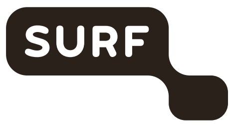


Technology Scouting 2019

Author(s): Ronald van der Pol

Version: 1.1

Date: 2020-02-19

Moreelsepark 48
3511 EP Utrecht
The Netherlands

PO Box 19035
3501 DA Utrecht
The Netherlands

+31 (0) 88 - 787 30 00
info@surf.nl
www.surf.nl

Contents

1 Introduction ... 3

2 IPv6 Segment Routing ... 4

2.1 SRv6 Packet Processing ... 5

2.2 SRv6 Network Programming ... 6

2.3 Segment Routing Mapped to IPv6 ... 7

2.4 Impact on the SURF Community ... 8

3 The QUIC Transport Protocol and HTTP/3 ... 9

3.1 QUIC Implementations ... 10

3.2 Impact on the SURF Community ... 11

4 The End of Moore's Law ... 12

4.1 The Shannon Limit ... 13

4.2 Limits To ASIC Serial I/O ... 14

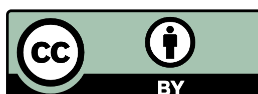
4.3 Impact on the SURF Community ... 14

5 Network Programming with XDP and BPF ... 15

5.1 Impact on the SURF Community ... 16

6 Domain-Specific Hardware and Domain-Specific Software ... 17

6.1 Impact on the SURF Community ... 18



This publication is licensed under a Creative Commons Attribution 4.0 International Licence
More information on the licence can be found on http://creativecommons.org/licenses/by/4.0/

1 Introduction

We constantly look for new trends and emerging technologies in the network area. We call this *scouting for new technology*. This is done in various ways: we look at what other ISPs are doing, we look at new protocols being designed in standard organisations, we follow open source project in the area of networking, we follow mailing lists, blogs, webinars, recordings of conferences, etc. And in our Next Generation Networking (NGN) Lab we familiarise ourselves with these new technologies by getting hands-on experience with them. This report is the short summary of some of the most interesting trends and technologies at end of 2019. We think that these have a potential use in our network or the networks of our connected institutes. The technologies are described at a high level. The author of this report can be contacted for a more detailed discussion.

2 IPv6 Segment Routing

Segment routing is a form of source routing in which a node that originates a packet (the headend node), adds an ordered list of segments (or instructions) to that packet. Such an ordered list is called a segment routing (SR) policy. These segments tell the routers along the path how that packet should be forwarded. Only the headend node to a segment routing domain maintains per flow state. There is no per flow state inside the network.

Segments have an identifier, the Segment Identifier (SID). Segments can be topological or service based. Examples of topological segments are:

- Node Segment with a prefix SID: these correspond to the address of a node and are used to steer traffic via certain nodes.
- Adjacency Segment with an adjacency SID: these correspond to interfaces (links) of a node and are used to steer traffic via certain links.

Examples of service segments are:

- A segment that steers a packet to a container or VM to be processed in some way.
- A segment that causes a certain QoS treatment to that packet.

The segment routing architecture is described in RFC 8402. It describes two data planes for segment routing: MPLS based (SR-MPLS) and IPv6 based (SRv6). In SR-MPLS (RFC 8660) the segment list corresponds to an MPLS stack and each segment is encoded as an MPLS label. SR-MPLS is being rolled out in the SURFnet8 network. In SRv6 the segment list is encoded in an extension header, the Segment Routing Header. Each segment is encoded as a 128-bit IPv6 address. An IPv6 address needs to be explicitly configured to be used as a SID. The remainder of this chapter describes IPv6 Segment Routing in more detail.

In SRv6, a node that acts as the headend of a segment routing domain encapsulates the original packet with an outer IPv6 header that contains a Segment Routing Header (SRH). The node adds segments to the SRH and by doing so determines how the packet is routed through the network and which services will act on the packet. The IPv6 Segment Routing Header is being defined in the IETF 6MAN working group as draft-ietf-6man-segment-routing-header and is show in Figure 1.

