FreeBSD® JOURNAL

January/February 2024

NETWORKING

10th Anniversary
FreeBSD Journal

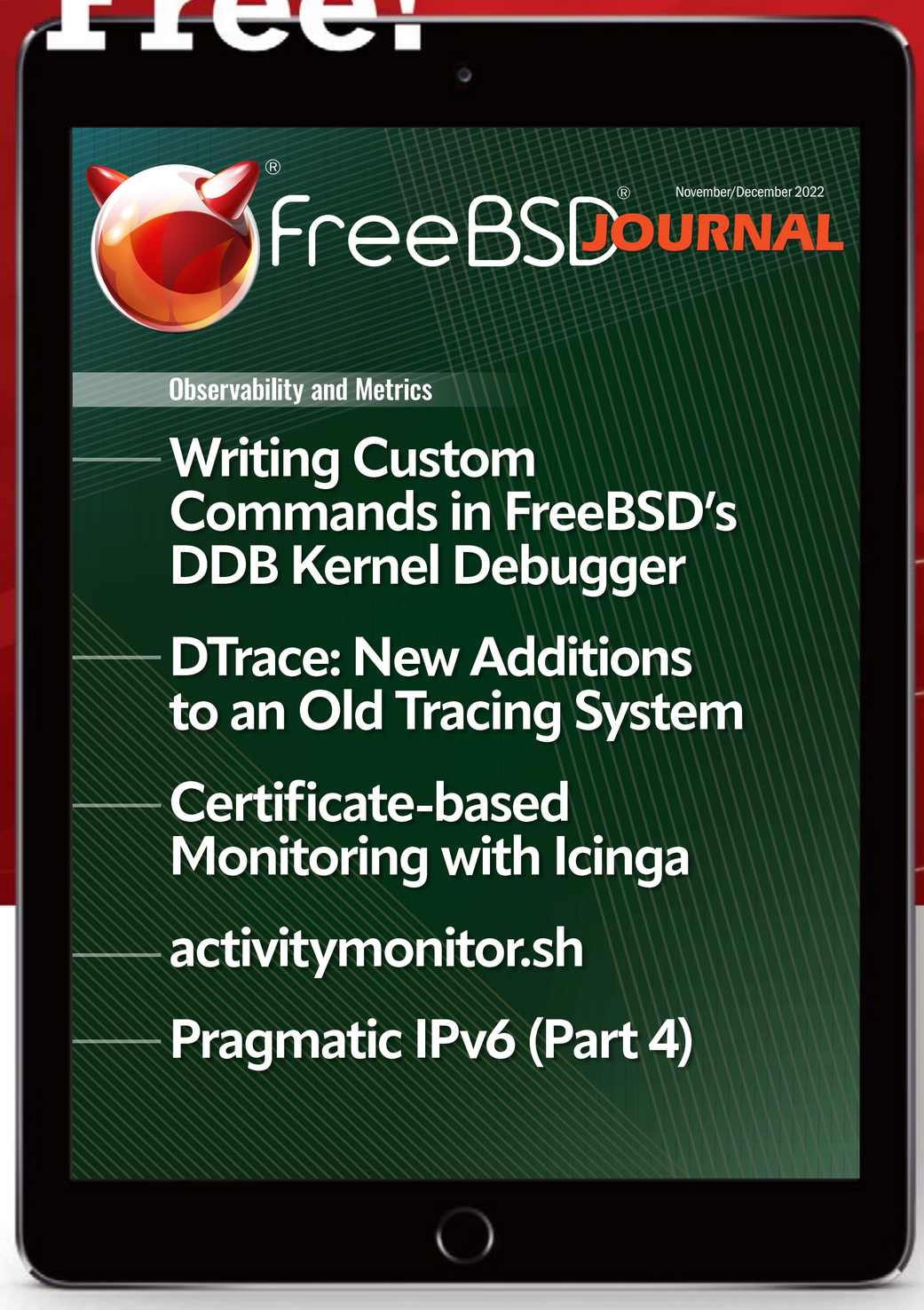
FreeBSD® JOURNAL

The FreeBSD Journal is Now Free!

Yep, that's right Free.

The voice of the FreeBSD Community and the BEST way to keep up with the latest releases and new developments in FreeBSD is now openly available to everyone.

DON'T MISS A SINGLE ISSUE!



2024 Editorial Calendar

- Networking (January-February)
- Development Workflow and CI (March-April)
- Configuration Management Showdown (May-June)
- Storage and File Systems (July-August)
- To come (September-October)
- To come (November-December)

Find out more at: freebsdfoundation/journal

S&W PUBLISHING LLC
PO BOX 3757 CHAPEL HILL, NC 27515-3757

Editor-at-Large • James Maurer
maurer.jim@gmail.com

Design & Production • Reuter & Associates

FreeBSD Journal (ISBN: 978-0-61 5-88479-0) is published 6 times a year (January/February, March/April, May/June, July/August, September/October, November/December).
Published by the FreeBSD Foundation,
3980 Broadway St. STE #103-107, Boulder, CO 80304
ph: 720/207-51 42 • fax: 720/222-2350
email: info@freebsdfoundation.org

LETTER from the Foundation

Welcome to the January/February edition of the FreeBSD Journal! A decade ago, we unveiled the inaugural issue, spotlighting the release of FreeBSD 10.0 and featuring discussions on pivotal topics such as the transition to clang as the base system C and C++ compiler, and ZFS. Over the past ten years, the Journal has showcased a wealth of content, with more than 250 articles, accompanied by numerous book reviews and engaging trip reports. None of this would have been possible without the invaluable contributions from authors across the FreeBSD community, both past and present members of the Editorial Board, and, of course, our dedicated readers. A heartfelt thank you to all!

As we embark on the next decade, we're pleased to announce two structural changes to the Journal. Firstly, we're departing from the browser-based format and introducing HTML versions alongside the traditional PDFs for each article. Secondly, starting with the November/December 2023 issue, the Editorial Board has collaborated with esteemed guest experts--authorities on each issue's theme—to help plan and develop content. These experts not only suggest topics and potential authors but, in some cases, contribute articles themselves. Our recent issue (FreeBSD 14.0) marked the beginning of this collaboration, benefiting from the insights of guest expert Warner Losh, and contributions from Michael Tüxen. Thank you, Warner and Michael!

We value the input of our readers and invite you to share your thoughts. Whether you have feedback on our articles, suggestions for future topics, or an interest in contributing an article yourself, please reach out to us at journaleditor@freebsdfoundation.org.

John Baldwin
Chair of the *FreeBSD Journal* Editorial Board

FreeBSD® JOURNAL

January/February 2024

RACK AND ALTERNATE TCP STACKS FOR FREEBSD

BY RANDALL STEWART AND MICHAEL TÜXEN

In 2017 changes were made to the TCP stack in FreeBSD, allowing the coexistence of multiple TCP stacks. This way, the existing TCP stack could be left untouched and allow innovation at the cost of a limited number of function calls. Some functionality is still shared between all TCP stacks: the implementation of the SYN-Cache including the handling of SYN-Cookies and the initial steps of the handling of incoming TCP segments like checksum verification and looking up the TCP endpoint based on the port numbers and IP addresses. At any given time, a TCP connection is handled by exactly one TCP stack, but this TCP stack can be changed during the lifetime of the TCP connection.

This is where the TCP RACK stack began as a complete rewrite of the original TCP stack from the call to the `tcp_do_segment()` function and many other modularized sub-functions. The initial goal was to add support for a loss detection method called Recent Acknowledgement (RACK). RACK was described in an Internet draft, which became RFC 8985 in 2021. This is where the name of this TCP stack—RACK—comes from. But the TCP RACK stack has grown far beyond just the addition of support for RFC 8985. Part of the rewrite includes a completely different way of handling selective acknowledgement (SACK) information. In the TCP RACK stack, a complete map of all user data sent is maintained that allows an improved handling of retransmissions of user data as well as the addition of the RACK loss detection described in RFC 8985. Many additional features have grown out of this rewrite and are described in this article.

The RACK stack is available in both FreeBSD CURRENT and FreeBSD 14.0.

How to Use the TCP RACK Stack

The RACK stack is available in both FreeBSD CURRENT and FreeBSD 14.0. How to make it available depends on the FreeBSD version.

For FreeBSD 14.0, one needs to add the following two lines to the kernel configuration file

```
option TCPHPTS
makeoptions WITH_EXTRA_TCP_STACKS=1
```

and rebuild the kernel. The first line results in compiling the TCP high precision timer system (HPTS) into the kernel. The second line results in generating a kernel loadable module for the TCP RACK stack (`tcp_rack.ko`). To use the TCP RACK stack, the kernel module must be loaded. This can be done on every reboot by adding the line

```
tcp_rack_load="YES"
```

to the file `/boot/loader.conf`.

In FreeBSD CURRENT, both TCP RACK and HPTS are built as kernel modules by default. Since **tcphpts.ko** is loaded automatically as a dependency of **tcp_rack.ko**, only the latter must be loaded using **kldload**. To load the TCP RACK stack on every reboot, the following two lines need to be added to the file **/boot/loader.conf**:

```
tcphpts_load="YES"
tcp_rack_load="YES"
```

Compiling the TCP RACK stack statically into the kernel of FreeBSD CURRENT is also possible by adding the following two lines to the kernel configuration file

```
option TCPHPTS
option TCP_RACK
```

and rebuilding the kernel.

Note that TCP Blackbox Logging (**option TCP_BLACKBOX**) is now built by default in FreeBSD 14.0 and higher and also in FreeBSD CURRENT for all 64-bit platforms, since it is the standard way that TCP transport developers are both instrumenting as well as debugging the various TCP stacks.

The above describes how to make the TCP RACK stack available on a FreeBSD system. A list of all available TCP stacks is shown by running

```
sysctl net.inet.tcp.functions_available
```

in a shell.

In the upcoming versions—FreeBSD 14.1 and higher—the usage of the TCP RACK stack will be the same as the one described above for FreeBSD CURRENT.

There are different ways of actually using the TCP RACK stack, some involving source code changes of applications, some only involving configuration changes.

The **sysctl**-variable **net.inet.tcp.functions_default** is used to specify the default TCP stack that is used for new TCP endpoints created using the **socket(2)** system call. Executing

```
sysctl net.inet.tcp.functions_default=rack
```

sets the default stack to the TCP RACK stack. By adding the line

```
net.inet.tcp.functions_default=rack
```

to the file **/etc/sysctl.conf** the TCP RACK stack will be the default TCP stack after rebooting the system. When a TCP endpoint is created via a listener, the TCP stack is either inherited from the listener or based on the default TCP stack depending on the **sysctl**-variable **net.inet.tcp.functions_inherit_listen_socket_stack** being non-zero or zero. The default value of this variable is one.

It is also possible to change the TCP stack of individual TCP connections by using the **tcpsso(8)** command line tool as described in the man-page of the tool.

If the source code can be changed, the **IPPROTO_TCP**-level socket option with the name **TCP_FUNCTION_BLK** can be used to switch the TCP stack being used for the socket to the TCP RACK stack. The option value has the type **struct tcp_function_set**. For example, the following code performs this:

```
struct tcp_function_set tfs;

strncpy(tfs.function_set_name, "rack", TCP_FUNCTION_NAME_LEN_MAX);
tfs.pcbcnt = 0;
setsockopt(fd, IPPROTO_TCP, TCP_FUNCTION_BLK, &tfs, sizeof(tfs));
```

Using the TCP RACK stack allows the use of a number of features that the default TCP stack does not currently support. A lot of these features can be controlled via **`IPPROTO_TCP`**-level socket options or **`sysctl`**-variables under **`net.inet.tcp.rack`**.

Features of the TCP RACK Stack

The following sections describe the most important features provided by the TCP RACK stack.

RACK/TLP

Recent Acknowledgement (RACK) and Tail Loss Probe (TLP) are two integrated features within the TCP RACK stack. Recent acknowledgement changes the way that packet loss is detected and retransmissions are triggered. The loss detection implemented in the FreeBSD base stack and specified in RFC 5681 takes three duplicate acknowledgments or acknowledgement arrivals with SACK to get the TCP stack to send out a retransmission. In some cases, where, for example, less than four packets have been sent, this will cause the TCP stack to send the retransmission only after a retransmission timeout has occurred. RACK changes this so that when a SACK arrives, if enough time has elapsed since the sending of the lost packets, a retransmission happens immediately. If not enough time has occurred (usually the time is a bit larger than the current RTT), then a small RACK timer is started, and when this expires the retransmission is sent. This then will fix many—but not all—of the cases where a retransmission timeout would have to force out data. The last case is solved by the TLP. This is where whenever the TCP RACK stack has sent data, it starts a TLP timer instead of a retransmission timer. If the TLP timer expires, the TCP RACK stack sends either a new segment or the last segment sent. The hope of this TLP-sent segment is that the sender would either receive an acknowledgement back indicating all data has been received (a case where the last acknowledgement was lost) or the TLP would elicit a SACK, which would allow the normal fast recovery mechanisms to take over without hitting a retransmission timeout and thus collapsing the congestion window to 1 MSS.

A user of the TCP RACK stack, just by enabling the stack, gets the benefits of both RACK and TLP automatically. No socket option or configuration is required by the upper layer.

Proportional Rate Reduction (PRR)

Proportional Rate Reduction (PRR) is another automatic built-in feature of the TCP RACK stack, specified in RFC 6937 and currently being updated by the IETF. PRR improves the way data is sent during fast recovery. When using the TCP congestion control as specified in RFC 5681, the congestion window is reduced in half on entering fast recovery. This then causes a stall in sending new data during fast recovery. Basically, the sender must wait for one-half of the outstanding data to be acknowledged, and then the sender can start sending new data (along with our retransmissions). This causes a "stall" in the data flow between the sender and receiver. PRR is designed to improve that, such that during fast recovery, a new data segment can be sent roughly every other acknowledgment. This then prevents

the data "stall" and keeps data continually moving thus keeping the RTT and other transport metrics active and updating.

