What's New in the Linux Network Stack?

Lukas M. Märdian
Advisors: Paul Emmerich, Daniel Raumer
Seminar Future Internet, Winter Term 14/15
Chair for Network Architectures and Services
Department for Computer Science, Technische Universität München
Email: maerdian@in.tum.de

ABSTRACT
In this paper, interesting features of the Linux kernel's network stack are analyzed, which were introduced during the development cycles from Linux v3.7 to Linux v3.16. Special attention is given to the low-latency device polling, introduced in Linux v3.11, the netfilter's SYNPROXY target, introduced in Linux v3.12 and the new Nftables framework, introduced in Linux v3.13. At the end a trend is presented, which shows the direction in which the Linux network stack is evolving.

Keywords
Linux, network, packet processing, SYN proxy, JIT compiler, firewall, low-latency, Berkeley Packet Filter

1. INTRODUCTION
The Linux kernel is a fast moving and always changing piece of free software. Having a very mature network stack, Linux is deployed widely, especially to drive and manage the ever growing and changing world wide web. It is sometimes hard to follow the newest developments and discussions regarding the Linux kernel, so this paper gives an analysis of the current state of the Linux kernel's network stack and presents some interesting features, which were introduced in the development cycles from Linux v3.7 to v3.16.

Chapter 2 describes the low-latency device polling feature, which was introduced in Linux v3.11. Chapter 3 gives an overview of the addition of a SYN proxy to the Netfilter subsystem, introduced in Linux v3.12. The new packed filtering framework *Nftables*, successively replacing *Iptables*, is introduced in Chapter 4. In Chapter 5 the trend of the Linux kernel's network evolution is discussed and finally a conclusion is presented in Chapter 6.

2. LOW-LATENCY DEVICE POLLING
In the Linux kernel v3.11 Eliezer Tamir et al. introduced a low-latency polling mechanism for network devices. The classical way how the Linux kernel handles its network devices, by using the New API (NAPI), provides a good trade off between efficiency and latency of the packet processing. Still, some users with more specific demands, such as the finance sector with their high-frequency trading systems or scientific research with high-performance computing, need to reach the lowest latency possible, even on high traffic network devices. This demands are not possible to reach with NAPI. [3]

2.1 Interrupts vs. Polling
Usually, the Linux kernel handles network devices by using the so called New API (NAPI), which uses interrupt mitigation techniques, in order to reduce the overhead of context switches: On low traffic network devices everything works as expected, the CPU is interrupted whenever a new packet arrives at the network interface. This gives a low latency in the processing of arriving packets, but also introduces some overhead, because the CPU has to switch its context to process the interrupt handler. Therefore, if a certain amount of packets per second arrives at a specific network device, the NAPI switches to polling mode for that high traffic device. In polling mode the interrupts are disabled and the network stack polls the device in regular intervals. It can be expected that new packets arrive between two polls on a high traffic network interface. Thus, polling for new data is more efficient than having the CPU interrupted and switching its context on every arriving packet. Polling a network device does not provide the lowest packet processing latency, though, but is throughput optimized and runs with a foreseeable and uniform work load. [1]

2.2 Low-latency Polling
In order to make network packets reach the network stack and user space as fast as possible, the low-latency device polling mechanism was introduced, so users of the network stack (applications) can poll for new packets, whenever they are ready to process new data. Even though polling is involved, this technique provides a lower latency than reenabling the per-packet interrupts, because on a high traffic network device, there would be hundreds or thousands of packet-interrupts per second. Handling all of them would introduce a larger latency than polling within very small intervals. [2, 3]

Technically, this is realized by a new function call, named `ndo_busy_poll()`, defined in `include/linux/netdevice.h`, which can be implemented by network device drivers [4]. In this function call the device drivers are supposed to poll the network hardware for new packets and return them immediately. If no packets are available at the moment, the drivers are supposed to *busy wait* for new arriving packets until a timeout is reached, as defined (in μs) by `sysctl.net.core.busy_read` and `sysctl.net.core.busy_poll`. The default timeout value is set to 0, which means busy waiting is disabled and the driver should poll only once. For latency critical applications a polling timeout of 50μs is recommended. [2, 3, 7]