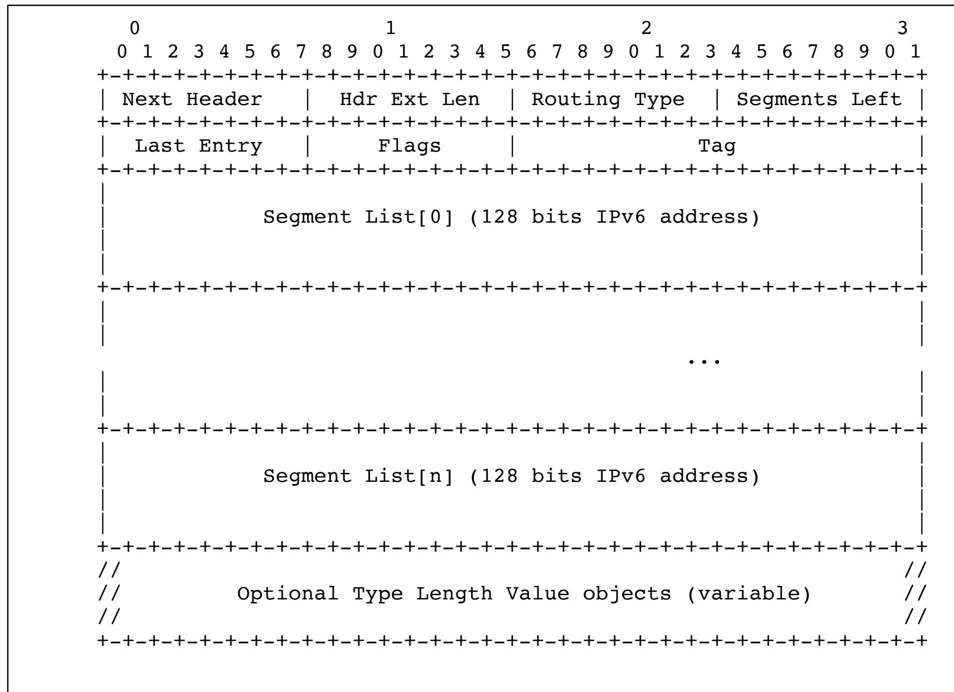


Figure 1: IPv6 Segment Routing Header

The "Segment List" is encoded in this header as an ordered list of IPv6 addresses. The currently active segment is always copied to the IPv6 destination address of the outer header of the packet. A "Segments Left" field in the SRH points to the next segment.

2.1 SRv6 Packet Processing

The headend node of a segment routing domain adds the encapsulating IPv6 header and the SRH to the packet, copies the active segment to the destination address, and sends the packet to that address according to its FIB. A node along a segment routing path examines the destination address. If the node does not have a segment that corresponds to that address, the packet is forwarded as a normal IPv6 packet. If the address corresponds to a segment of that node, an appropriate action is taken, the segment (address) pointed to by the "Segments Left" field is copied to the destination address field, and the "Segments Left" field is decremented. If a segment routing enabled node receives a packet in which the "Segments Left" field is zero, the packet is just forwarded as a normal IPv6 packet. Figure 2 shows the pseudo code that describes the forwarding process of a node.

```
S01. When an SRH is processed {
S02.   If Segments Left is equal to zero {
S03.     Proceed to process the next header in the packet,
         whose type is identified by the Next Header field in
         the Routing header.
S04.   }
S05.   Else {
S06.     If local configuration requires TLV processing {
S07.       Perform TLV processing (see TLV Processing)
S08.     }
S09.     max_last_entry  =  ( Hdr Ext Len /  2 ) - 1
S10.     If  ((Last Entry > max_last_entry) or
S11.          (Segments Left is greater than (Last Entry+1)) {
S12.       Send an ICMP Parameter Problem, Code 0, message to
           the Source Address, pointing to the Segments Left
           field, and discard the packet.
S13.     }
S14.     Else {
S15.       Decrement Segments Left by 1.
S16.       Copy Segment List[Segments Left] from the SRH to the
           destination address of the IPv6 header.
S17.       If the IPv6 Hop Limit is less than or equal to 1 {
S18.         Send an ICMP Time Exceeded -- Hop Limit Exceeded in
             Transit message to the Source Address and discard
             the packet.
S19.       }
S20.       Else {
S21.         Decrement the Hop Limit by 1
S22.         Resubmit the packet to the IPv6 module for transmission
             to the new destination.
S23.       }
S24.     }
S25.   }
S26. }
```

Figure 2: SRv6 Forwarding Process

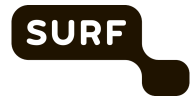
2.2 SRv6 Network Programming

More elaborate forwarding behaviour is being defined in the SPRING working group draft: draft-ietf-spring-srv6-network-programming. Examples are the decapsulation of the outer IPv6 and SRH headers and:

- Forward the packet as IPv6 packet
- Forward the packet as IPv4 packet
- Forward the packet to an L2 interface
- Forward the packet to an Ethernet VLAN

There is also the concept of a “Binding SID” for inter domain stitching. The Binding SID can be used for traffic-engineering policies across multiple domains. In such a case, an SRv6 headend node can include one or more binding SIDs in its segment list. Binding SIDs are used to forward a packet to a next domain and tell that next domain which SR policy it needs to apply to the packet. The headend node chooses one of the binding SIDs that correspond to one of the SR policies that the next domain supports. If domains further along the path also support SRv6 and binding SIDs, the headend node can choose to include their binding SIDs in the segment list too.

A domain that supports binding SIDs and receives a SRv6 packet from another domain does the following. Each binding SID is associated with an SR policy B and an IPv6 source address A. The node that applies the policy encapsulates the packet with an IPv6 header and an SRH. It fills the segment list in its SRH with segments



corresponding to the SR policy. The IPv6 sources address of the outer IPv6 header is set to A. Figure 3 shows the pseudo code for this handling of binding SIDs.

```
S01. When an SRH is processed {
S02.   If (Segments Left == 0) {
S03.      Send an ICMP Parameter Problem message to the Source Address
             Code 4 (SR Upper-layer Header Error),
             Pointer set to the offset of the upper-layer header.
             Interrupt packet processing and discard the packet.
S04.   }
S05.   If (IPv6 Hop Limit <= 1) {
S06.       Send an ICMP Time Exceeded message to the Source Address,
             Code 0 (Hop limit exceeded in transit),
             Interrupt packet processing and discard the packet.
S07.   }
S08.   max_LE = (Hdr Ext Len / 2) - 1
S09.   If ((Last Entry > max_LE) or (Segments Left > (Last Entry+1)) {
S10.      Send an ICMP Parameter Problem to the Source Address,
             Code 0 (Erroneous header field encountered),
             Pointer set to the Segments Left field.
             Interrupt packet processing and discard the packet.
S11.   }
S12.   Decrement Hop Limit by 1
S13.   Decrement Segments Left by 1
S14.   Push a new IPv6 header with its own SRH containing B
S15.   Set the outer IPv6 SA to A
S16.   Set the outer IPv6 DA to the first SID of B
S17.   Set the outer PayloadLength, Traffic Class, FlowLabel and
          Next-Header fields
S18.   Submit the packet to the egress IPv6 FIB lookup and
          transmission to the new destination
S19. }
```