RACK Rapid Recovery (RRR)

RACK Rapid Recovery (RRR) is an interesting feature that started as a bug. In the initial development, the TCP RACK stack inadvertently allowed a case where when a SACK arrived that declared more than a single segment missing and the RACK timer expired for all of the data, the TCP RACK stack would send one segment and start a RACK timer. When the RACK timer expired (which was set to the RACK minimum timeout value of 1 ms), the TCP RACK stack would send another one of the missing segments. This would repeat until all of the missing segments were sent. This effectively ignores PRR during the initial recovery with a cost of sending further PRR segments much later. So, for example, if RRR sent 3 segments, the first retransmission and two extras, it would take the arrival of roughly 6 more acknowledgements before PRR would send out a new segment.

When this bug was discovered and "fixed," the quality of experience (QoE) for the users degraded. This is because those early segment losses often hold up the delivery of quite a few segments of data. This led to adding this as a feature that can be turned off and also has programmability into the amount of time since the time in question effectively makes RRR paced at 12Mbps in its default setup. By default, this feature is on with the RRR recovery rate set for one segment every millisecond. This results in a rate of 12 Mbps assuming a maximum transmission unit (MTU) of 1500 bytes.

SACK Attack Detection

One of the downsides of keeping a complete map of what is being sent is that this map can grow quite large in some circumstances. The TCP RACK stack attempts at all times to collapse the map to as small as possible yet still keep track of all stages of outstanding data. There is, however, an introduced possibility that a malicious peer can be designed to attack the memory and CPU resources used by the TCP RACK stack for a TCP connection by constantly splitting the sendmap into smaller and smaller pieces so that TCP RACK stack uses large amounts of memory and spends excessive amounts of time searching through that memory. An example might be where an attacker sends SACKs for every other byte. This can pose a serious threat and can impact a machine in undesired ways.

The TCP RACK stack includes the optional compiled in feature called TCP_SAD_DETECTION. The SAD stands for SACK Attack Detection (SAD). One can enable it for the TCP RACK stack by adding the line

```
option TCP_SAD_DETECTION
```

to the kernel configuration file and rebuilding the kernel.

Once added, it is on by default. It montors for a malicious peer and if detected, it disables the processing of SACKs from the peer. This degrades that single peer's performance but does not prevent the connection from making progress. It, in effect, becomes a connection that responds as if no SACK was ever enabled. This penalizes loss recovery, but still allows the connection to continue.

Burst Mitigation

Built into the TCP RACK stack, and on without any user intervention, is burst mitigation. To mitigate bursts, the stack will only send out a set size (the max burst size) at a send

opportunity and start either a small timer (to send out more) or depend on the returning acknowledgement stream to prompt the sending of more data. This helps mitigate large bursts that can cause excessive loss.

Support for TCP Blackbox Logging (BBLog)

One of the interesting aspects of the TCP RACK stack is the extensive support of TCP Blackbox Logging for both debugging and just general statistical analysis and instrumentation. This makes it much easier to track down problems and to acquire analysis of connection behavior.

Large Receive Offload (LRO) Integration for Burst Mitigation

TCP Large Receive Offload (LRO) is a feature to reduce the CPU resources needed for a receiver by coalescing multiple received TCP segments into a single one before passing them into the TCP stack. Often this results in a loss of information about the individual received segments but reduces the CPU resources needed, since fewer TCP segments need to be processed by the TCP stack.

An interesting feature interaction is a set of changes that have been made to the LRO code for better support of pacing in the TCP RACK stack. When a TCP connection is doing burst mitigation, it tends to walk through the send path more often, sending smaller bursts. Due to this, changes were made to the LRO code allowing all of the timing data on packet arrival information to be carried through to the TCP RACK stack without loss. Basically, during packet processing, the LRO code looks up to see if the packet is associated with a connection that allows it to queue packets directly to the TCP RACK stack. If so, the packets are enqueued directly to the connection and, depending upon connection state, the connection may be woken up. In cases where the TCP RACK stack is doing burst mitigation or pacing, that wake up is deferred until a timer expires and something can be done with the inbound acknowledgments. These steps also bypass IP stack processing and thus provide an additional mild reduction of needed CPU resources.

A Host of Alternate Features

Many other features are available in the TCP RACK stack via various socket options and `sysctl`-variables. Currently, TCP RACK stack supports 58 socket options that enable various features including pacing, burst mitigation options and recovery response modifications. Besides the socket options, around 150 `sysctl`-variables exist to either make a socket option apply to all connections or to modify various TCP RACK stack default configurations. All of these features and configuration are available to help adjust the TCP RACK stack to better conform to your network conditions and requirements.

How Netflix Evolves the TCP RACK Stack

Netflix currently uses only the TCP RACK stack, the FreeBSD default stack is present, but not in use. The way Netflix uses the TCP RACK stack is a bit novel and worth noting. Netflix actually keeps several generations of the TCP RACK stack named for its release numbers. At all times, it keeps the "latest" TCP RACK stack with all of the leading-edge features under development by its transport group.

Periodically, when a release is cut, the latest TCP RACK stack under development is copied and supported based on a release number. This TCP stack is then evaluated based upon QoE and CPU performance in comparison to the previously released TCP stack which is the

default in use. When the newest TCP RACK stack is at least as good or better than the old TCP RACK stack, the default is switched to the newer TCP RACK stack in the next release. The old TCP RACK stack is maintained for several releases and eventually removed.

New features on TCP RACK stacks are also tested this way so that it can be determined if a feature adds value or not. Reducing network impact with no degradation of QoE for Netflix's users is one of the transport team's main goals, so that Netflix is both a better network citizen and at the same time providing a good overall QoE.

Conclusion and Outlook

The TCP RACK stack provides a strong alternative to the FreeBSD base stack. It adds more features and options that provide a richer set of alternatives for the application developer to better tailor the TCP experience for users.

The TCP RACK stack was extensively tested using the Netflix setup and workload. But it is important to also test it in other setups and workloads. Therefore, it would be great if users could test the TCP RACK stack on their hardware, using their setup, and under their workload. Please report any issues found during testing to net@freebsd.org or to the authors of this article. Depending on the feedback and further testing, the TCP RACK stack might become the default stack for FreeBSD in the future.

RANDALL STEWART (rrs@freebsd.org) has been an operating system developer for over 40 years and a FreeBSD developer for over 10 years. He specializes in Transports including TCP and SCTP but has also been known to poke into other areas of the operating system. He currently works at Netflix in its transport team, supporting the TCP stack while innovating to constantly improve user QoE.

MICHAEL TÜXEN (tuexen@freebsd.org) is a professor at the Münster University of Applied Sciences, a part-time contractor for Netflix, and a FreeBSD source committer since 2009. His focus is on transport protocols like SCTP and TCP, their standardization at the IETF and their implementation in FreeBSD.

Updates On TCP in FreeBSD 14

BY RICHARD SCHEFFENEGGER

It's been about 3 ½ years since I last reported on the area of the FreeBSD project I focus on, namely, the TCP protocol implementation. For those who don't know, FreeBSD doesn't feature only one TCP stack, but multiple ones with development occurring dominantly in the RACK and base stack. Currently, the one used by default (base stack) is the stack long evolved and derived from BSD4.4. Also, since 2018, we have had a completely refactored stack ("RACK stack" – with the ***R***ecent ***ACK***nowledgement mechanism as its namesake), that provides many advanced capabilities that are lacking in the base stack. For example, the RACK stack provides high granularity pacing capabilities. That is, the stack can time the sending of packets and even out the consumption of network resources. In contrast, if an application presents the base stack with a sudden burst of data to transmit, there are instances where this data will be sent out in a large burst at near line rate (the speed of the interface, provided the CPU and internal busses are not the bottleneck). This happen most notably whenever there is a short application pause from the last application IO by a couple tens of milliseconds. (Further details on the RACK stack are beyond the scope of this article and can be read in the accompanying article by Michael Tuexen and Randall Stewart.).

Here, I want to highlight some of the new features that have been brought into the base stack – many of which are enabled by default, and some of which may need to be specifically activated. Each feature will be described with details that may help improve the networking experience.

Overall, there have been around 1033 commits since the release of FreeBSD 13.0 to the sys/netinet directory where all the transport protocols traditionally live. This gives an overview of selected changes to the base stack, where functionality was improved:

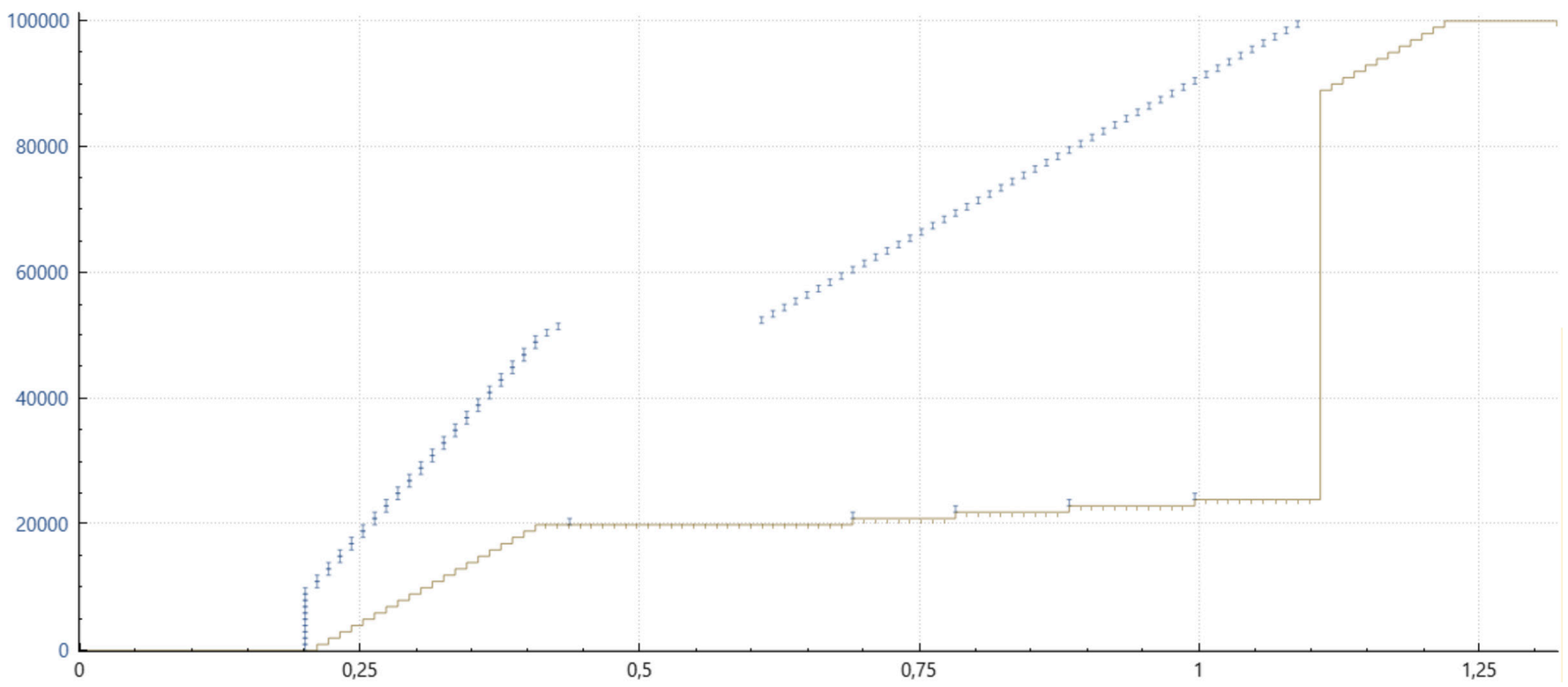
Proportional Rate Reduction

The first feature brought into the base stack is PRR – Proportional Rate Reduction (RFC6937). To understand PRR on a high level, let's first understand how SACK behavior functions during loss recovery. One issue with standard SACK loss recovery was that while the congestion window is adjusted on entering loss recovery (e.g., to 50% of the value at onset of congestion with NewReno or 70% with Cubic) after a single packet loss, the estimation of how many packet are still in flight will initially not allow any packets to be sent while ACKs return for the first half (NewReno) or initial 30% of the window. Once that limit has been reached, every incoming ACK will elicit a new packet, but this may well happen at an effective rate that overwhelms the congestion point in the network. The initial quiet period may serve to drain queues and allow for the subsequent, faster-than-ideal transmissions. Often that behavior leads to subsequent losses (perhaps even losses of retransmitted packets – more on that later).