On devices, whose drivers implement this low-latency polling function call, e.g. Intel's *ixgbe* driver [5], the network stack will poll the hardware for new network data at certain situations, e.g. when it is instructed to do so by a user space application via the `poll()` system call. This way the user space can ask for new network packets, whenever it is ready to process new data. The driver will then either directly flush the new network packets, which arrived since the last poll, or (if no packet arrived) it will poll the network device in a busy waiting loop and flush new data, arriving before the timeout. Another situation where the network stack will issue the low-latency polling function call is in the `read()` system call: When a user space application tries to read new data from a socket but there are no packets in the queue, the network device will be polled in a busy waiting loop until the timeout is reached. On high traffic devices, the driver will most likely be able to return new data to the user space as a response to its `read()` call. [2, 3]

With this low-latency polling mechanism, an latency improvement of about 30% is possible (2.5μs within an UDP connection and 2.2μs within a TCP connection), introducing just a minimally increased CPU load on a typical system, using two Intel Xeon E5-2690 CPUs, X520 optical NICs and the *Netperf* benchmark (c.f. Figure 1). [7]

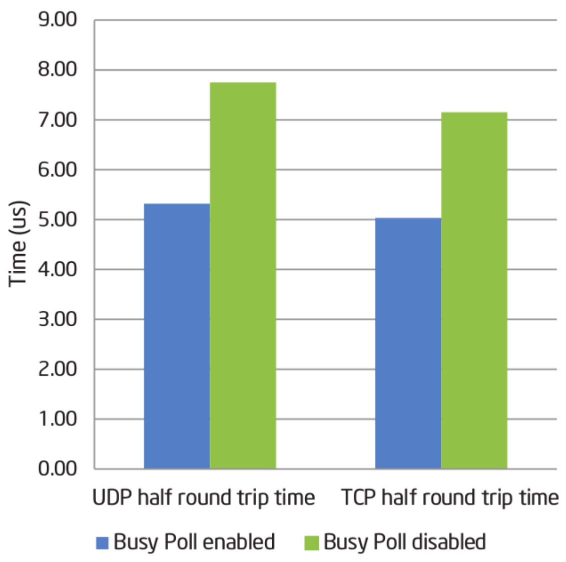

Figure 1: Netperf Latency results [7]

3. NETFILTER: SYNPROXY TARGET

Introduced in the Linux kernel v3.12 by Patrick McHardy et al. was a *SYNPROXY* target for the kernel's internal packet filtering framework, called *Netfilter*. This implements the concept of a stateful firewall, where connection attempts are filtered very early in the network stack, before they reach their destination. So they can be handled before they are tracked by a data structure, called *transmission control block* (TCB), in the filtering subsystem (*conntrack*).

3.1 SYN-Flooding Attacks

In order to establish a TCP connection between a server and a client, a three way handshake is used: The client sends a SYN packet to the server to request a connection, then the server responds with a SYN/ACK packet to confirm the client's request. If everything is fine, the client answers with an ACK packet and the connection is established. This TCP three way handshake can be attacked by a SYN flooding attack, which is a commonly used Distributed Denial-of-Service (DDoS) attack. In this attack, the attacker sends many SYN requests with spoofed IPs to the server, using a botnet. For each of those faked requests the server needs to initialize a TCB data structure, respond with a SYN/ACK packet and wait for an ACK from the client. In case of a DDoS attack the ACKs will never arrive, as the senders IP addresses are not real. Still, the server needs to store the TCBs for a certain amount of time, which will fill up its TCB buffer and leads to a situation where real (i.e. non-attack) requests cannot be handled anymore and thus the server is rendered unreachable. [8, 9] Reducing the timeout of the transmission control blocks, which is 5 seconds in the default case (initial retransmission timeout (1 sec) * `tcp_synack_retries` (5), c.f. `man tcp(7)`, [10]), or increasing the TCB buffer will help to handle such situations. Depending on the attacker's botnet size, the server will be out of service anyway, though. In order to defend such DDoS attacks, countermeasures, such as SYN cookies and SYN proxies have been invented, which are presented in the following.