Figure 3: Handling of a Binding SID

2.3 Segment Routing Mapped to IPv6

Segments in SRv6 are 128-bit IPv6 addresses. When there is a large segment list the overhead of the SRH with respect to the payload can become big. Some, including Juniper, have a problem with that and have come up with an alternative architecture: segment routing mapped to IPv6 (SRm6). Proponents of SRm6 stress that it is optimised for ASIC-based forwarding at high data rates. In January 2020 there is still a heated debate between proponents of SRv6 and those of SRm6. SRm6 is defined in the (currently personal) draft: draft-bonica-spring-sr-mapped-six. Ron Bonica is a Juniper employee.

SRm6 uses a different routing header than is used for SRv6, the Compressed Routing Header (CRH). There are actually two flavours, CRH-16 with 16-bit length segments and CRH-32 with 32-bit length segments. These headers are defined in draft-bonica-6man-comp-rtg-hdr. Figure 4 and Figure 5 show what those headers look like.

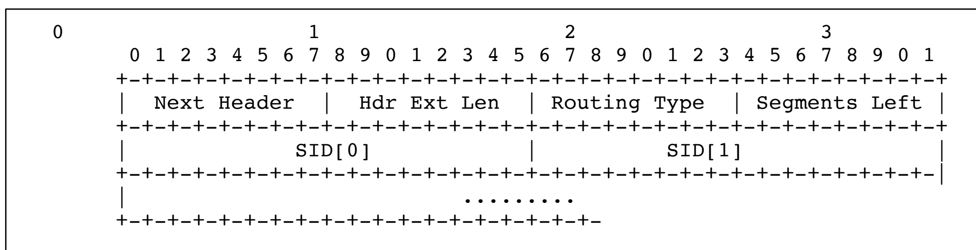


Figure 4: CRH-16

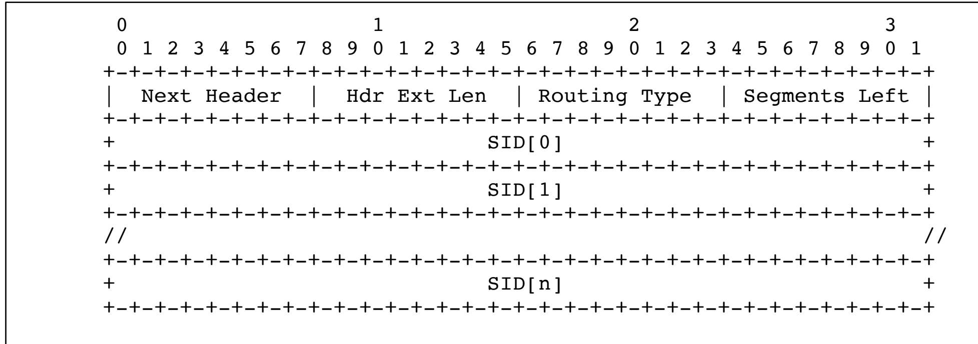


Figure 5: CRH-32

Compared to the SRH, there are no Last Entry, Flags, or Tag fields. The Last Entry is not needed in SRm6 because there is no variable option data. So, the Header Extension Length and the Routing Type determine how many SIDs are present. SRm6 defines three segment types (adjacency, node, and binding) and two service instruction types (per-segment and per-path). SRm6 nodes have an additional level of indirection compared to SRv6 nodes. A SID is not just an IPv6 address, but it is a lookup key in a table with additional information, such as segment type, and the IPv6 address to be used as destination address in the outer IPv6 header. For an adjacency segment type there is also an interface identifier. For a binding segment type there is a SID list length and a SID list.

2.4 Impact on the SURF Community

The SURFnet8 network uses Segment Routing over MPLS (SR-MPLS), but no MPLS labels are used between SURFnet8 and the customer network. Traffic from customers is mapped to SR-MPLS services, such as L3VPN. This is because most customer networks do not use MPLS. SRv6 might be easier to support in customer networks because SRv6 packets can be forwarded by plain IPv6 routers. So SRv6 can be deployed on select routers only. Moreover, SRv6 is supported in the Linux kernel (since 4.10). Therefore, a Linux host could send a SRv6 packet with a binding SID that sends that traffic to a certain segment routing policy offered by the SURFnet network. This could even be extended to services offered by other customers. We should start to discuss possible SRv6 uses cases with customers in order to help deciding whether to consider migrating to SRv6 or not at some point in the future.

3 The QUIC Transport Protocol and HTTP/3

QUIC started as a Google project to improve the HTTP protocol. It has been picked up by the IETF and the QUIC working group is standardising the protocol since 2016. This is also called IETF QUIC and is very different from the original Google QUIC. Google has migrated to IETF QUIC and Google QUIC is no longer used. By the way, QUIC is not an acronym.