To quickly adjust the effective sending rate – and also deal more appropriately when there are multiple losses of data packets or maybe even losses of ACK packets – PRR will calculate how much data should be sent out for every new, incoming ACK and sends out as many full-sized packets as appropriate at that that time. In the simple example with NewRe-

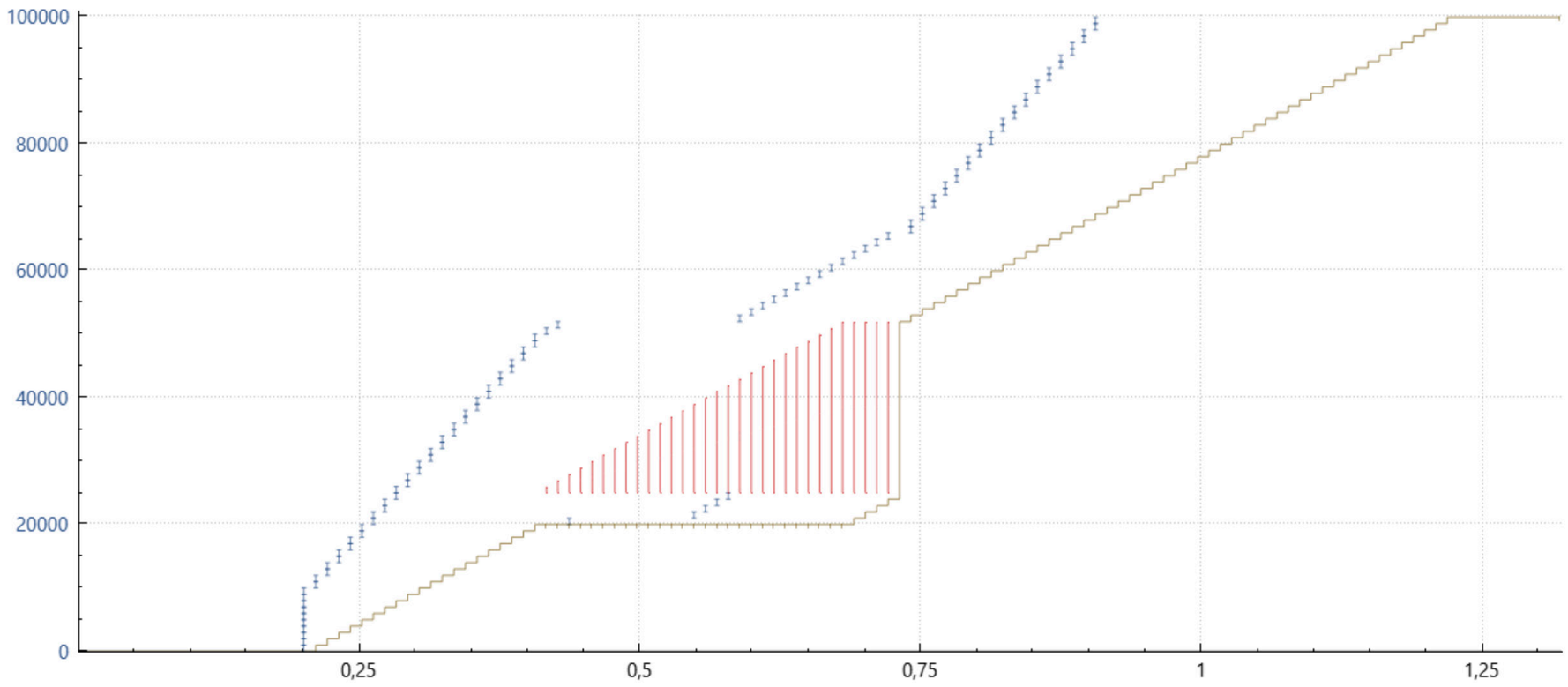
no with a reduction of the congestion window to half and just a single loss, this will cause one new packet to be sent for every two ACKs returned. Thus, the sending rate will adjust instantly to exactly half of what it used to be – stopping the congested device from being overloaded. In the presence of multiple data losses or ACKs discarded, PRR may inject even more than one packet when an ACK is finally received, making good on these missed sending opportunities. Overall, this behavior ensures that at the end of the window (RTT) when loss recovery happens, the effective congestion window is as close as possible to the expected congestion window, and that no transmit opportunities are missed, even under problematic scenarios like multiple packet losses or ACK losses.

Hopefully, a few graphs can explain this niche detail better. Below, we have time-sequence graphs which can be obtained from wireshark or the combination of tcptrace and associated xplot. The small blueish vertical bars indicate when a particular packet covering the data sequence was sent – as on the left axis. The greenish more horizontal line below indicates which data was received contiguous by the receiver. Red vertical lines signify any discontinuous range of data that made it to the receiver.



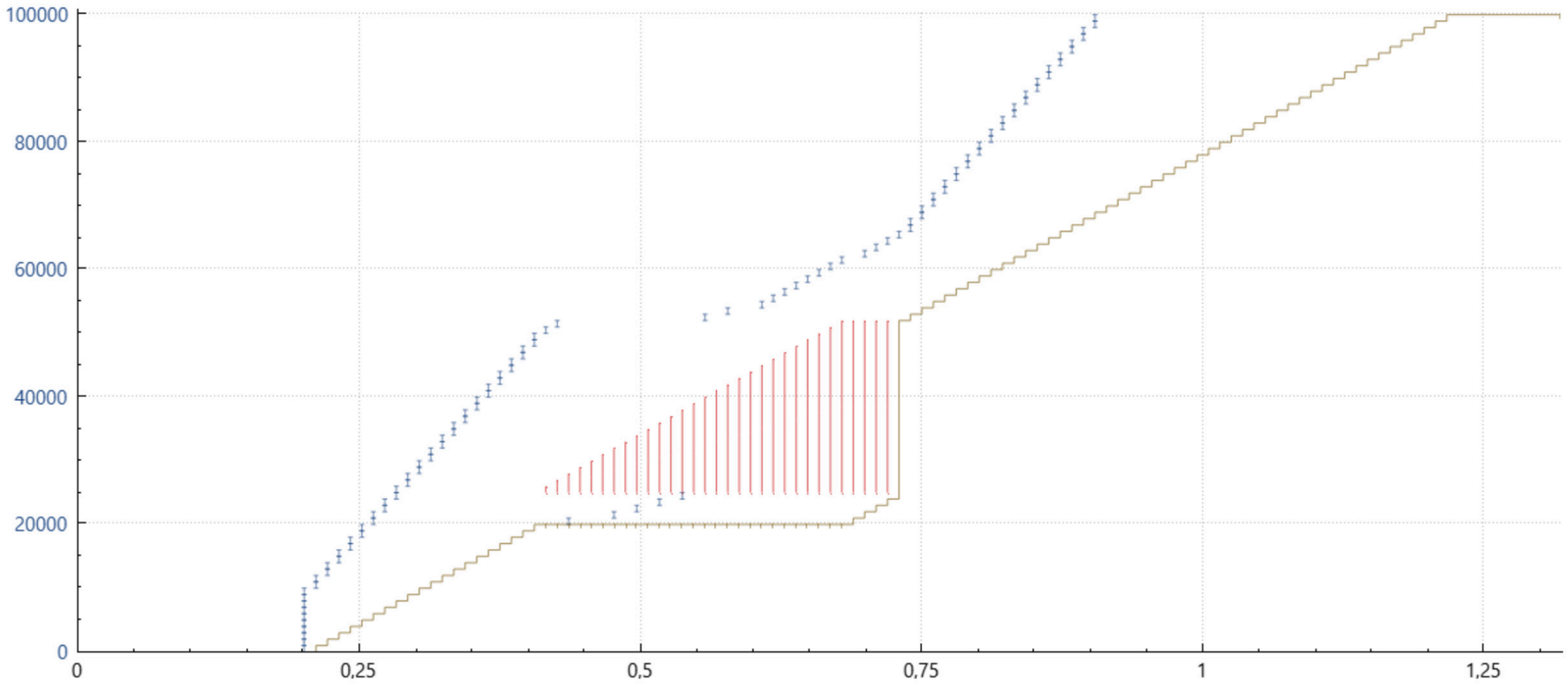
Cubic Without SACK or PRR, Classic NewReno Loss Recovery

Note that only a single data packet can be recovered within one window (or round-trip time) and the long stretch of horizontal green line indicates the latency induced before the receiving application gets to process additional data.



Cubic with SACK, but no PRR

As this example shows, SACK dramatically improves the situation, since all the lost packets can (typically) be retransmitted within one RTT. However, take note of the pause and resumption of sending on each ACK later. This behavior drives data at an effective rate that caused some packets to be dropped into the network. Often, this causes one or more of the retransmissions to arrive too quickly and the network drops the retransmission. The only recourse then is to wait for a retransmission timeout (RTO).



Cubic with SACK (6675) and PRR

The improvement with PRR depicted here is subtle. Where previously no data was sent for half a window, and then at the old, likely too high rate for the second half, PRR injects packets approximately every other received ACK until the new sending rate has been reached, and then on nearly each subsequent incoming ACK. This serves to reduce the effective sending rate of the retransmissions and making it less likely that these will get discarded by the network. Fewer RTOs and improved latency are the consequences.

The graph shown here is not entirely correct but attempts to convey the aspect of PRR "dithering" packets sent appropriately over the received ACKs to send them out – in this case, on average, 0.7 packets for every ACK including those which may have been discarded by the network.

The final update in this space was that PRR now automatically switches to a less conservative mode unless there are additional losses in loss recovery. This effectively improves the transmission speed during loss recovery, similar to what would happen during normal operation in the congestion avoidance phase. PRR works best (naturally) in conjunction with SACK, but also when only non-SACK duplicate ACKs are available. Even with nothing but ECN feedback, PRR improves the transmission timings.

SACK Handling

In recent years, the adherence of the base stack to SACK loss recovery as specified in RFC6675 has been improved. But while parts of the estimation on how much data is still outstanding in the network were improved, other aspects of RFC6675 were missing.

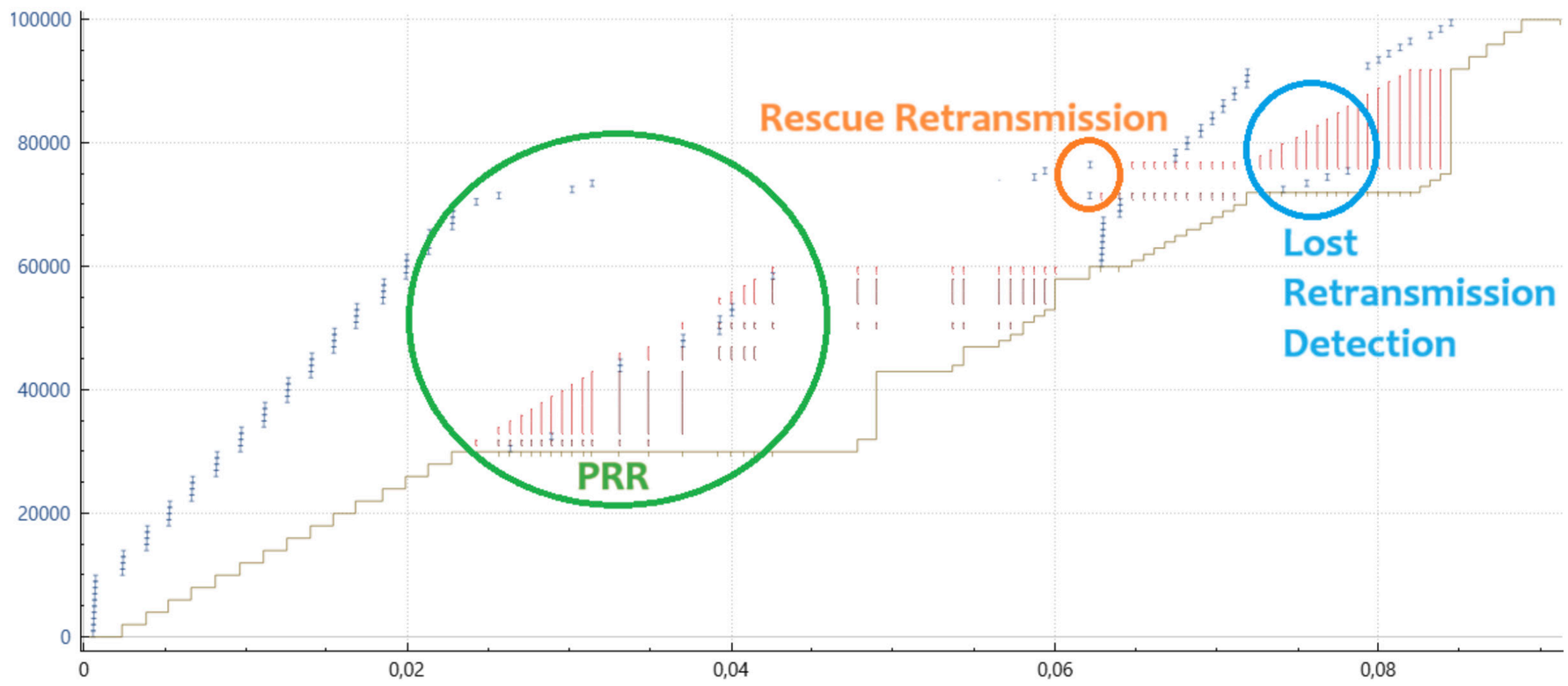
Improvements in this space now include the use of so-called rescue retransmissions – a precursor of the Tail-Loss Probe, which is implemented in the RACK stack. In short, when the final few packets of a transfer are lost in addition to earlier packet losses, the stack can detect the problem and will retransmit the ultimate packet to perform a timely loss recovery.

And, by implementing additional accounting when processing any incoming SACK block,

the stack keeps better track of whether a particular packet ought to have left the network either by being received or having very likely been dropped.

A final enhancement was to track whether retransmissions may also have been dropped by the network, but unlike RACK, which uses the time domain, the base stack looks at the sequence domain. While this lost retransmission detection is not specified in the RFC series, it's an extremely valuable addition to reduce the flow completion time / IO service response time for any request-response (e.g., RPC) protocol using the TCP stack. Tracking and recovering from lost retransmissions does not yet happen by default. In FreeBSD 14, this can be activated with net.inet.tcp.do_lrd – but with FreeBSD 15, this will move to net.inet.tcp.sack.lrd and be enabled by default.

Overall, these changes make the base stack more resilient during frequently encountered pathological issues around congestion in the IP network.



Finally, the base stack (and the RACK stack) creates DSACK (RFC2883) responses when receiving spurious duplicate data packets. While receiving such DSACK information doesn't influence the stack behavior, providing this to a remote sender may permit that sender to better adjust to the specific network path behavior – e.g., Linux could increase the dupthresh or detect spurious retransmissions because of a spike in the path round-trip time (RTT).

Logging and Debugging

Over the decades, the base stack accumulated several different mechanisms to be debugged on a live system. One of the least known tools, trpt, and its support was removed in FreeBSD 14. Still, numerous other options exist (dtrace, siftr, bblog, …).

