers/bpf-usenix93.pdf

[32] Microsoft. "Scalable Networking: Eliminating the Receive Processing Bottleneck—Introducing RSS", Retrieved January 2016, http://download.microsoft.com/download/5/D/6/5D6EAF2 B-7DDF-476B-93DC-7CF0072878E6/NDIS_RSS.doc.

[33] Miller, D., "How SKBs work", Retrieved January 2016, vger.kernel.org/~davem/skb.html.

[34] Miller, D., "Netdev 1.1 – Hardware Checksumming: Less is More", YouTube, 2015, https://www.youtube.com/watch?v=6VgmazGwL_Y.

[35] Previdi, S. *et al*., "IPv6 Segment Routing Header (SRH)", *IETF* Internet-Draft ietf-6man-segment-routing-header, December 2015.

[36] Ramakrishnan, K. and Floyd, S., "A Proposal to add Explicit Congestion Notification (ECN) to IP", *IETF* RFC3168. January 1999.

[37] Rekhter, Y., Moskowitz, B., Karrenberg, D., de Groot, G., and Lear, E., "Address Allocation for Private Internets", *IETF* RFC1918, February 1996.

[38] Rescorla, E., "HTTP Over TLS", *IETF* RFC2818, May 2000.

[39] Saab, P., "Scaling memcached at Facebook", December 2008, https://www.facebook.com/note.php?note_id=39391378919&ref=mf.

[40] Schulist, J., Borkmann, D., and Starovoitov, A.. "Linux Socket Filtering aka Berkeley Packet Filter (BPF)", January 2016, https://www.kernel.org/doc/Documentation/networking/filter.txt.

[41] Stewart, R., Gurney, J., and Long, S., "Optimizing TLS for High-Bandwidth Applications in FreeBSD", 2015, https://people.freebsd.org/~rrs/asiabsd_2015_tls.pdf.

[42] Tahiliani, R., Tahiliani, M., and Sekaran, K., "TCP Variants for Data Center Networks: A Comparative Study", *ISCOS '12*, 2012.

[43] Watson, D., "Under the Hood: Building and open-sourcing fbthrift", February 2014. https://code.facebook.com/posts/1468950976659943/under-the-hood-building-and-open-sourcing-fbthrift/.