3.2 SYN-Cookies

One countermeasure to defend a SYN flooding attack is the use of so called SYN cookies. The modern type of SYN cookies was introduced in 1996 by D. J. Bernstein et al. [11]. The idea is to compute the cryptographic hash of the characteristic data of the connection, like the sender's and receiver's IP addresses and the ports used. This data is joined with a secret key, which is only known to the server, using a hashing algorithm, such as MD5 or SHA. The resulting hash is called SYN cookie and is set to be the sequence number (SQN) of the SYN/ACK TCP packet. If the SYN/ACK reaches a real client, which responds with a real ACK message, the cookie can be reconstructed from that message's SQN field. Thus the server does not need to store a TCB for each SYN request, but instead just needs to check the cookie of arriving ACK packets. If the SYN cookie contained in the SQN field of the arrived message is the same as the hash, which the server can reconstruct from the secret key and the connection's characteristics, the server can be sure that he send a SYN/ACK message to that client before. Using this method, SYN flooding can be circumvented, as there is no TCB queue, which could be flooded. There are some drawbacks in using SYN cookies, though. For example the server needs to have enough computing power to compute SYN cookies for all arriving SYN packets in real time and the firewall needs to know the server's TCP options, to determine the connection characteristics. [9]

3.3 SYN-Proxies

A SYN proxy is an entity in the same network as the server, which ought to be protected. Its purpose is to filter and/or load balance incoming SYN requests, using different methods and technologies (e.g. SYN cookies). A common method