BlackBox Logging was introduced with the RACK stack and extended to cover more and more of the base stack as well. Tools are being prepared to extract internal state changes from a running system as well as extract them from core dumps – along with the packet trace itself. (See https://github.com/Netflix/tcplog_dumper and https://github.com/Netflix/read_bbrlog)

Cubic

As described in my previous article, TCP Cubic is the de facto standard congestion control algorithm in use virtually everywhere. Recently, Cubic was also made the default for FreeBSD – regardless of which TCP stack is being used.

One notable extension here is the addition of HyStart++. When a TCP session starts up, the congestion control mechanism quickly ramps up bandwidth during a phase called slow start. Traditionally, the slow start phase ends when the first indication of congestionpacket loss, or possibly an explicit congestion notification (ECN) feedback – is received. With HyStart++, which is implemented as part of the Cubic module and always enabled, the RTT is monitored. When the RTT starts to rise – possibly because network queues start forming – a less aggressive phase (conservative slow start) is entered and the RTT is still monitored because any timing-based signal is notoriously hard to obtain reliably. If it turns out that the RTT reduces again while in this conservative slow start phase, regular slow start is resumed. If not, the less aggressive sending pace in CSS limits the so-called overshoot – which is how much data may need to be recovered due to inevitable losses.

Accurate Explicit Congestion Notification

As alluded to earlier, ECN is a mechanism to avoid packet losses as the sole signal to indicate congestion events. Over the last decade, there has been a large effort in the Internet Engineering Task Force (IETF) to improve this signaling. While, originally, ECN was viewed as an "identical" signal to packet loss, a more frequent signal with different semantics was found to work better to maintain shallow (fast) queues across a large range of bandwidths. The full architecture is named Low Latency, Low Loss, Scalable (L4S). While not all the pieces in FreeBSD are currently ready to implement a proper "TCP Prague" implementation, many individual features – such as the DCTCP congestion control module and, relevant here, Accurate ECN (AccECN) – are now part of the stack in FreeBSD 14.

ECN is a mechanism to avoid packet losses as the sole signal to indicate congestion events.

While in classic ECN, only a single congestion experience mark can be signaled per RTT. This necessitates a heavy-handed management by the congestion control module. In fact, CE marks are viewed as equal to packet loss indications when adjusting the TCP bandwidth while operating in RFC3168 mode. In contrast, with AccECN, an arbitrary number of explicit congestion marks can be signaled back to the data sender by the receiver. This enables a more modulated and fine-grained signal to be extracted from the network. This becomes relevant in environments where DCTCP – with the modified, much more aggressive marking thresholds by the intermediate switches – is to be used. It is also one of the key ingredients of the Low Latency, Low Loss, Scalable (L4S) architecture – also known as TCP Prague.

Authentication and Security

Recently, the RACK stack gained the capability to fully handle MD5 authentication of TCP packets. This is an improvement that allows the use of BGP with the RACK stack – another step in making the RACK stack fully featured and useable in any generic circumstance.

For a long time, there has been a tight coupling between two of the features in RFC7323

(RFC1323) – Window Scaling and Timestamp options. In this space, we now allow either of these to be enabled independently of the other while the default still permits both to be active. This can now be achieved by setting net.inet.tcp.rfc1323 not only to on (1) or off (0), but also 2 (only window scale) and 3 (timestamps only). Furthermore, in accordance with RFC7323, it is now possible to further secure TCP sessions by requiring proper use of TCP timestamps under all circumstances. This is achieved by setting net.inet.tcp.tolerate_missing_ts to 0.

What's Next?

While the improvements of various aspects of TCP features are well into the diminishing returns phase, there are still a couple of further enhancements under discussion.

For example, an erratum to RFC2018 (Selective Acknowledgments) now allows information to be retained during a Retransmission Timeout (RTO), unlike previously. The main motivation at the time of the original standard was allowing for "reneging" by the receiver. Unless explicitly acknowledged, subsequent data could still be discarded, e.g., because of memory pressure. In practice, such reneging hardly ever happens, but retransmission timeouts during a SACK loss recovery phase do occur quite frequently. Retaining this information allows more efficient retransmissions even after an RTO. The challenge is that the base stack has implicit tight couplings with other aspects of what should happen after a retransmission timeout (such as slow starting from a very small congestion window). Also, the impact of this change after an RTO needs to be evaluated – driving some additional capabilities into the dummynet path emulator to model loss in more controllable ways.

While the improvements of various aspects of TCP features are well into the diminishing returns phase, there are still a couple of further enhancements under discussion.

RICHARD SCHEFFENEGGER has been a FreeBSD committer since April 2020 and is interested in improving the features and functionality of the TCP stack, mainly focusing on the slow path (loss recovery, congestion control handling), and actively developing enhancements such as Accurate ECN with the IETF.

if_ovpn or OpenVPN

BY KRISTOF PROVOST

Today[1], you're going to be reading[2] about OpenVPN's DCO.

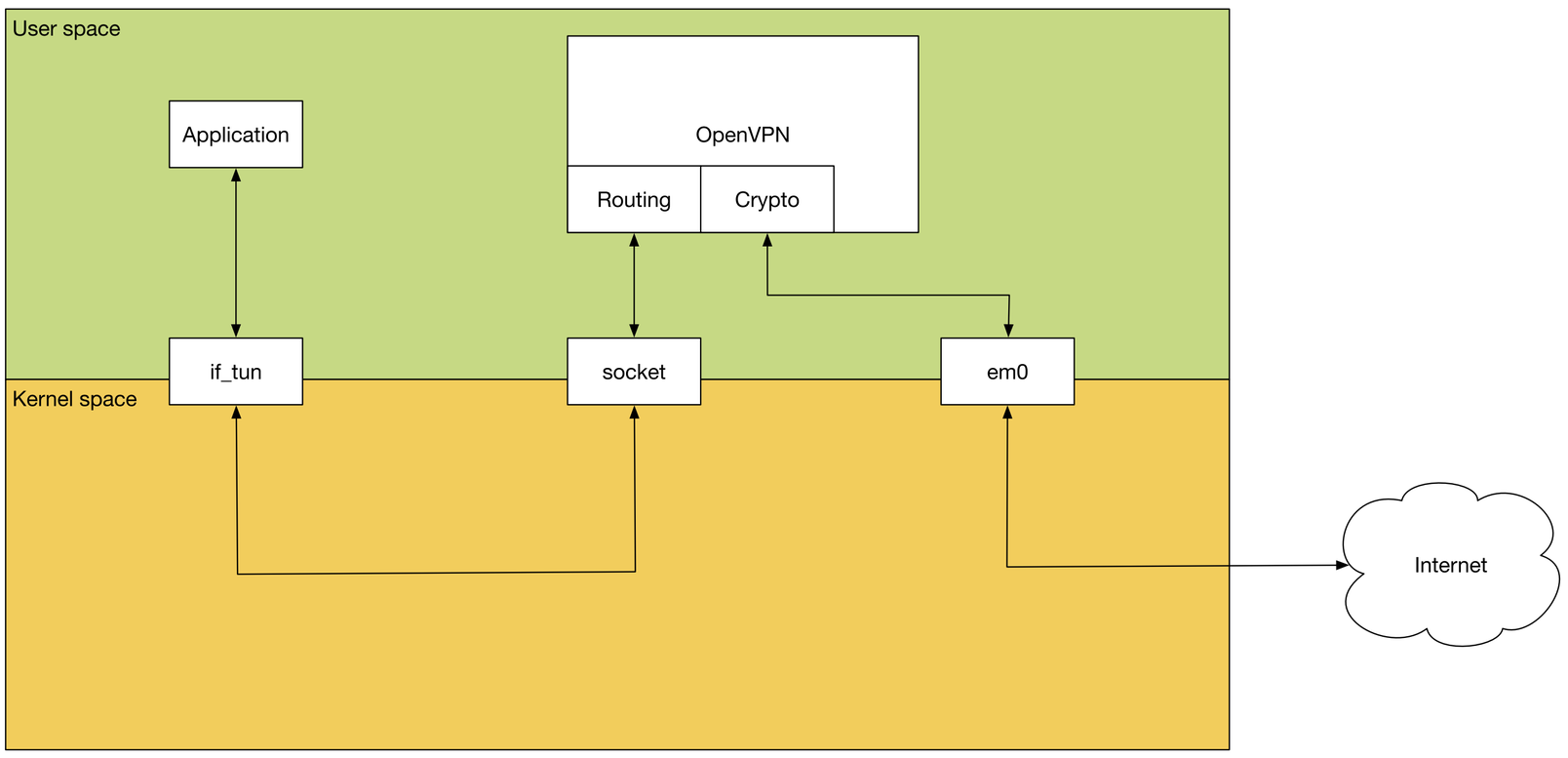
Initially developed by James Yonan, OpenVPN saw its first release on May 13, 2001. It supports many common platforms (such as FreeBSD, OpenBSD, Dragonfly, AIX, ...) and a few less common ones (macOS, Linux, Windows) as well. It supports peer-to-peer and client-server models, with pre-shared key, certificate, or username/password-based authentication.

As you'd expect with any project that's been around for more than 20 years, it grew many features for many different use cases.

The Problem

While OpenVPN is very nice, clearly there must be a problem. Without a problem, this article wouldn't be very interesting[3]. There is, indeed, an issue, and it is that OpenVPN is implemented as a single-threaded, userspace process.

It uses if_tun to inject packets into the network stack. As a result, its performance has not kept up with current connectivity rates. It also makes it difficult to take advantage of modern multi-core hardware or cryptographic offload hardware.



The main issue with OpenVPN's performance is its userspace nature. Incoming traffic is naturally received by a NIC, which would typically DMA the packet into kernel memory. It is then processed further by the network stack until that works out what socket the packet belongs to, and passes it to userspace. This socket may be UDP or TCP.

Passing the packet to userspace involves copying it, at which point the userspace OpenVPN process verifies and decrypts the packet and re-injects it into the network stack using if_tun. This means copying the plain-text packet back into the kernel for further processing.

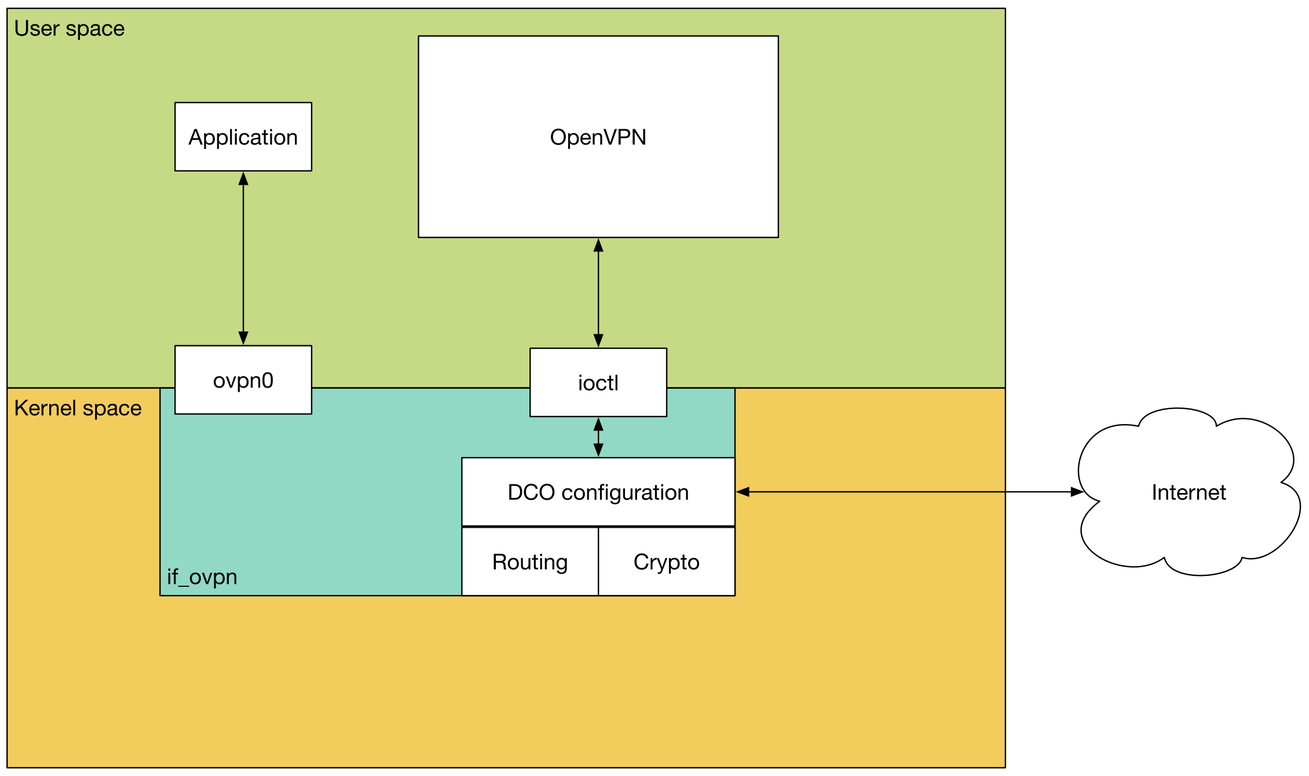
Inevitably all this context switching and copying back and forth has a significant impact on performance.

In the current architecture it's very hard to make significant performance improvements.

What is DCO

Now that we've established what our problem is, we can start thinking about solutions[4].

If our problem is context switches to userspace then one plausible solution is to keep the work inside the kernel, and that's what DCO–Data Channel Offload–does.



DCO moves the data channel, that is, the cryptographic operations and the tunneling of the traffic, into the kernel. It does this through a new virtual device driver, if_ovpn. The OpenVPN userspace process remains responsible for connection setup (including authentication and option negotiation), coordinating with the if_ovpn driver via a new ioctl interface.