IETF QUIC is a new transport protocol. The core transport protocol is described in draft-ietf-quic-transport. Initially, HTTP on top of QUIC is being standardised (draft-ietf-quic-http). But other protocols, like DNS, will follow. This chapter describes HTTP over QUIC, which is also called HTTP/3.

But let's first look at HTTP/2, which was an improvement over the original HTTP/1 and HTTP/1.1 protocols. HTTP/1.1 set up multiple parallel TCP sessions for each element of an HTML page. Especially on servers this caused problems with too many open file (socket) descriptors. Therefore, a browser typically set up a couple of TCP connections to a server. But this caused the problem of head of line blocking. Loading a web object from a server had to wait until one of the connections was finished with loading and available for loading another object. This was especially troublesome for large figures. Another problem was that connections were typically teared down and set up again during object loading. Therefore, the TCP connections were mostly running in the startup phase only and never reached full speed.

HTTP/2 on the other hand uses many parallel streams over each TCP connection. When the web page objects come from the same server, only one TCP connection needs to be set up and it can reach it optimal speed phase. It also solved the head of line blocking because of the multiple streams per connection. But there is still another problem. When a packet is lost, TCP has to slow down and retransmit. This impacts all parallel streams in that connection.

HTTP with TLS encryption (known as HTTPS) was developed in parallel. It can be used with both HTTP/1.1 and HTTP/2.

HTTP over QUIC (known as HTTP/3) has several improvements over HTTP/2. HTTP/3 replaces the HTTP/2 TLS and TCP layers. QUIC runs on top of UDP and (TLS 1.3) encryption is an integral part of QUIC (see Figure 6).

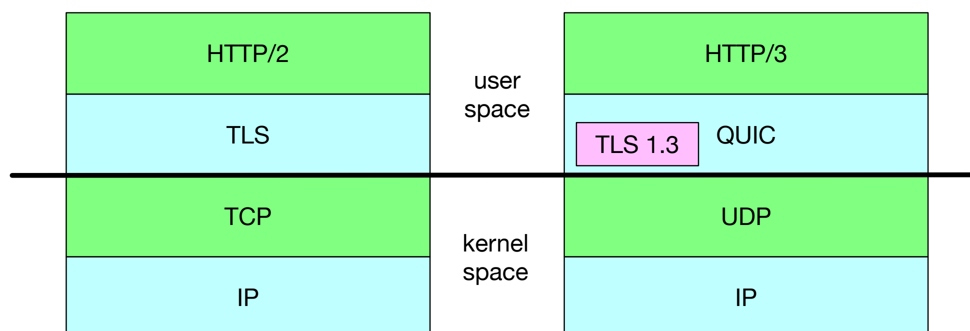


Figure 6: HTTP/2 versus HTTP/3

All current QUIC implementations are implemented in user space and they use UDP sockets. Congestion control is part of QUIC (so it also runs in user space). There are QUIC implementations in various languages (C++, C, Go, Rust, Python, Java), but there is no common API for applications. This means that application developers that choose a certain implementation are stuck with the API of that implementation. Another area of concern is congestion control. Because this is now running in user space, it is much easier to replace congestion control algorithms. With TCP, the congestion control algorithms are part of the kernel and changing them has a higher barrier because more people are looking at the code before it gets accepted.

HTTP/3 also supports multiple streams, but they are a service of the QUIC transport protocol. QUIC uses many logical flows within a single UDP session and thus prevents head of line blocking. These logical flows all have their own independent congestion control loop. A lost packet has impact on only one logical flow. Other applications that use QUIC can use that same stream service.

The QUIC handshake requires less round-trip times than the TLS+TCP handshake because in QUIC the transport handshake and the encryption handshake are integrated. QUIC not only encrypts the packet payload, but also much of the headers. QUIC uses two types of headers: long and short. The “long header” is used only during connection establishment. After the connection is established, QUIC uses the “short header” that only exposes a couple of flags, and a destination connection identifier. It can be as short as four bytes. The remainder of the packet is encrypted. Each endpoint of a connection has a connection identifier that can be used by its peer. Endpoint can also change their connection identifier during a session. This can be used by a client when moving (roaming) from one network to another. In each network a different identifier can be used. This protects a client from being tracked across networks. In this way QUIC offers the locator/identifier split paradigm where the identifier is the connection identifier and the locator is the IP address.

There are some challenges. Currently, the Linux UDP stack is not optimised yet (compared to the highly optimised TCP stack). UDP is slower and has a higher CPU load. Several parts of TCP processing can be offloaded to hardware on NICs. This is not the case for UDP yet. When QUIC takes off, work needs to be done to optimise the Linux UDP stack too. Networks also need to pay more attention to UDP. E.g. firewalls should not blindly block all UDP packets. This raises an interesting question about how HTTP/3 will be introduced. There are currently three proposed options. A browser can use any of them. The first is the “Alt-Svc” HTTP response header. It is used by the web server to tell the browser that the service is also available via QUIC on “host, protocol, port”. The second option is to try TCP port 443 and QUIC (probably on UDP port 443) at the same time. The third option is to use DNS. There is a new HTTPSSVC resource record that return the port number and IP address of HTTP/2 or HTTP/3 servers.