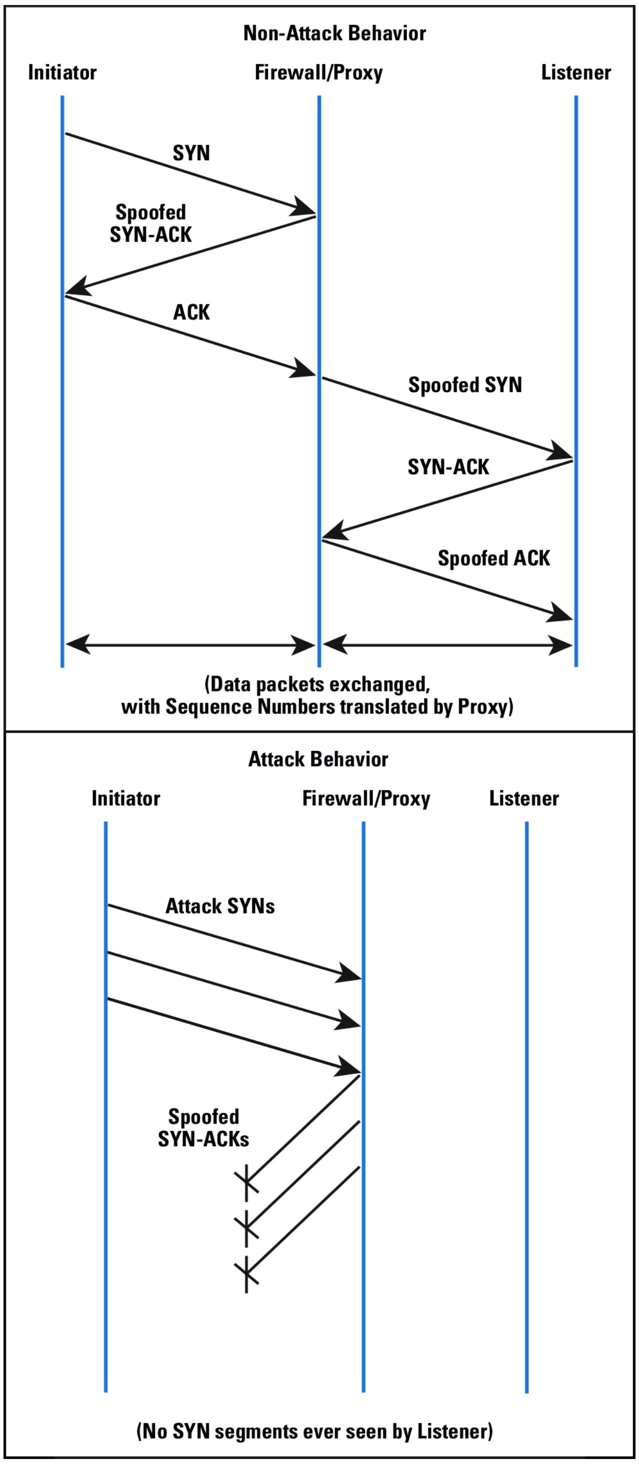

Figure 2: SYN-ACK spoofing Proxy [14]

of such a stateful firewall is the spoofing of SYN/ACK packets to the client (cf. Figure 2), in order to avoid invalid connection attempts to reach the real server and open a connection, including a TCB. [14, 12]

In Linux v3.12 and beyond the new SYNPROXY target can be setup as shown in the following:

1. Don't track incoming SYN requests:
   ```
   iptables -t raw -I PREROUTING -i $DEV -p tcp \
   -m tcp --syn --dport $PORT -j CT --notrack
   ```
2. Mark unknown ACK packets as invalid:
   ```
   /sbin/sysctl -w \
   net/netfilter/nf_conntrack_tcp_loose=0
   ```
3. Handle untracked (SYN) and invalid (ACK) packets, using the SYN proxy:
   ```
   iptables -A INPUT -i $DEV -p tcp -m tcp \
   --dport $PORT -m state --state \
   INVALID,UNTRACKED -j SYNPROXY --sack-perm \
   --timestamp --wscale 7 --mss 1460
   ```
4. Drop the remaining invalid (SYN/ACK) packets:
   ```
   iptables -A INPUT -i $DEV -p tcp -m tcp \
   --dport $PORT -m state --state INVALID -j DROP
   ```
5. Enable TCP timestamps, needed for SYN cookies:
   ```
   /sbin/sysctl -w net/ipv4/tcp_timestamps=1
   ```

Using this setup and a little bit of *conntrack* tuning, it was shown, that the performance of a server during a SYN flooding attack can be increased by the order of one magnitude, using this *Netfilter* SYNPROXY target [12]. This is a nice improvement, but of course using this stateful firewall comes at a cost: Due to the extra layer, which the SYN proxy introduces, the connection establishment phase will take longer, leading to an increased latency. Also TCP connection parameters and characteristics, used by the SYN proxy, must match the servers TCP settings. These settings have to be set in the SYNPROXY module manually, leading to a configuration overhead and a new source of errors. To fix the configuration problem, the authors have already proposed a solution where the connection characteristics can be detected automatically, by forwarding the first connection request for normal processing to the server and sniffing the settings from its response [13].

4. NFTABLES

Nftables is a dynamic firewall system, which is based upon a bytecode interpreter. The project was initially started in 2009 by Patrick McHardy but discontinued, due to a lack of interest. In October 2012 the Netfilter maintainer Pablo Neira Ayuso announced to revitalize Nftables, based upon McHardy's work. With his additional ideas he could attract more developers and together they were able to integrate Nftables as a new feature in Linux v3.13. [15]

4.1 Iptables, Ip6tables, Arptables, Ebtables

Prior to the general purpose firewall system *Nftables* several other, protocol specific solutions existed in the Linux kernel, namely *Iptables* to filter IPv4 connections, *Ip6tables* to filter IPv6 connections, *Arptables* to filter ARP connections and *Ebtables* to filter ethernet bridging connections. Those static solutions had some drawbacks, such as tightly coupled data structures, which got passed back and forth between the kernel and user space. This made the kernel's implementation of those filters quite inflexible and optimizations to the structures very hard to implement. Furthermore, the ruleset for the filters was represented as one big chunk of binary data, which made it impossible to incrementally add new rules to the firewall system. The solution to this problem was, that the firewall management tools dumped the firewall's current state to a file, modified the whole chunk of data (added, removed, altered rules) and injected the file as a whole back into the firewall system. This approach works but is problematic, because it has a quadratic complexity for incremental changes. Also, by injecting the new rule set, the firewall will loose its current state, which makes it unable to continue tracking the currently open connections, so it has to start over. [17]