The OpenVPN project decided that the introduction of DCO was a good opportunity to remove some legacy features and do some general tidying up. As part of that, they've taken the Henry Ford approach to encryption algorithm choice. You can have any algorithm you like, as long as you like AES-GCM or ChaCha20/Poly1305. In black.

DCO also does not support compression, layer 2 traffic, non-subnet topologies or traffic shaping[5].

It's important to note here that DCO does not change the OpenVPN protocol. It's possible for a client to use it with a server that does not, or vice versa. You'll get the biggest benefit when both sides use it, of course, but that's not required.

Considerations

This is the part where I talk up just how hard this all was, so you'll all be impressed that I actually got this to work. Does that still work if I tell you that's what I'm doing? Let's find out!

Anyway, there are a couple of things that needed special attention:

Multiplexing

The first issue is that OpenVPN uses a single connection to transport both the tunneled data and the control data. The tunneled data needs to be handled by the kernel, and the control data is handled by the OpenVPN userspace process.

You can see the issue. The socket is initially opened and fully owned by OpenVPN itself. It sets up the tunnel and handles the authentication. Once that's completed, it partially hands over control to the kernel side (i.e., if_ovpn).

This means informing if_ovpn of the file descriptor (which the kernel uses to look up the in-kernel struct socket), so it can hold a reference to it. That ensures the socket doesn't go away while the kernel is using it. Perhaps because the OpenVPN process was terminated, or because it was having a bad day and decided to mess with us. It's userspace, it does crazy things.

For those of you who want to follow along in the kernel code, you're looking for the `ovpn_new_peer()`[6] function.

Having looked up the socket we can now also install the filtering function via `udp_set_kernel_tunneling()`. The filter, `ovpn_udp_input()`, looks at all incoming packets for the specified socket, and decides if it's a payload packet which it should handle, or a control packet which OpenVPN in userspace should handle.

This tunneling function is also the only change I had to make to the rest of the network stack. It needed to be taught that certain packets would be handled by the kernel and others could still be passed through to userspace. That was done in https://cgit.freebsd.org/src/commit/?id=742e7210d00b359d81b9c778ab520003704e9b6c .

The `ovpn_udp_input()` function is the main entry point for the receive path. The network stack hands packets over to this function for any UDP packets arriving on the socket it's been installed on.

The function first checks if the packet can be handled by the kernel driver. That is, the packet is a data packet and it's destined for a known peer id. If that's not the case the filter function tells the UDP code to pass the packet through the normal flow as if there were no filter function. That means the packet will arrive on the socket and be processed by OpenVPN's userspace process.

> Pretty much every modern CPU has multiple cores, and it'd be kind of nice to be able to use more than just one of them.

Early versions of the DCO driver had separate ioctl commands to read and write control messages, but both the Linux and FreeBSD drivers have been adapted to use the socket instead. This simplifies handling of both control packets and new clients.

If, on the other hand, the packet is a data packet for a known peer, it is decrypted, has its signature validated, and is then passed on to the network stack for further processing.

For those of you following along, that's done here https://cgit.freebsd.org/src/tree/sys/net/if_ovpn.c?id=da69782bf06645f38852a8b23af#n1483 .

UDP

OpenVPN can be run over both UDP and TCP. While UDP is the obvious choice for a layer 3 VPN protocol, some users need to run it over TCP to transit firewalls.

The FreeBSD kernel offers a convenient filter function for UDP sockets, but has no equivalent for TCP, so FreeBSD if_ovpn currently only supports UDP and not TCP.

The Linux DCO driver developer was rather more ... courageous and has chosen to implement TCP support as well. The developer did, against the odds, in fact survive this experience, and is now significantly wiser.

Hardware Cryptography Offload

if_ovpn relies on the in-kernel OpenCrypto framework for cryptographic operations. This means it can also take advantage of any cryptographic offload hardware present in the system. This can further improve performance.

It's already been tested with Intel's QuickAssist Technology (QAT), the SafeXcel EIP-97 crypto accelerator and AES-NI.

Locking Design

Look, if you thought you were going to get a discussion of kernel code without having to talk about locking, I don't know what to tell you. That was naively optimistic of you.

Pretty much every modern CPU has multiple cores, and it'd be kind of nice to be able to use more than just one of them. That is, we can't just lock out other cores while one core is doing work. It's impolite. It also doesn't perform well.

Happily, this turned out to be reasonably easy to do. The entire approach is based on distinguishing read and write accesses to if_ovpn's internal data structures. That is, we allow many different cores to look up things at the same time but will only ever allow one to change things (and then not allow any readers while the change is being made). That turns out to work well enough because—most of the time—we don't need to change things.

Each OpenVPN DCO platform has its own unique way of communicating between userspace OpenVPN and the kernel module.

The common case, when we receive or send packets, just needs to look up keys, destination addresses and ports and other related information.

It's only when we modify things (i.e., on configuration changes or re-keying) that we need to take a write lock, and that we pause the data channel. That's brief enough that our puny human brains won't notice it, and that makes everyone happy.

There's one exception to this "we don't make changes to process data" rule, and that is packet counters. Every packet gets counted (twice even, once for the packet count, once for a byte count), and that has to be done concurrently. Here, too, we are lucky, in that the kernel's `counter(9)` framework is designed exactly for this situation. It keeps totals per CPU core so that one core will not affect or slow down another. It's only when the counters are actually read that it will ask each core for its total and will add them up.

Control Interface

Each OpenVPN DCO platform has its own unique way of communicating between userspace OpenVPN and the kernel module.

On Linux, this is done through netlink, but the if_ovpn work was completed before FreeBSD's netlink implementation was ready. As I'm still on probation for my last causality violation, I decided to use something else instead.

The if_ovpn driver is configured through the existing interface ioctl path. Specifically, the `SIOCSDRVSPEC`/`SIOCGDRVSPEC` calls.

These calls pass a **`struct ifdrv`** to the kernel. The **`ifd_cmd`** field is used to pass the command, and the **`ifd_data`** and **`ifd_len`** fields are used to pass device-specific structs between kernel and userspace.

if_ovpn deviates somewhat from the established approach, in that it transmits serialized nvlists rather than structs. This makes extending the interface easier. Or, rather, it means we can extend the interface without breaking existing userspace consumers. If a new field is added to a struct, its layout changes which either means that the existing code will refuse to accept it due to its size mismatch[7] or get very confused because fields no longer mean what they used to mean.

Serialized nvlists allow us to add fields without confusing the other side. Any unknown fields will just be ignored. This makes adding new features much easier.

Routing Lookups

You might think that if_ovpn wouldn't need to worry about routing decisions. After all, the kernel's network stack has already made the routing decision by the time the packet arrives at the network driver. You'd be wrong. I'd make fun of you for that, but it took me a while to figure it out, too.

The issue is that there are potentially multiple peers on a given if_ovpn interface (e.g., when it's acting as a server and has multiple clients). The kernel has figured out that the packet in question needs to go to one of them, but the kernel operates on the assumption that all these clients live on a single broadcast domain. That is, a packet sent on the interface would be visible to all of them. That's not the case here, so if_ovpn needs to work out which client the packet has to go to.

OpenVPN will install the new key using the OVPN_NEW_KEY command.

This is handled by **`ovpn_route_peer()`**. This function first looks through the list of peers to see if any peer's VPN address matches the destination address. (Done by **`ovpn_find_peer_by_ip()`** or **`ovpn_find_peer_by_ip6()`**, depending on address family). If a matching peer is found the packet is sent to this peer.

If not **`ovpn_route_peer()`** performs a route lookup, and repeats the peer lookup with the resulting gateway address.

Only when if_ovpn has figured out the peer to send the packet to can it be encrypted and transmitted.

Key Rotation

OpenVPN will from time to time change the key used to secure the tunnel. That's one of those hard jobs if_ovpn leaves to userspace, so some coordination between OpenVPN and if_ovpn is required.

OpenVPN will install the new key using the OVPN_NEW_KEY command. Each key has an ID, and every packet includes the key ID that was used to encrypt it. This means that during key rotation, all packets can still be decrypted, as both the old and new keys are known and kept active in the kernel.

Once the new key is installed, it can be made active using the OVPN_SWAP_KEYS command. That is, the new key will be used to encrypt outgoing packets.

Sometime later the old key can be deleted using the OVPN_DEL_KEY command.

vnet

Yes, we're going to have to talk about vnet. I'm writing this, it's inevitable.

I'm too lazy to explain it entirely, so I'll just point you at an article written by much better author Olivier Cochard-Labbé: "Jail: vnet by examples"[8].

Think of vnet as turning jails into virtual machines with their own IP stacks.

This isn't strictly required for the pfSense use case, but it makes testing much, much easier. It means we can test on a single machine, without needing any external tools (other than OpenVPN itself, for what should be pretty obvious reasons).

For those interested in how this is done there's another FreeBSD Journal article that might be useful: "The Automated Testing Framework," by ... wait, I think I know that guy, Kristof Provost.

Performance

After all of that, I bet you're asking yourself "Does this actually help though?"

Well, fortunately for me: yes, yes it does.

One of my colleagues at Netgate spent some time gently teasing a Netgate 4100[10] device with iperf3 and got these results:

`if_tun`	`207.3 Mbit/s`
`DCO Software`	`213.1 Mbit/s`
`DCO AES-NI`	`751.2 Mbit/s`
`DCO QAT`	`1,064.8 Mbit/s`

"if_tun" is the old OpenVPN approach without DCO. It's worth noting that it used AES-NI instructions in userspace, and the 'DCO software' setup did not. Despite this blatant attempt at cheating, DCO was still slightly faster. On a level playing field (i.e., where DCO does use AES-NI instructions) there's no contest. DCO is more than three times faster.

There's some good news for Intel too: their QuickAssist offload engine is even faster than AES-NI, making OpenVPN five times faster than it was previously.

Future Work

Nothing is so good that it cannot be improved, but in some ways this next enhancement is a result of the success of DCO's design.

The on-wire OpenVPN protocol uses a 32 bit initialization vector (IV), and for cryptographic reasons I won't explain here[11], it's a bad idea to re-use IVs with the same key.

That means that keys must be re-negotiated before we get to that point. OpenVPN's default renegotiation interval is 3600 seconds, and with a 30% margin for safety, that would translate to 2^32 * 0.7 / 3600, or about 835.000 packets per second. That's "only" 8 to 9 Gbit/s (assuming 1300 byte packets).

With DCO, that's already more or less within reach of contemporary hardware.

While it's a good problem to have, it's still a problem, so the OpenVPN developers are working on an updated packet format that will use 64-bit IVs.

Thanks

The if_ovpn work was sponsored by Rubicon Communications (trading as Netgate) for use with their pfSense product line. It's been in use there since the 22.05 pfSense plus release[12]. This work was upstreamed to FreeBSD and is part of the recent 14.0 release. It requires OpenVPN 2.6.0 or newer to use.

I'd also like to thank the OpenVPN developers, who were very welcoming when the initial FreeBSD patches turned up, and without whose assistance this project would not have gone anywhere near as well as it did.

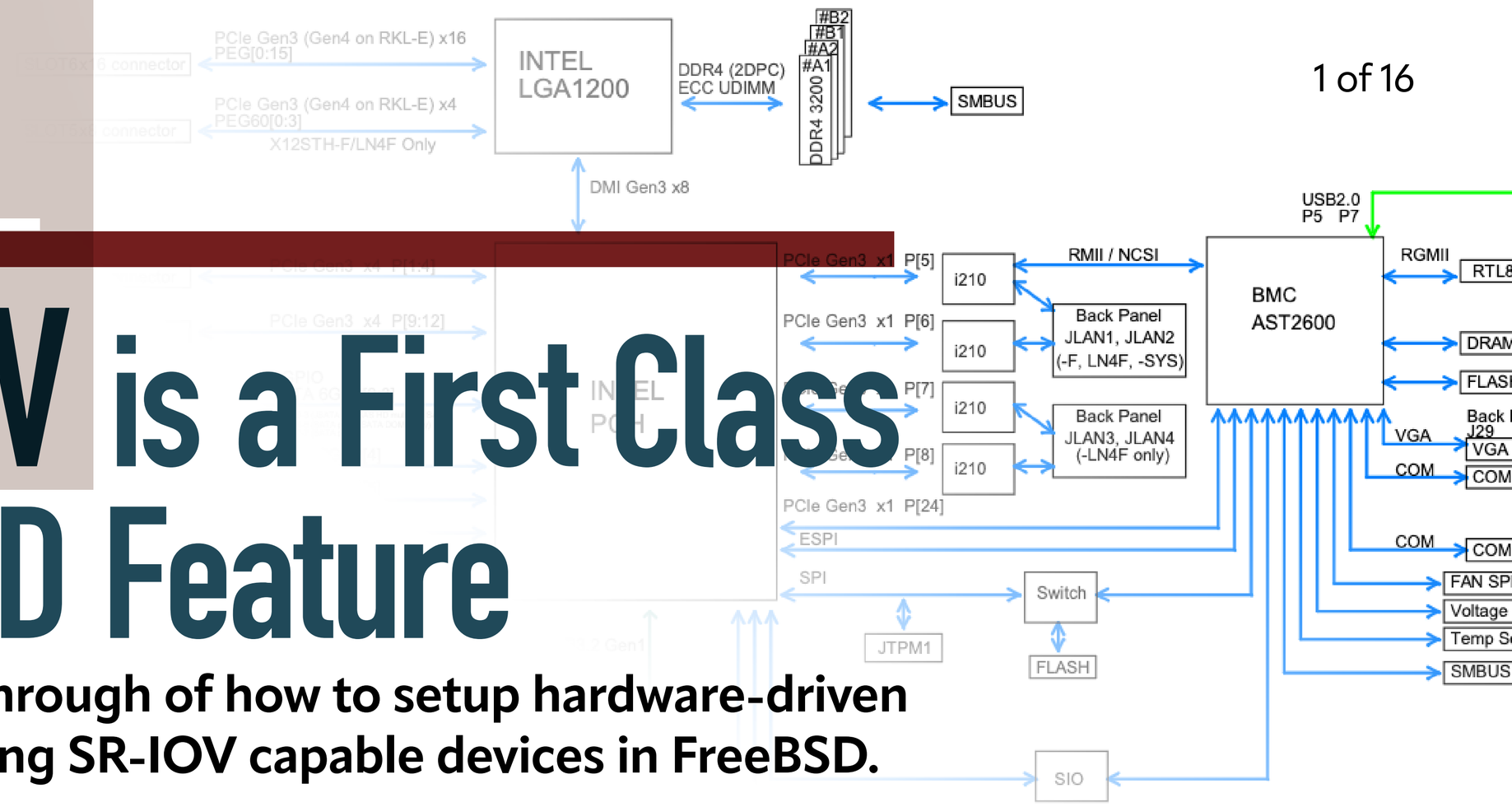
Footnotes:

1. Or. whenever you read this.
2. Fine. Writing. Reading. Look, if you're going to be pedantic about this, we'll be at this all day.
3. Look, if you're not interested in DCO you can just go read the next article. I'm sure it's very nice.
4. I say "we", but as much as I'd like to take credit for the solution it was the OpenVPN developers who came up with the DCO architecture and implemented it for Windows and Linux. All I did was what they did, but for FreeBSD.
5. In OpenVPN. DCO can be combined with the OS's traffic shaping (i.e. dummynet).
6. https://cgit.freebsd.org/src/tree/sys/net/if_ovpn.c?id=da69782bf-06645f38852a8b23af#n490
7. You might also say because the struct got fat. You might, I'm too polite for that.
8. https://freebsdfoundation.org/wp-content/uploads/2020/03/Jail-vnet-by-Examples.pdf
9. https://freebsdfoundation.org/wp-content/uploads/2019/05/The-Automated-Testing-Framework.pdf
10. https://shop.netgate.com/products/4100-base-pfsense
11. Mostly because I do not understand them myself.
12. https://www.netgate.com/blog/pfsense-plus-software-version-22.05-now-available

---

KRISTOF PROVOST is a freelance, embedded software engineer specializing in network and video applications. He's a FreeBSD committer, maintainer of the pf firewall in FreeBSD. He currently spends most of his time working on pfSense for Netgate.

Kristof has an unfortunate tendency to stumble into uClibc bugs, and a burning hatred for FTP. Do not talk to him about IPv6 fragmentation.



SR-IOV is a First Class FreeBSD Feature

A detailed walkthrough of how to setup hardware-driven virtualization using SR-IOV capable devices in FreeBSD.

BY MARK McBRIDE

One of my favorite hardware features is called Single-Root Input/Output Virtualization (SR-IOV). It makes a single physical device appear like multiple similar devices to the operating system. The FreeBSD approach to exposing SR-IOV capabilities is one of several reasons I tend to prefer FreeBSD on my servers.

SR-IOV for Networking Overview

Virtualization is a great solution if your demand for network devices exceeds the number of physical network ports on your server. There are many ways to accomplish this with software, but a hardware-based alternative is SR-IOV, which lets a single physical PCIe device to present itself as many to the OS.

There are several upsides to using SR-IOV. It offers the best performance compared to other means of virtualization. If you're a stickler for security, SR-IOV better isolates memory and the virtualized PCI devices it creates. It also results in a very tidy setup as everything is a PCI device, i.e., no virtual bridges, switches, etc.

To make use of SR-IOV networking, you'll need an SR-IOV capable network adapter and an SR-IOV capable motherboard. I've used several SR-IOV capable network cards over the years, such as the Intel i350-T4V2 Ethernet Adapter, the Mellanox ConnectX-4 Lx, and the Chelsio T520-SO-CR Fiber Network Adapter. For this article, I'll be using an Intel X710-DA2 Fiber Network Adapter (product brief) in a FreeBSD 14.0-RELEASE server. It's a nice option as it requires no special firmware configuration and driver support is built into the FreeBSD kernel by default. And as a bonus, it uses a fraction of the power of many alternatives, maxing out at only 3.7 Watts.

The Intel X710-DA2 PCIe 3.0 Fiber Network Adapter

The X710-DA2 has two physical SFP+ fiber ports. In SR-IOV terms, these correspond to physical functions (PFs). Without SR-IOV enabled, the PFs behave like the ports on any network adapter card and will show up as two network interfaces in FreeBSD. With SR-IOV enabled, each PF is capable of creating, configuring, and managing several Virtual Functions (VFs). Each VF will appear in the OS as a PCIe device.

In the case of the X710-DA2 specifically, its 2 PFs can virtualize up to 128 VFs. From the standpoint of FreeBSD, it's as if you have a network card with 128 ports. These VFs can then be allocated to jails and virtual machines for isolated networking.

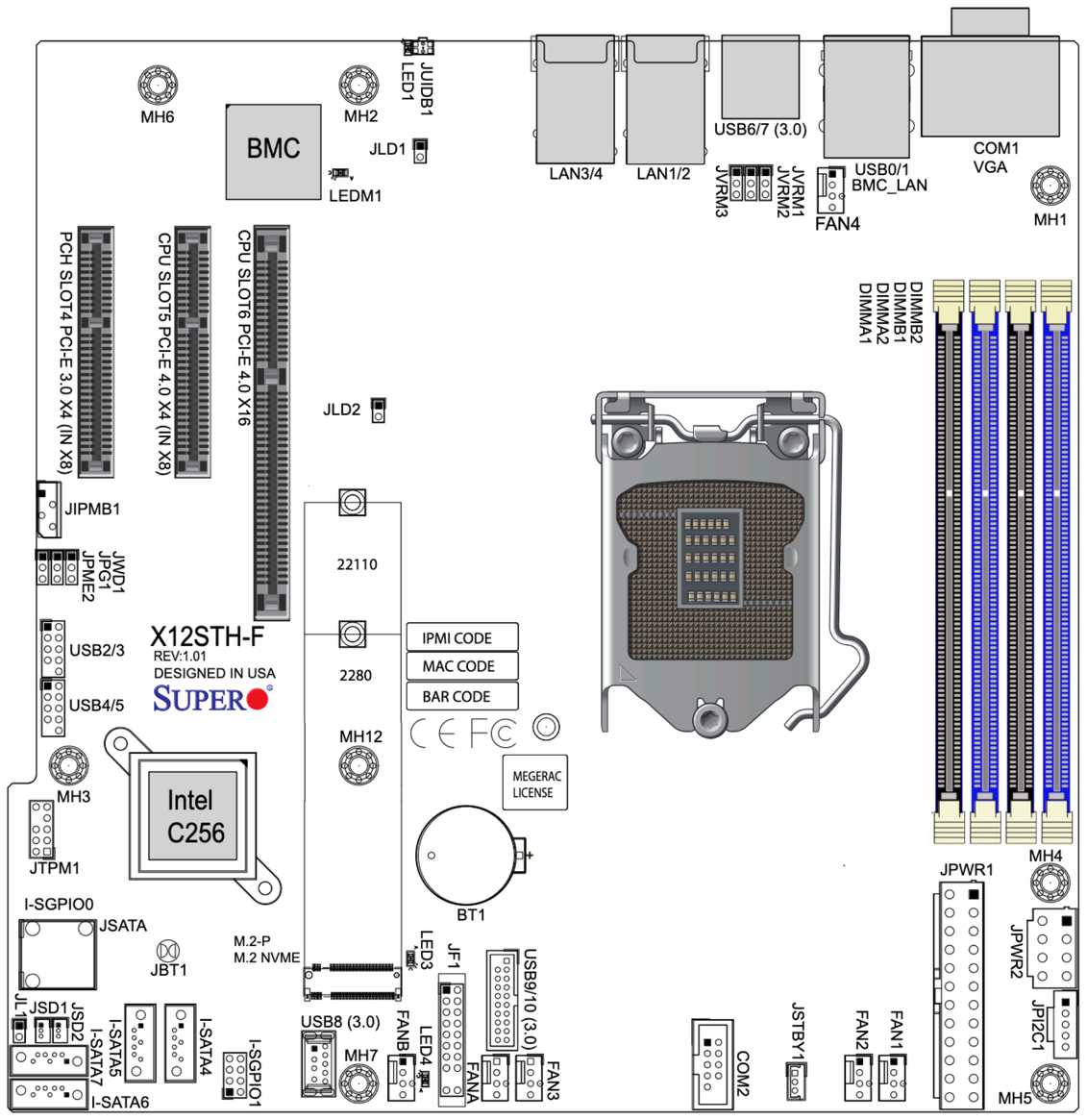
Using SR-IOV in FreeBSD

We touched a bit on how SR-IOV conceptually works, but I find it easier to understand with practical examples. Let's walk through setting up SR-IOV in FreeBSD from scratch. To do this, we'll focus on:

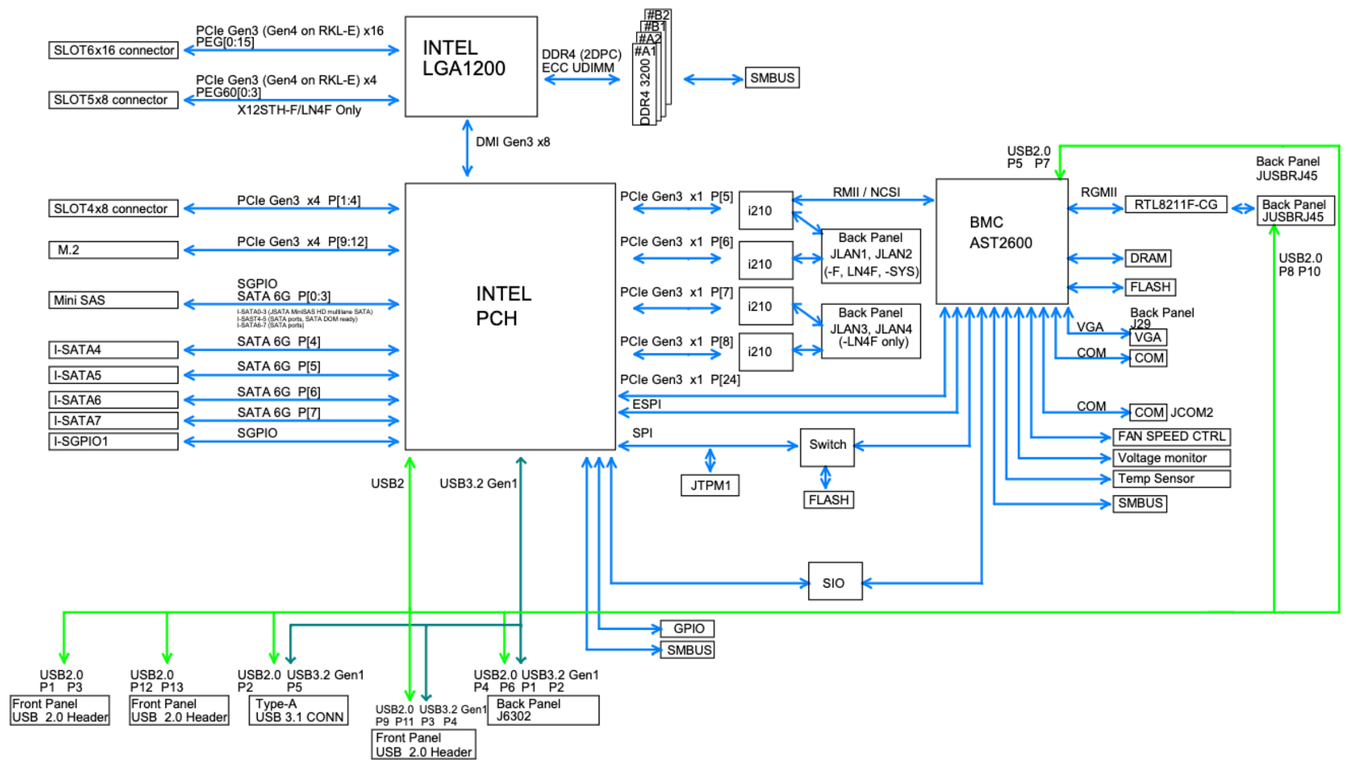
- Hardware Installation
- Hardware Configuration
- FreeBSD Configuration of SR-IOV
- Using an SR-IOV Network VF in a Jail
- Using an SR-IOV Network VF in a Bhyve Virtual Machine

Hardware Installation

The installation of the SR-IOV capable X710-DA2 is easy enough, but there is one major consideration. Not all PCIe slots on motherboards are created equally. I highly recommend you take a look at your motherboard's manual before getting started. For this example, I'll be using a Supermicro X12STH-F motherboard. The manual provides two insightful diagrams:

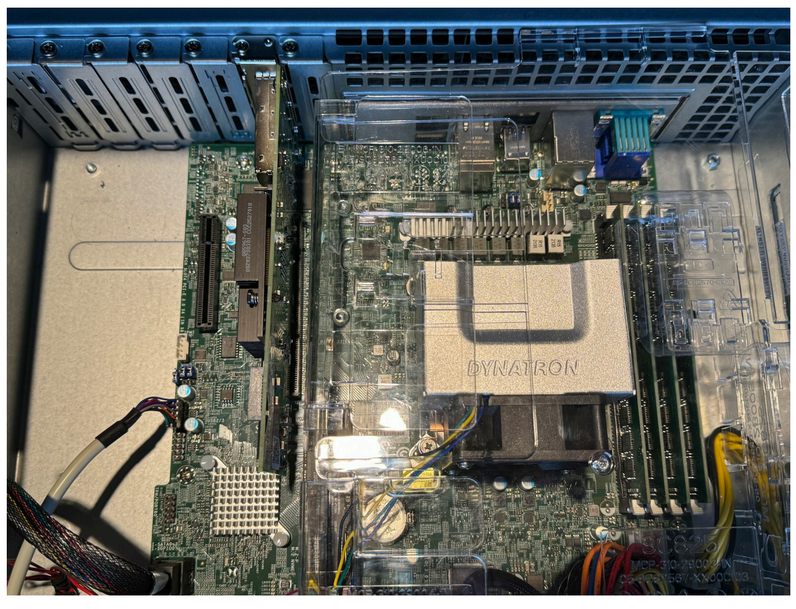


X12STH-F Motherboard Physical Map



X12STH-F Motherboard Block Map

In the first diagram we see our PCIe slots are numbered 4, 5, and 6, left-to-right. If you look closely, you'll see slot 4 has a "PCH" prefix while 5 and 6 have a "CPU" prefix. The block map diagram shows what this means in a bit more detail. Slots 5 and 6 connect directly to the CPU in the LGA1200 socket, while slot 4 connects to the Platform Controller Hub. Depending on the specific components in your system, this may determine which slots will allow SR-IOV to work as expected. There's no easy way to know until later when we configure FreeBSD, but as a rule of thumb, especially with older motherboards, I find the CPU slots to be a reliable choice. If you do find in later steps that SR-IOV is not working, try using a different PCIe slot. Motherboard documentation isn't always detailed, so trial and error is sometimes the quickest way to see which slot will work.