3.1 QUIC Implementations

Table 1: QUIC Implementations (source: IETF QUIC Working Group Wiki)

Name	Language	Role	License	Company
Aioquic	Python	Client, Server, Library	3-clause BSD	Independent
AppleQUIC	C, Objective-C	Client, Server	Closed	Apple
Ats	C++	Client, Server	Apache 2.0	Apache
F5	C	Client, Server	Closed	F5
Haskell Quic	Haskell	Client, Server, Library	3-clause BSD	Independent
Kwik	Java	Client	LGPL	Independent
LiteSpeed QUIC	C	Client, Server, Library	MIT	LiteSpeed Technologies
Microsoft QUIC	C	Client, Server	Closed	Microsoft
Mvfst (move fast)	C++	Client, Server, Library	MIT	Facebook
Neqo	Rust	Client, Server, Library	Apache2/MIT	Mozilla
Ngtcp2	C	Client, Server, Library	MIT	Independent
NGINX QUIC	C	Server	2-clause BSD	Cloudflare

Name	Language	Role	License	Company
Node.js QUIC	C++, Javascript	Client, Server	Node.js	Node.js
Pandora	C	Library	Not yet public	Aalto and TUM University
Picoquic	C	Library	MIT	Christian Huitema
Quant	C	Client, Server, Library	2-clause BSD	NetApp (Lars Eggert)
Quiche	Rust	Client, Server, Library	2-clause BSD	Cloudflare
QUICker	Typescript	Client, Server, Library	Unknown	Independent
Quicly	C	Client, Server	MIT	Fastly
Quincy	Java	Client, Server, Library	Unknown	Netty Project
Quinn	Rust	Client, Server, Library	Apache2/MIT	Independent
Sora_quic	Erlang/OTP	Server, Library	Unknown	Independent
Quic-go	Go	Client, Server, Library	MIT	Independent
Akamai QUIC	Unknown	Server	Unknown	Akamai

3.2 Impact on the SURF Community

Table 1 shows that there are many QUIC implementations. It is therefore safe to assume that the use of QUIC will start to rise once the QUIC RFCs are published. For infrastructure operators it is important to monitor the usage of QUIC and make sure that legitimate traffic flows uninterrupted, e.g. QUIC should not be blocked by firewalls. QUIC traffic can already be analysed with Wireshark. However, Wireshark must be told which UDP port is, e.g.

```
$ tshark –dudp.port==443,quic
```

Currently, the result of this is highly dependent on the version of QUIC on the wire and the version of QUIC that is used by your Wireshark version. When there is a version mismatch, you will get a “malformed packet” or similar warning. These versions correspond to the version of the QUIC transport IETF draft. But these version mismatches probably disappear as soon as QUIC is published as an RFC. There will be one QUIC version and this will be version defined in the RFC. Current estimates are that this will not happen until well into 2020.

4 The End of Moore's Law

For about half a century we have been living in a world where the speed of computers grew at an exponential rate. This is known as Moore's law, which is actually an observation of Gordon Moore that the number of transistors in an integrated circuit doubled approximately every two years. But today, that rate has almost levelled off. And Moore's law is not the only exponential that has come to its end. The same is true for Dennard scaling. In networking we are also hitting limits. We have reached the Shannon limit in optical communication. And in network ASIC design we reached the limit of serial bandwidth I/O.

For many decades the number of transistors on an integrated circuit grew exponentially. At the same time the power consumed by a transistor decreased, by lowering its operating voltage and current. As a consequence, the dissipated heat per transistor also decreased. Combined with Moore's law, it luckily resulted in an almost constant amount of heat dissipated by an integrated circuit during all these decades. This is called Dennard scaling.

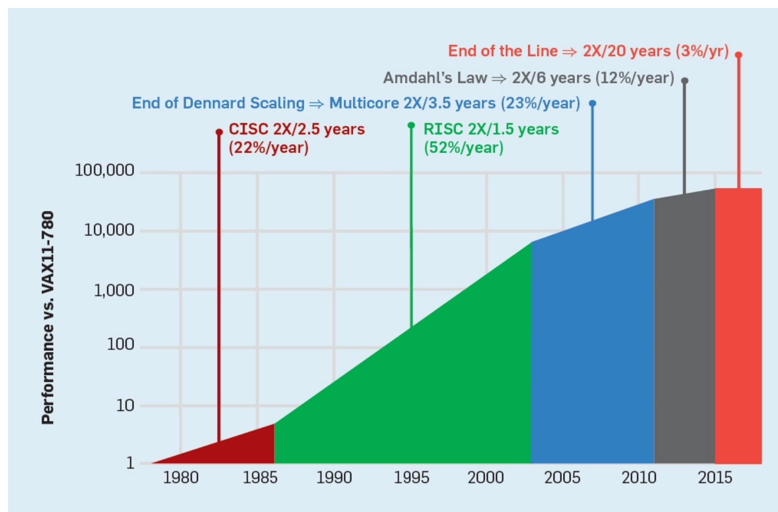


Figure 7: © 2019 CACM *, Vol 62 No. 2, A New Golden Age for Computer Architecture, J. Hennessy, D. Patterson*

However, as Figure 7 above shows, the era of exponential growth in performance has gone. Today improvements in CPU performance are very modest compared to what we were used to 20-30 years ago. We are also reaching physical limits. Transistors have become so small in size that manufacturing is getting very complex and expensive. Figure 8 shows that the number of transistors you can buy per Dollar is decreasing since several years.