4.2 Virtual Machine

The concept behind the dynamic *Nftables* system, which helps to overcome the above mentioned problems of static and protocol specific firewall systems, are based upon a virtual machine. All the static filtering modules, used by *Iptables* and the other protocol filters, are replaced by a single kernel module, providing "a simple virtual machine [...] that is able to execute bytecode to inspect a network packet and make decisions on how that packet should be handled" [15]. The idea of this approach is inspired by the *Berkeley Packet Filter* (BPF) virtual machine. The virtual machine on its own is not able to do any packet filtering, instead it is dependent on small bytecode programs, which describe how a specific package should be handled. Those small, individual programs get compiled by Nftables' corresponding management tools in the user space, e.g. the command-line utility `nft`, and can incrementally be put into the interpreter at runtime. [16]

One of the biggest benefits of this virtual machine approach is a massive reduction in complexity. By replacing four filtering systems for different protocols with a single, universal system, a lot of code can be removed. This leads to less problems, because duplicated code is removed and the remaining common is reviewed by more people. By moving the filtering logic out of the Linux kernel into the firewall management tools, while just keeping a small virtual machine to execute the bytecode, provided by the management tools, the complexity of the filtering logic is reduced as well.

In order to stay compatible with the old firewall management tools, such as `iptables` and `ip6tables`, Nftables provides a compatibility layer, which is integrated in the Iptables project and enables long time users of Iptables to easily migrate their old firewall rules (using old syntax). Internally the new bytecode for the Netfilter virtual machine will be compiled. However, new users are encouraged to use the new `nft` command-line utility to manage their firewall in Linux kernels of version v3.13 or beyond. [16, 15]

4.3 New possibilities of Bytecode filtering

In contrast to the old firewall system, the new dynamic system, using its small bytecode programs, is much more flexible and provides quite some improvements: Each rule change, which is represented as a small bytecode program, can be performed atomically during the runtime of the system and without interfering with the general state of the firewall. Also, open connections can be kept open, if not requested differently be the new rules. This atomic replacement of rules does speed up changes in firewalls with big rule sets quite a bit, as it is not needed anymore to dump, modify and replay the whole state of the system, but instead just the single, wanted modification can be executed. In addition to the speedup it also helps to avoid race conditions, which could occur during the rule set change in the old system. [15]

Another benefit is, that new matching types (i.e. characteristics of a packet to filter for) can be added easily to a bytecode program. Whereas the old system used to depend on an extra kernel module for each matching rule, which could be set by the management tools. This led to a large amount of over 100 modules. Getting new matches/modules into the kernel took much longer than writing a simple program, which can be injected into the firewall at runtime. Furthermore, handling all the matching in small programs, compiled by user space tools, makes it possible to have the rule set optimized. Using generic compiler optimizations, faster execution in the in-kernel virtual machine can be achieved, e.g. by automatically removing duplicated or unreachable rules, which the firewall administrator did not think of. [17]

Using the `libnftnl` library, provided by the Nftables project, it is easily possible to write new and improved firewall management tools, too. This library enables those user space tools also to listen to changes in the firewall system and notify the applications in real time about the current state. [17] One tool leveraging this library is `nft`, the main firewall management tool, provided by the Nftables project. It has a similar functionality as the old `iptables` tool but tries to be more intuitive, by using natural language to describe the different filters instead of lots of configuration switches. For example if somebody wants to drop all IPv4 HTTP traffic, this can be established as follows: [18]

1. Setup an iptables like chain, using the ipv4-filter file, provided by Nftables:
   ```
   nft -f files/nftables/ipv4-filter
   ```
2. Drop all incoming TCP packets with the destination of port 80 (HTTP):
   ```
   nft add rule ip filter input tcp dport 80 drop
   ```

5. LINUX NETWORKING TREND

In this final chapter, a general trend is shown, which can be observed in the development of the Linux kernel's network stack: It can be seen that the kernel developers introduce more and more dynamic features such as virtual machines, bytecode interpreters and JIT-compilers into the kernel, which help to abstract certain features and move the complexity into the userspace.