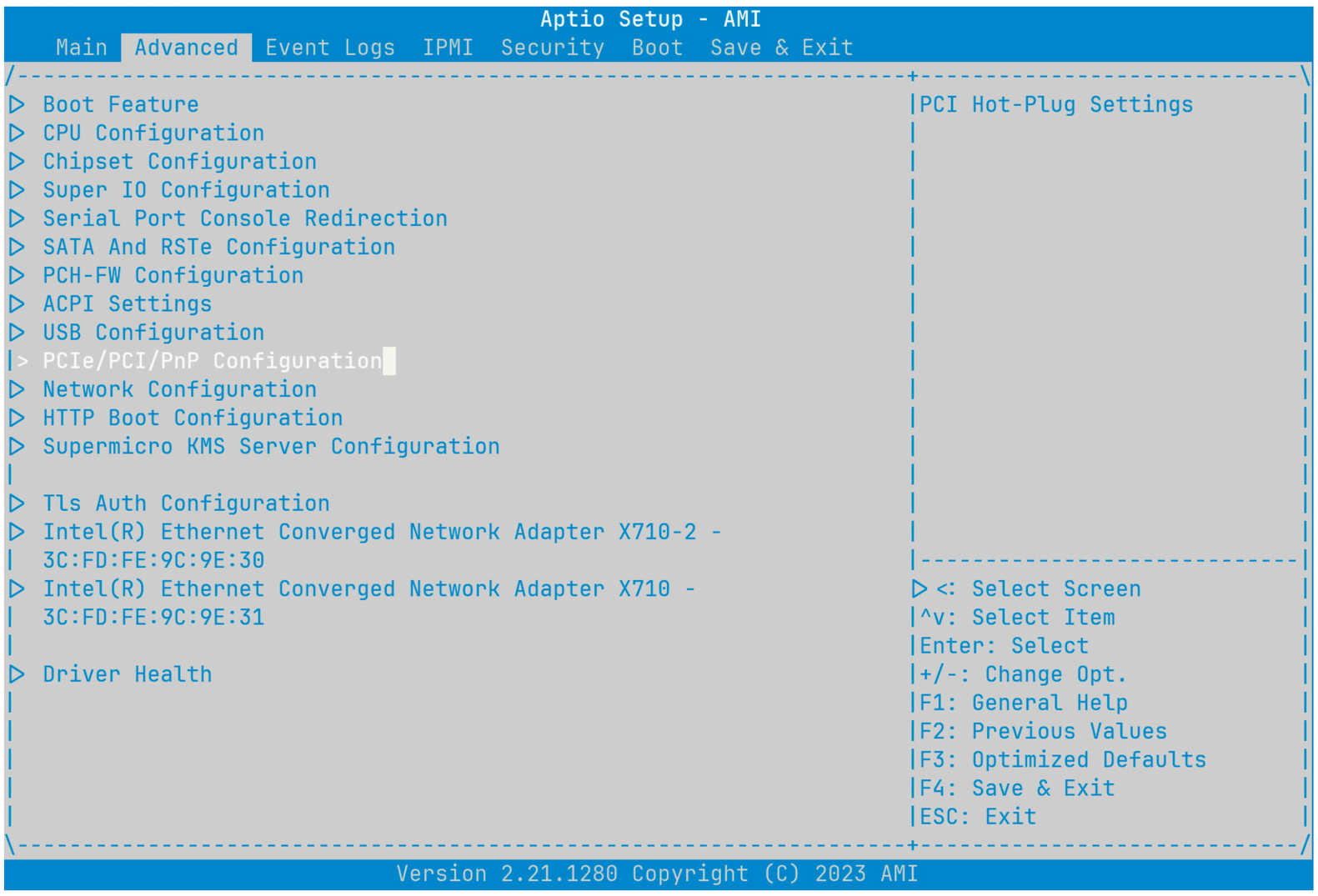
Supermicro X12STH-F Motherboard, CPU PCIe Slot 6 with Intel X710-DA2
(Also: Intel Xeon E-2324G w/ 4x8GB ECC UDIMM in a Supermicro 825TQC-R740LPB 2U Chassis)

Intel X710-DA2 SFP+ Ports with DAC Cables Attached

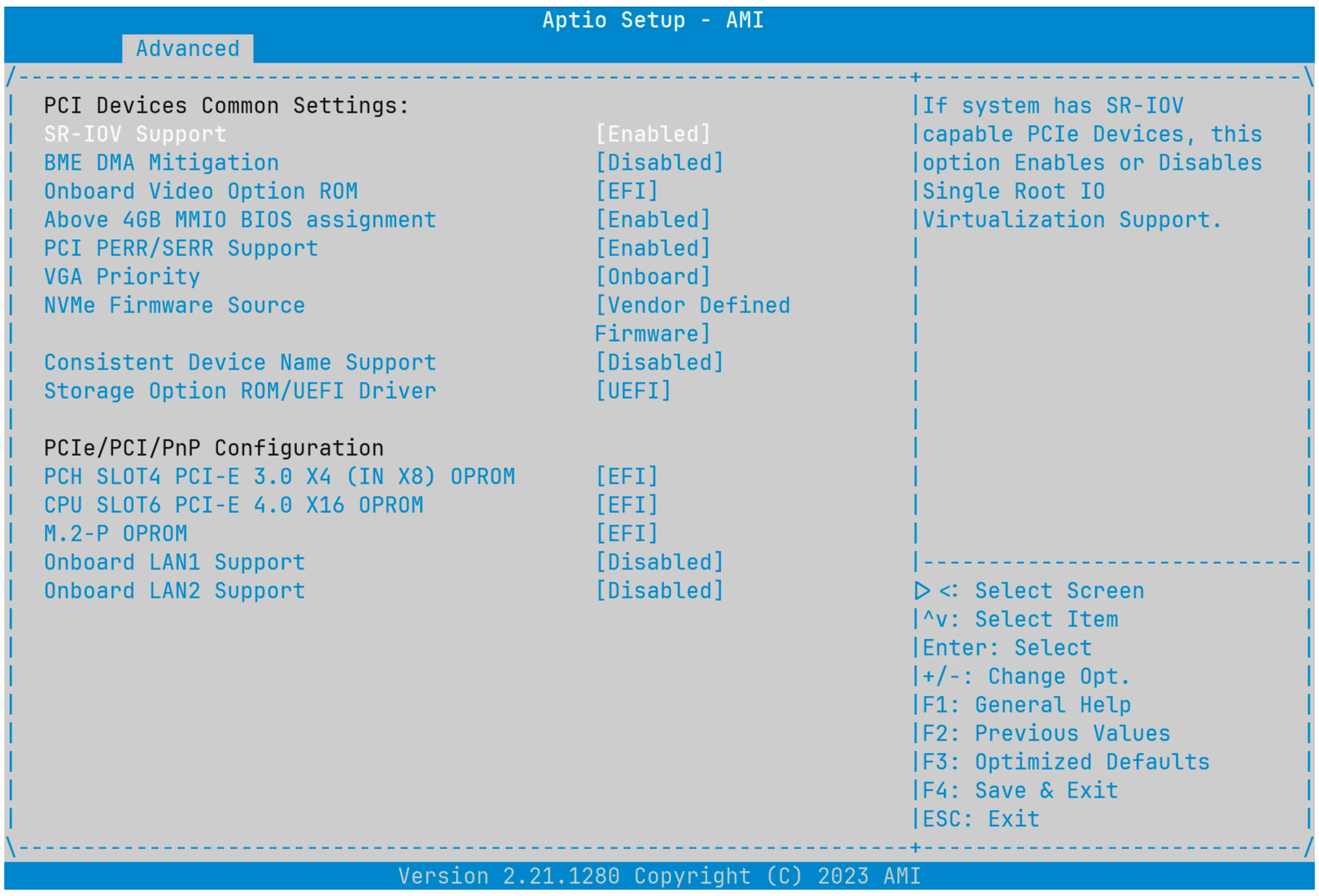
Hardware Configuration

The X710-DA2 will behave like a non-SR-IOV capable card until you enable SR-IOV in your motherboard settings. It's easy to do, but also quite easy to forget, so be sure you don't skip this important step.

The exact procedure will vary by motherboard, but most will have a screen with PCIe configuration options. Find that screen and enable SR-IOV. While you're there it's a good idea to check other settings are enabled that you're likely to use in conjunction with SR-IOV, like CPU virtualization.



X12STH-F Motherboard Setup, PCIe Configuration on Advanced Screen



X12STH-F Motherboard Setup, SR-IOV Configuration on PCIe Screen

We can now boot FreeBSD and take a look at dmesg(8). Here's a snippet from mine.

```
ixl0: <Intel(R) Ethernet Controller X710 for 10GbE SFP+ - 2.3.3-k> mem
      0x6000800000-0x6000ffffff,0x6001808000-0x600180ffff irq 16 at device 0.0 on pci1
ixl0: fw 9.120.73026 api 1.15 nvm 9.20 etid 8000d87f oem 1.269.0
ixl0: PF-ID[0]: VFs 64, MSI-X 129, VF MSI-X 5, QPs 768, I2C
ixl0: Using 1024 TX descriptors and 1024 RX descriptors
ixl0: Using 4 RX queues 4 TX queues
ixl0: Using MSI-X interrupts with 5 vectors
ixl0: Ethernet address: 3c:fd:fe:9c:9e:30
ixl0: Allocating 4 queues for PF LAN VSI; 4 queues active
ixl0: PCI Express Bus: Speed 2.5GT/s Width x8
ixl0: SR-IOV ready
ixl0: netmap queues/slots: TX 4/1024, RX 4/1024
```

On the third line we see some SR-IOV references. "PF-ID[0]" is associated with ixl0, and this PF is capable of 64 VFs. And on the tenth line we get a nice confirmation that this PCIe device is "SR-IOV ready." The reason for the "ixl" name is that this card uses the ixl(4) Intel Ethernet 700 Series Driver.

There's nothing else you need to do to configure the X710-DA2's hardware. Some cards (like the aforementioned Mellanox) require you to configure the card's firmware, while other cards (like the aforementioned Chelsio) require driver configuration in `/boot/loader.conf`. Neither is needed with the X710-DA2, though you may want to check the card's firmware version and update it if necessary.

With this, we're ready to shift our focus from hardware setup to FreeBSD configuration.

FreeBSD Configuration of SR-IOV

Using PFs

A nice thing about SR-IOV is regardless of whether or not you tell a PF to create VFs you can still use the PF as a network interface. I'll add the following to my `/etc/rc.conf` and assign an IP address to the PF for use in the host.

```
ifconfig_ixl0="inet 10.0.1.201 netmask 255.255.255.0"
defaultrouter="10.0.1.1"
```

Now when I boot the system, I can expect the ixl0 device to have an IP address that I can use to connect to the system regardless of whether SR-IOV is enabled or not.

Telling PFs to Create VFs

Management of PFs and VFs in FreeBSD is handled by iovctl(8), which is included in the base OS. To create VFs, we need to create a file in the `/etc/iov/` directory with some specifics of what we want. We will execute a simple strategy and create one VF to assign to a jail, and a second for a bhyve virtual machine. The iovctl.conf(5) manual page will give us the most important parameters.

```
OPTIONS
     The following parameters are accepted by all PF drivers:

     device (string)
             This parameter specifies the name of the PF device.  This
             parameter is required to be specified.

     num_vfs (uint16_t)
             This parameter specifies the number of VF children to create.
             This parameter may not be zero.  The maximum value of this
             parameter is device-specific.
```

I like to set num_vfs to what I need. We could set it to the max, but I find it makes looking at ifconfig and other command output more difficult.

Additionally, as different cards have different drivers, each driver has options you can set based on the hardware capability. The ixl(4) manual page lists several optional parameters.

```
IOVCTL OPTIONS
     The driver supports additional optional parameters for created VFs
     (Virtual Functions) when using iovctl(8):

     mac-addr (unicast-mac)
             Set the Ethernet MAC address that the VF will use.  If
             unspecified, the VF will use a randomly generated MAC address.
```

Or, alternatively, you can use the iovctl command for a terse summary of what parameters are valid for a PF and its VFs, and what their defaults are.

```
(host) $ sudo iovctl -S -d ixl0
The following configuration parameters may be configured on the PF:
        num_vfs : uint16_t (required)
        device : string (required)

The following configuration parameters may be configured on a VF:
        passthrough : bool (default = false)
        mac-addr : unicast-mac (optional)
        mac-anti-spoof : bool (default = true)
        allow-set-mac : bool (default = false)
        allow-promisc : bool (default = false)
        num-queues : uint16_t (default = 4)
```

We'll make use of the **`mac-addr`** parameter to set specific MAC addresses for each VF. Setting the MAC address is a bit arbitrary in this case, but I'll do it to demonstrate how a config file looks with PF parameters, default VF parameters, and parameters specific to individual VFs.

```
PF {
        device : "ixl0"
        num_vfs : 2
}

DEFAULT {
        allow-set-mac : true;
}

VF-0 {
        mac-addr : "aa:88:44:00:02:00";
}

VF-1 {
        mac-addr : "aa:88:44:00:02:01";
}
```

This instructs ixl0 to create two VFs. By default, every VF will be allowed to set it's own MAC. And each VF will have an initial MAC address assigned to it (which can be overridden with the previous default setting).