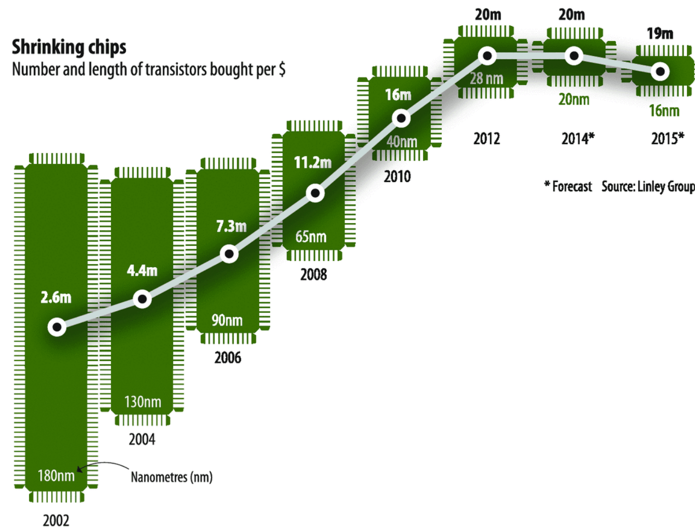


Figure 8: Number and length of transistors bought per Dollar (Linley Group)

Now that we have hit these scaling limitations, it is time to think about the way forward. Many see domain specific hardware and software as a solution (see chapter 6).

4.1 The Shannon Limit

In networking we are also running into boundaries. The Shannon limit imposes an upper limit to the amount of information you can send across a communication channel. This is dependent on the signal to noise ratio. Many see domain specific hardware as a way forward in ASIC evolution. The next chapter explores domain specific hardware and domain specific software.

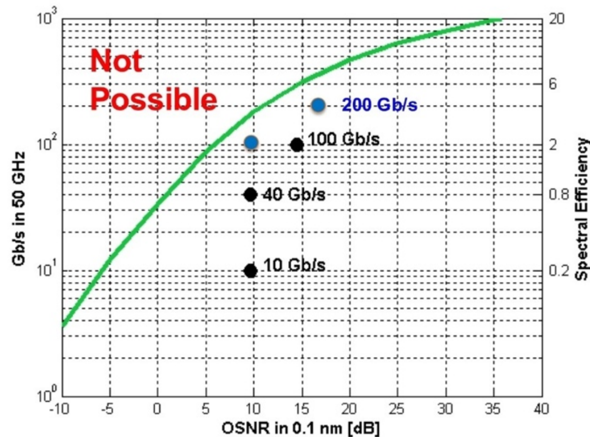


Figure 9: Optical transmission speeds in relation to Shannon's limit (source: Ciena)

Figure 9 shows that we have reached that limit in optical fiber. In practice this means that you have to make a choice: transmit your data at a lower speed over larger distances, or transmit your data at a higher speed over shorter distances.

4.2 Limits To ASIC Serial I/O

Modern high capacity networking ASICs have many I/O ports. Many popular data centre switches these days have 32 ports at 100 Gbps. These all end up on the ASIC, but not as 100 Gbps ports but as (4 times) 25 Gbps ports (SerDes). Therefore, these ASICs actually have 128 ports (4x32) which run an electrical signal at a rate of 28 Gbps. SerDes with a speed of 56 Gbps are starting to arrive. But the next step, 112 Gbps SerDes, is expected to be the last rate. After that manufacturing is getting too complex and expensive. This is because all these 128 ASIC ports need copper tracks on the circuit board to the front panel ports. Maintaining signal integrity of all those tracks is difficult and these electrical signals also have a large heat dissipation.

Silicon photonics could be a solution for the ASIC I/O challenges. A main problem is the length of copper tracks between ASIC and front panel ports. That length can be reduced by replacing part of the copper track by an optical link.

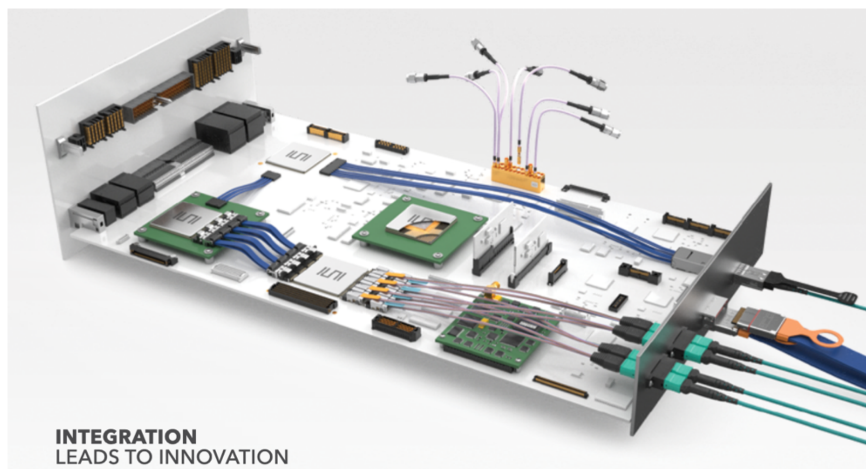


Figure 10: Optical link between ASIC and front panel port (source: Samtec)

Figure 10 shows a solution by Samtec. An electrical to optical chip is placed close to the network ASIC. Fiber connects that chip to optical front panel ports. Silicon photonics integrates optical lasers on the chip and lowers the cost of a solution like this.