5.1 Berkeley Packet Filter VM

The Berkeley Packet Filter (BPF) has for very long been part of the Linux kernel. It is a tool, which can filter network packets on a low level, in order to function as a per-application firewall, forward only relevant packets to the user space and allow the tracing of network traffic, using tools such as `tcpdump` [19]. Simple examples of the filter's internal workings can be found in the original paper by McCanne et al. 1993, e.g. a small BPF program, which loads a packet's Ethernet protocol type field at offset 12 and accepts the packet if it is of type *IP* or rejects it otherwise:

Listing 1: BPF program: IP filter [23]

```
    ldh     [12]
    jeq     #ETHERTYPE_IP , L1 , L2
L1: ret     #TRUE
L2: ret     #0
```

Since the release of Linux kernel v3.0, the BPF has been improved continuously. It started with the introduction of a just-in-time (JIT) compiler for the filter, by Eric Dumazet, which enabled the Linux kernel to translate the virtual machine instructions to assembly code on the x86_64 architecture in real time and continued with other performance improvements and functional extensions in Linux kernel v3.15 and v3.16. [19, 20]

In recent Linux kernel versions (v3.15, v3.16), the BPF virtual machine has been extended from having a very simple and network specific architecture with just a few registers and capabilities, to being more of a general purpose filtering system, whose capabilities can be mapped pretty close to modern hardware. The instructions of this virtual machine, as can be seen in Listing 1, are usually mapped 1:1 to real assembly instructions of the underlying hardware architecture. [19, 21]

5.2 Dynamic Firewall Systems

The concept of dynamic filtering systems on different levels of abstraction is another development trend in the Linux kernel. As discussed before, there is Nftables, which has its filtering rules created and optimized dynamically at user space level, using tool such as `nft`. These rules, which implement the filters, can then be fed into the firewall system dynamically at runtime and are being processed by a virtual machine in the kernel, in a dynamic manner (c.f. Chapter 4).

In addition to the general purpose firewall *Nftables*, which aims to protect a Linux system from the outside world and regulate the incoming and outgoing network traffic, there is the *Berkeley Packet Filter* (BPF), too. It can function as a per-application firewall and can be used for system internal packet processing, e.g. to reduce the network traffic, which is directed to a specific application. The BPF is implemented as a dynamic bytecode interpreter, too. (cf. chapter 5.1)

This evolution from static tools, such as `iptables` or the early, static variant of the *Berkeley Packet Filter*, to more dynamic and abstract tools, is characteristic for the direction of development in the current versions of the Linux kernel. A remaining question is, why those quite similar tools are still separated and not merged into one universal solution for dynamic packet filtering. This is because the tools come from different backgrounds and as of today none of them is yet fully able to replace all the others. McHardy explained, why they did not build *Nftables* upon an existing solution, such as the BPF virtual machine: "A very important feature, one that is missing from all other filters that are built similar in the kernel (like BPF, TC u32 filter, ...), is reconstruction of high level constructs from the representation within the kernel. TC u32 for example allows you to specify 'ip daddr X', but when dumping the filter rules it will just display an offset and length." [22]

As the abstraction level continues to rise, and the firewall systems are not just used for packet filtering any more, but also for general Linux kernel tracing, it is plausible that the different virtual machines in the Linux kernel are combined into a single, general purpose interpreter in the future.

5.3 Future possibilities

Watching the current trend of developments in the Linux kernel, a little outlook how the kernel's network stack might evolve in the future is presented now: The inclusion of dynamic bytecode interpreters into the Linux kernel is a popular way to increase the level of abstraction in kernel development and also to reduce the complexity. Right now, there are several separate virtual machines in the kernel, which is not an optimal solution, because some common code needs to be implemented in the same (or minimally different) ways in all of the implementations. That is also why the kernel developers rejected the inclusion of yet another virtual machine named *Ktap* (used for kernel tracing) in favor of the BPF's tracing capabilities. [24]

In general, the BPF is a very well known virtual machine, as it is in the kernel for a very long time already and got considerable performance and functional improvements in the recent Linux kernel releases. With its split into the *classic BPF* variant, providing a legacy interface to the classic BPF network filtering functionality, and the new *internal BPF* variant, which is a generalized virtual machine, including a JIT compiler. This JIT compiler has seen various performance improvements, by optimizing its commands to use architecture specific assembly code. In the future such optimizations are likely to be continued, especially for newer architectures, such as ARM64. [21]

With the Berkeley Packet Filter's architecture, it is already possible to use it for network packet filtering and also for more general tasks, such as syscall tracing, as used by the Linux kernel's *secure computing* (seccomp) system, similar to what *Ktap* wanted to achieve [21]. Some functionality is still missing in the BPF, though. For example *Ktap* wanted to enable the possibility to use filters, supplied by user space applications and so does *Nftables* with its rule sets, generated in user space, too. Another thing, which needs to be implemented in the BPF, is the possibility to reconstruct high level data structures, i.e. reverse the optimizations, which have been done by the JIT compiler, as this is one of the reasons why *Nftables* was not build upon the BPF (c.f. Chapter 5.2).

If those and potentially some other drawbacks would be improved, the Berkeley Packet Filter could become the single, general purpose and highly optimized virtual machine in the Linux kernel and all the other tracking, tracing and filtering systems could build upon it. This would be a large benefit to Linux developers and users, as the sharing of code leads to less potential problems and a single spot to apply optimizations in, which will in turn benefit all systems.

6. CONCLUSION

In conclusion, the developments in the Linux kernel's network stack keep pace with the fast and always changing Internet. This global network has very versatile demands, which the Linux kernel tries to adopt to, be it from a security, performance or complexity perspective.

With the addition of a low-latency device polling mechanism in Linux v3.11, the kernel can now easier be used in scenarios where very low network latencies are critical, such as high performance computing or high frequency trading. Before the inclusion of this feature, the companies who needed low latencies usually implemented their own low-latency network stack in the user space, in order to bypass the kernel. With the low-latency device polling in place, those companies can now work together with the Linux community, to continuously improve the low-latency network stack in a single place, saving a lot of development resources. Benchmarks by Intel [7] show an improvement in latency of about 30% for the TCP and UDP protocols and a use case where lots of small network packets with low-latency demands are send.

In terms of security, the Linux kernel was improved by the introduction of stateful and dynamic firewall additions. The SYN proxy, introduced in Linux v3.12, adds an extra stage to the firewall, where new network connection attempts are tracked, using SYN cookies. Only if the connection was successful, they get forwarded to the rest of the network stack, which reduces the overhead in case of a SYN flooding DDoS attack and enables a server to handle ten times more packets, with just a small increase in packet latency [12]. The *Nftables* dynamic firewall system on the other hand, introduced in the Linux kernel v3.13, improves the kernel's network stack with the new possibilities of bytecode filtering, where the filters are not statically coded into kernel modules, but rather the rules are compiled and optimized to small bytecode programs in user space. Those small programs are then executed in an in-kernel virtual machine at runtime. This way the management of the firewall system is much more flexible and can dynamically adopt to the changing demands of the Internet.