4.3 Impact on the SURF Community

We are no more living in a luxury world where the CPU power doubles every year or two. This means that CPU power becomes a precious resource and it is important to optimise the tasks offered to the CPU. Programmers need to take more care of the efficiency of their algorithms. They also need to choose the best programming language for the job at hand. Sometimes, this will mean choosing a lower level language (like C, C++, or Rust) than Java or Python. Additionally, several new technologies can help too. The next chapters describe two of them.

In chapter 4.2 the replacing of copper wiring on the motherboard by more optical connections was described. This will probably be a gradual step by step process, driven by the equipment manufacturers. But this process should be monitored to understand what it means for operating such equipment. One possibility is that pluggables on front panel ports will be replaced by optical connectors and that the laser moves inside to the motherboard or ASIC. These lasers will probably have modest power output and short reach. In such a scenario a separate optical transmission system with long range lasers is needed. SURFnet8 already has this design.

5 Network Programming with XDP and BPF

The BPF framework can be used to load programs into the Linux kernel at runtime. It can be used for tracing and network programming. The BPF code runs in a virtual machine inside the kernel. BPF has a small (around 100 opcodes) RISC-like 64-bit Instruction Set Architecture (ISA). It uses 11 registers and a 512-byte stack. BPF programs can be written in various languages (C, Go, Rust, etc) and compiled with clang/LLVM to BPF bytecode.

There are several limitations to BPF programs. When the program is loaded into the kernel, a verifier checks if the program is safe. The verifier rejects programs when they have for example loops (unless they can be unrolled), out of bound memory access or jumps, calls to functions other than helper functions, or unreachable instructions. If the program passes the verifier, it can be JIT compiled to native instructions for the hardware.

BPF programs can be attached to various parts of the kernel. These programs can be divided in two major categories: programs used for kernel tracing and programs used for networking. This chapter focusses on BPF programs for networking that attach to the eXpress Data Path (XDP). XDP sits between the network driver and the TCP/IP stack and provides high performance packet processing. It is similar to DPDK (Data Plane Development Kit), except that DPDK runs in user space and XDP runs in kernel space (see Figure 11).

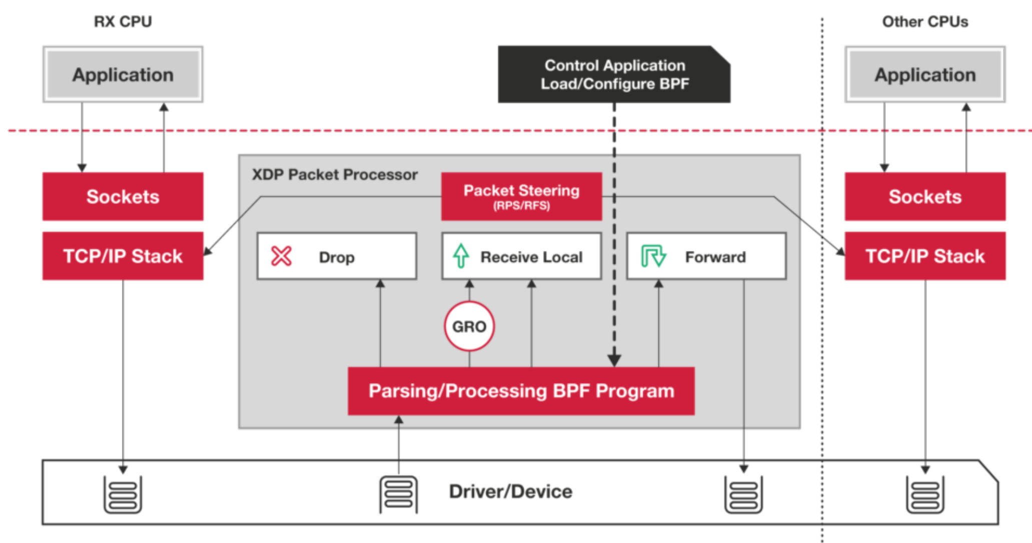


Figure 11: XDP Packet Processing (source: IO Visor Project)

DPDK has been around since 2010 and has become a very popular toolkit for high rate packet processing. XDP/BFP is newer (since around 2016) and tries to offer high rate packet processing inside the kernel. Network programming that uses XDP/BPF inside the kernel has the advantage of using the security features of the kernel, such as application isolation. With DPDK, the DPDK application needs to take care of that. Another advantage of XDP/BPF over DPDK is that XDP/BPF does not need dedicated cores. DPDK programs use one or more dedicated cores for packet processing. As such, XDP/BPF scales better when the packet load increases because more cores are added by the kernel scheduler to do the processing. But there are still interesting use cases for DPDK, such as software routers, like VPP (Vector Packet Processing), or high rate traffic generators, like TRex or pktgen. BPF and XDP, together with the BPF Compiler Collection (BCC), are being developed in the IO Visor Project (https://www.iovisor.org).

XDP/BPF programs have access to packet metadata that contains a pointer to the start and the end of a packet. The BPF program can change the packet (e.g. RTT decrement, rewrite MAC addresses, etc) and add or remove headers (e.g. VLAN pop or push). After processing the packet, the BPF program returns an action code. The currently defined codes are:

- XDP_ABORTED - program error, drop the packet
- XDP_DROP – drop the packet
- XDP_PASS – forward the packet to the network stack
- XDP_TX – forward the packet to the interface it arrived from
- XDP_REDIRECT – forward the packet to another interface

XDP/BPF has much potential in use cases that require network programming at the edge.

5.1 Impact on the SURF Community

XDP/BPF is an example of a technology that can reduce the load on the CPU. A good example is dropping unwanted traffic. When this traffic can be dropped at the XDP module, these packets do not need to be processed by the TCP/IP stack which takes a lot of CPU processing power. Tests have shown that this significantly reduces the load on the CPU, especially when XDP/BPF can be offloaded to a smart NIC. Work is already going on to replace iptables in Linux by XDP/BPF based filtering.

6 Domain-Specific Hardware and Domain-Specific Software

Now that we have reached the end of Moore's law, we need to find solutions. The 2017 Turing Award winners, John L. Hennessy and David A. Patterson, actually see a bright future for computer science. They called their Turing award lecture *"A New Golden Age for Computer Architecture"*[1]. In their lecture they seek the solution in new chip architectures and bringing various disciplines together. That should result in a close cooperation between processor designers, compiler architects and computer language designers, and security experts. A new ASIC architecture should have security built in from the start (security by design). They also expect domain-specific architectures. A general-purpose CPU architecture is too slow. We need to design domain-specific hardware and domain-specific software, tailored for the job at hand.

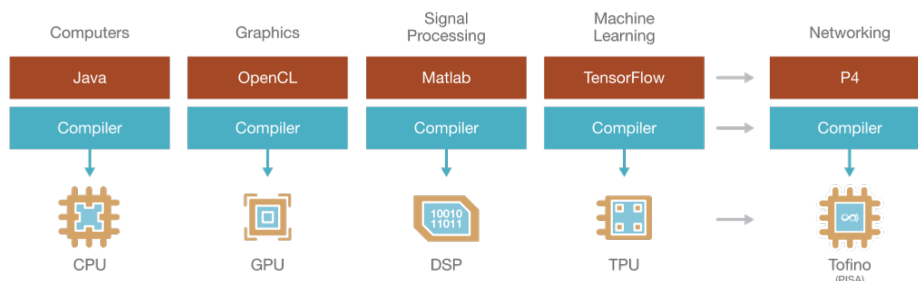


Figure 12: Domain Specific Hardware and Software (source: Barefoot Networks)

Figure 12 shows a couple of examples of existing domain-specific architectures. In the SURF NGN Lab we have been working with P4 in networking for several years. The NGN Lab has servers with Netronome smart NICs that can be programmed by either P4 or BPF (via hardware offload). There are also a couple of switches with a Tofino ASIC is P4 programmable. These are ASICs with 32 ports at 100 Gbps that can run at line rate.

In a P4 switch a packet is first parsed by a programmable parser and the packet is split in different protocol fields. Next it enters one or more match-action tables. P4 supports several match types, such as Longest Prefix Match (LPM), exact match, or ternary (TCAM). Actions define what to do with a packet, e.g. decrement the TTL, update the MAC layer addresses, and forward the packet to an outgoing interface. Figure 13 shows the packet flow inside a P4 switch, such as the Tofino ASIC.

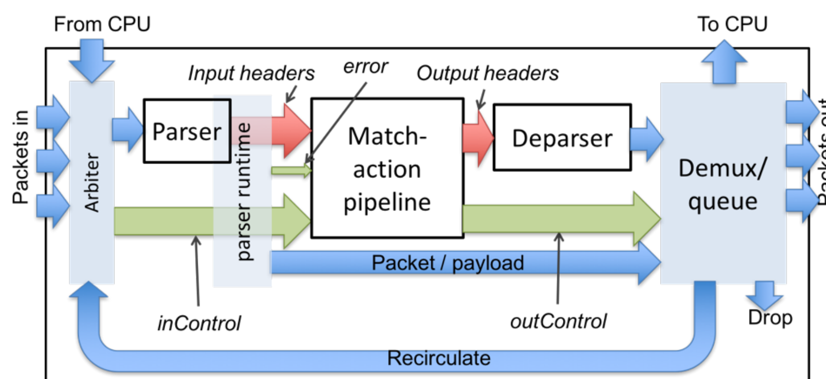


Figure 13: Packet Flow inside P4 Switch

[1] https://cacm.acm.org/magazines/2019/2/234352-a-new-golden-age-for-computer-architecture/fulltext

The programmable parts (parser and match-action tables) are defined by a P4 program. P4 has a C-like syntax and consists of three main parts:

- Protocol headers (fields and the size of those fields in bits)
- The parser
- The match-action tables

A P4 program is compiled and the binary is used to program the parser and match-action tables of the switch. The P4 compiler also produces a control plane API that can be used by e.g. a routing daemon.

The Netronome smart NICs in the NGN Lab can also be used for BPF offloading. Parts of the XDP/BPF program that runs in the Linux kernel can be offloaded to the NIC's hardware.

6.1 Impact on the SURF Community

In previous chapters the importance of reducing the load on the CPU was discussed. Offloading network functionality to a smart NIC or programmable switch is an example of that. The smart edge, edge computing, etc. are all part of this paradigm. For each use case one needs to consider which part is running on a general purpose CPU, which part is running as XDP/BPF (offloaded to a smart NIC) and which part is running on a P4 programmable switch. We will keep experimenting with proof of concepts in all these areas in the NGN Lab. The SURF community is called upon to discuss possible use cases with us.