All in all the developments evolve into a direction, where more systems can be controlled from outside the kernel, by user space applications. Also, the controlling can be done in a dynamic manner, be it by the activation of a low-latency path for network packets, the redirection of network traffic to a SYN proxy or the execution of bytecode programs. In the future, the different packet processing systems in the Linux kernel might be merged into a single, abstract system, which is able to handle lots of tasks by executing code, provided by the user space in a dynamic manner. Such a system would not only be able to handle network related tasks, but could also deal with other tracking, tracing and filtering tasks.

7. REFERENCES

[1] J. Hadi Salim, R. Olsson, A. Kuznetsov: *Beyond Softnet*, In Proceedings of the 5th Annual Linux Showcase & Conference, pages 165-172, 2001

[2] J. Corbet: *Low-latency Ethernet device polling*, In Linux Weekly News, Eklektix Inc., May 2013, `https://lwn.net/Articles/551284/`

[3] J. Brandeburg: *A way towards Lower Latency and Jitter*, At Linux Plumbers Conference, August 2012

[4] E. Tamir: *net: add low latency socket poll*, June 2013, `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=060212928670593`

[5] E. Tamir: *ixgbe: add support for ndo_ll_poll*, June 2013, `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=5a85e737f30c`

[6] E. Tamir: *net: low latency Ethernet device polling*, May 2013, `https://lkml.org/lkml/2013/5/19/20`

[7] J. Cummings, E. Tamir: *Open Source Kernel Enhancements for Low Latency Sockets using Busy Poll*, October 2013

[8] W. M. Eddy: *RFC-4987: TCP SYN Flooding Attacks and Common Mitigations*, August 2007, `http://www.ietf.org/rfc/rfc4987.txt`

[9] J. Bongertz: *Tatort Internet: Nach uns die SYN-Flut*, In c't 15/2011, 2011, `http://heise.de/-1285780`

[10] Paxson, et al.: *RFC-6298: Computing TCP's Retransmission Timer*, June 2011, `http://tools.ietf.org/html/rfc6298`

[11] D. J. Bernstein, E. Schenk: *SYN cookies*, `http://cr.yp.to/syncookies.html`

[12] J. D. Brouer: *DDoS protection, Using Netfilter/iptables*, At DevConf.cz, February 2014

[13] J. D. Brouer: *RFE: Synproxy: auto detect TCP options*, January 2014, `https://bugzilla.redhat.com/1059679`

[14] W. M. Eddy: *Defenses Against TCP SYN Flooding Attacks*, In The Internet Protocol Journal - Volume 9, Number 4, Cisco Press, December 2006

[15] J. Corbet: *The return of nftables*, In Linux Weekly News, Eklektix Inc., August 2013, `https://lwn.net/Articles/564095/`

[16] E. Leblond: *Nftables, what motivations and what solutions*, At Kernel Recipes, September 2013

[17] P. McHardy: *nftables - a successor to iptables, ip6tables, ebtables and arptables*, At Netfilter Workshop (NFWS), 2008

[18] E. Leblond: *Nftables quick howto*, February 2014, `https://home.regit.org/netfilter-en/nftables-quick-howto/`

[19] J. Corbet: *A JIT for packet filters*, In Linux Weekly News, Eklektix Inc., April 2011, `https://lwn.net/Articles/437981/`

[20] T. Leemhuis: *Kernel-Log – Was 3.0 bringt (1): Netzwerk*, In Heise Open Source, Heise Zeitschriften Verlag, 2011, `http://heise.de/-1257064`

[21] J. Corbet: *BPF: the universal in-kernel virtual machine*, In Linux Weekly News, Eklektix Inc., May 2014, `https://lwn.net/Articles/599755/`

[22] P. McHardy: *nftables*, August 2008, `http://web.archive.org/web/20081003040938/people.netfilter.org/kaber/weblog/2008/08/20/`

[23] S. McCanne, V. Jacobson: *The BSD Packet Filter: A New Architecture for User-level Packet Capture*, In Proceedings of the USENIX Winter 1993 Conference (USENIX), pages 259-270, January 1993

[24] J. Corbet: *Ktap or BPF?*, In Linux Weekly News, Eklektix Inc., April 2014, `https://lwn.net/Articles/595565/`