Before we make it effective, let's take a look at our current environment. We'll find two ixl PCI devices, and two ixl network interfaces.

```
(host) $ ifconfig -l
ixl0 ixl1 lo0

(host) $ pciconf -lv | grep -e ixl -e iavf -A4
ixl0@pci0:1:0:0:        class=0x020000 rev=0x01 hdr=0x00 vendor=0x8086
device=0x1572 subvendor=0x8086 subdevice=0x0007
    vendor     = 'Intel Corporation'
    device     = 'Ethernet Controller X710 for 10GbE SFP+'
    class      = network
    subclass   = ethernet
ixl1@pci0:1:0:1:        class=0x020000 rev=0x01 hdr=0x00 vendor=0x8086
device=0x1572 subvendor=0x8086 subdevice=0x0000
    vendor     = 'Intel Corporation'
    device     = 'Ethernet Controller X710 for 10GbE SFP+'
    class      = network
    subclass   = ethernet
```

To make our /etc/iov/ixl0.conf configuration effective, we use iovctl(8).

```
(host) $ sudo iovctl -C -f /etc/iov/ixl0.conf
```

Should you change your config file, delete and recreate the VFs.

```
(host) $ sudo iovctl -D -f /etc/iov/ixl0.conf
(host) $ sudo iovctl -C -f /etc/iov/ixl0.conf
```

To check that it worked, let's run the same ifconfig and pciconf commands from before.

```
(host) $ ifconfig -l
ixl0 ixl1 lo0 iavf0 iavf1

(host) $ pciconf -lv | grep -e ixl -e iavf -A4
ixl0@pci0:1:0:0:        class=0x020000 rev=0x01 hdr=0x00 vendor=0x8086 device=0x1572 subvendor=0x8086 subdevice=0x0007
    vendor     = 'Intel Corporation'
    device     = 'Ethernet Controller X710 for 10GbE SFP+'
    class      = network
    subclass   = ethernet
ixl1@pci0:1:0:1:        class=0x020000 rev=0x01 hdr=0x00 vendor=0x8086 device=0x1572 subvendor=0x8086 subdevice=0x0000
    vendor     = 'Intel Corporation'
    device     = 'Ethernet Controller X710 for 10GbE SFP+'
    class      = network
    subclass   = ethernet
--
iavf0@pci0:1:0:16:      class=0x020000 rev=0x01 hdr=0x00 vendor=0x8086 device=0x154c subvendor=0x8086 subdevice=0x0000
    vendor     = 'Intel Corporation'
    device     = 'Ethernet Virtual Function 700 Series'
    class      = network
    subclass   = ethernet
iavf1@pci0:1:0:17:      class=0x020000 rev=0x01 hdr=0x00 vendor=0x8086 device=0x154c subvendor=0x8086 subdevice=0x0000
    vendor     = 'Intel Corporation'
    device     = 'Ethernet Virtual Function 700 Series'
    class      = network
    subclass   = ethernet
```

Voilà! Our shiny new VFs have arrived. In the `pciconf` output we still see our ixl devices, but now there are two iavf devices. The iavf(4) manual page let's us know that this is the driver for Intel Adaptive Virtual Functions.

In addition to seeing new PCI devices, ifconfig confirms that they are indeed recognized as network interfaces. For the most common aspects of a network device, you'll probably not be able to tell the difference between a PF and VF. If you want to get into the details and differences, check out the driver documentation and the -c capabilities flag of pciconf, e.g. **`pciconf -lc iavf`**.

To make this config persistent across reboots, amend your /etc/rc.conf file.

```
# Configure SR-IOV
iovctl_files="/etc/iov/ixl0.conf"
```

Now we've got two VFs ready for action. Let's put them to use!

Using an SR-IOV Network VF in a Jail

This section assumes you have a basic understanding of FreeBSD Jails. As such, setting up a jail from scratch is out of scope. For more information how to do this, see the Jails and Containers chapter of the FreeBSD Handbook.

I don't use any jail management ports and rely on what come in the base OS. If you've used something like Bastille, the specifics on how/where to put your configs might vary a bit, but the concept is the same. In this example we're working with a jail named "desk."

```
exec.start += "/bin/sh /etc/rc";
exec.stop = "/bin/sh /etc/rc.shutdown";
exec.clean;
mount.devfs;

desk {
        host.hostname = "desk";
        path = "/mnt/apps/jails/desk";
        vnet;
        vnet.interface = "iavf0";
        devfs_ruleset="5";
        allow.raw_sockets;
}
```

That's it! The jail now has access to its own dedicated VF network device setup via vnet(9). I'll tweak the jail's /etc/rc.conf file to enable it.

```
ifconfig_iavf0="inet 10.0.1.231 netmask 255.255.255.0"
defaultrouter="10.0.1.1"
```

Now let's start the jail and check that it works.

```
(host) $ sudo service jail start desk
Starting jails: desk.

(host) $ sudo jexec desk ifconfig iavf0
iavf0: flags=1008843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST,LOWER_UP> metric 0 mtu 1500
        options=4e507bb<RXCSUM,TXCSUM,VLAN_MTU,VLAN_HWTAGGING,JUMBO_MTU,VLAN_HWCSUM,TSO4,
TSO6,LRO,VLAN_HWFILTER,VLAN_HWTSO,RXCSUM_IPV6,TXCSUM_IPV6,HWSTATS,MEXTPG>
        ether aa:88:44:00:02:00
10.0.1.231 netmask 0xffffff00 broadcast 10.0.1.255
        media: Ethernet autoselect (10Gbase-SR <full-duplex>)
        status: active
        nd6 options=29<PERFORMNUD,IFDISABLED,AUTO_LINKLOCAL>

(host) $ sudo jexec desk ping 9.9.9.9
PING 9.9.9.9 (9.9.9.9): 56 data bytes
64 bytes from 9.9.9.9: icmp_seq=0 ttl=58 time=19.375 ms
64 bytes from 9.9.9.9: icmp_seq=1 ttl=58 time=19.809 ms
64 bytes from 9.9.9.9: icmp_seq=2 ttl=58 time=19.963 ms
```

As expected, we see the iavf0 interface in the jail and it appears to be working normally. But what about that device in the host OS? Is it still there? Let's check.

```
(host) $ ifconfig -l
ixl0 ixl1 lo0 iavf1
```

As expected, the iavf0 interface is no longer visible to the host OS. You'll still see the PCI device with pciconf, but will not be able to do anything with it. The jail is in full control of this device. If you stop the jail, the iavf0 device will return to the host OS and once again be present in ifconfig output.

Using an SR-IOV Network VF in a Bhyve Virtual Machine

You can achieve a similar result with bhyve(8) virtual machines, though the approach is a bit different. With jails we can assign/release VFs during runtime. With bhyve, this must be done at boot time and requires a tweak to our SR-IOV config. First, let's have a look again at pciconf before we change anything.

```
(host) $ pciconf -l | grep iavf
iavf0@pci0:1:0:16:      class=0x020000 rev=0x01 hdr=0x00 vendor=0x8086 device=0x154c
subvendor=0x8086 subdevice=0x0000
iavf1@pci0:1:0:17:      class=0x020000 rev=0x01 hdr=0x00 vendor=0x8086 device=0x154c
subvendor=0x8086 subdevice=0x0000
```

Look at the unused VF, iavf1. The first column can be read as "there's a PCI0 device using the driver iavf, ID 1, with a PCI selector of bus 1, slot 0, function 17". While you don't need them yet, those last three numbers are how we'll eventually tell bhyve which device to use. Before we get to that, let's ensure we load vmm(4) at boot time to enable bhyve, and tweak our second VF so that it's ready for passthrough to bhyve.

```
## Load the virtual machine monitor, the kernel portion of bhyve
vmm_load="YES"

# Another way to passthrough a VF, or any PCI device, is to
# specify the device in /boot/loader.conf. I show this for reference.
# We'll use our iovctl config instead as it keeps things in one place.
# pptdevs="1/0/17"
```

To reserve the VF for passthrough to bhyve, we use the iovctl passthrough parameter.

```
    passthrough (boolean)
            This parameter controls whether the VF is reserved for the use of
            the bhyve(8) hypervisor as a PCI passthrough device.  If this
            parameter is set to true, then the VF will be reserved as a PCI
            passthrough device and it will not be accessible from the host
            OS.  The default value of this parameter is false.
```

```
PF {
        device : "ixl0"
        num_vfs : 2
}

DEFAULT {
        allow-set-mac : true;
}

VF-0 {
        mac-addr : "aa:88:44:00:02:00";
}

VF-1 {
        mac-addr : "aa:88:44:00:02:01";
        passthrough : true;
}
```

When we next boot our system, we'll find iavf1 absent because the iavf driver will never get assigned to our second VF. Instead it will get marked "ppt" for "PCI passthrough" and only bhyve will be able to make use of it.

With those tweaks, reboot.

Right away you'll notice dmesg output is different. There is no mention of iavf1 this time. And remember the `1:0:17` selector we saw in pciconf? We see it here with a slightly different format.

```
ppt0 at device 0.17 on pci1
```

`pciconf` confirms that the device is reserved for passthrough.

```
(host) $ pciconf -l | grep iavf
iavf0@pci0:1:0:16:      class=0x020000 rev=0x01 hdr=0x00 vendor=0x8086 device=0x154c subvendor=0x8086 subdevice=0x0000

(host) $ pciconf -l | grep ppt
ppt0@pci0:1:0:17:       class=0x020000 rev=0x01 hdr=0x00 vendor=0x8086 device=0x154c subvendor=0x8086 subdevice=0x0000
```

The rest we do in bhyve. This article assumes you know how to get a bhyve virtual machine up and running. I use the [vm-bhyve](https://man.freebsd.org/cgi/man.cgi?query=vm) tool for easy management of virtual machines (but see the end of this section for raw bhyve parameters if you don't use vm-bhyve). I'll add the ppt VF to a Debian VM called debian-test. All we need to do is define the device we want to passthrough in the config and remove any lines pertaining to virtual networking.

```
loader="grub"
cpu=1
memory=4G
disk0_type="virtio-blk"
disk0_name="disk0.img"
uuid="b997a425-80d3-11ee-a522-00074336bc80"
```

```
# Passthrough a VF for Networking
passthru0="1/0/17"

# Common defaults that are not needed with a VF available
# network0_type="virtio-net"
# network0_switch="public"
# network0_mac="58:9c:fc:0c:fd:b7"
```

All we have to do now is start our bhyve virtual machine.

```
(host) $ sudo vm start debian-test
Starting debian-test
  * found guest in /mnt/apps/bhyve/debian-test
  * booting...

(host) $ sudo vm console debian-test
Connected

debian-test login: root
Password:
Linux debian-test 6.1.0-16-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.1.67-1 (2023-12-12) x86_64

root@debian-test:~# lspci | grep -i intel
00:05.0 Ethernet controller: Intel Corporation Ethernet Virtual Function 700 Series
(rev 01)

root@debian-test:~# ip addr
2: enp0s5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether aa:88:44:00:02:01 brd ff:ff:ff:ff:ff:ff
    inet 10.0.1.99/24 brd 10.0.1.255 scope global dynamic enp0s5
       valid_lft 7186sec preferred_lft 7186sec
    inet6 fdd5:c1fa:4193:245:a888:44ff:fe00:201/64 scope global dynamic mngtmpaddr
       valid_lft 1795sec preferred_lft 1795sec
    inet6 fe80::a888:44ff:fe00:201/64 scope link
       valid_lft forever preferred_lft forever

root@debian-test:~# ping 9.9.9.9
PING 9.9.9.9 (9.9.9.9) 56(84) bytes of data.
64 bytes from 9.9.9.9: icmp_seq=1 ttl=58 time=20.6 ms
64 bytes from 9.9.9.9: icmp_seq=2 ttl=58 time=19.8 ms
```

Success! We now have an SR-IOV VF device for networking in our bhyve VM.

If you're a purist and don't want to use **`vm-bhyve`**, details are appended to a`**`vm-bhyve.log`** file when you use a **`vm`** command. In it you will see the parameters that were passed to **`grub-bhyve`** and **`bhyve`** to start the VM.

```
create file /mnt/apps/bhyve/debian-test/device.map
      -> (hd0) /mnt/apps/bhyve/debian-test/disk0.img
grub-bhyve -c /dev/nmdm-debian-test.1A -S \
      -m /mnt/apps/bhyve/debian-test/device.map \
      -M 4G -r hd0,1 debian-test
bhyve -c 1 -m 4G -AHP
      -U b997a425-80d3-11ee-a522-00074336bc80 -u -S \
      -s 0,hostbridge -s 31,lpc \
      -s 4:0,virtio-blk,/mnt/apps/bhyve/debian-test/disk0.img \
      -s 5:0,passthru,1/0/17
```

bhyve PCI passthrough is an emerging feature

While using VFs with vnet for jails is very stable, bhyve PCI passthrough in general is still under heavy development as of 14.0-RELEASE. Using byhve with passthrough alone works great. However, I have found that if I am also using VFs with jails, certain hardware combinations and volumes of devices can create unexpected behavior. Improvements land with each release. If you find an edge case, be sure to submit a bug.

FreeBSD SR-IOV in Summary

To make use of SR-IOV enabled virtual PCIe devices in FreeBSD, we:

- install an SR-IOV capable network card onto an SR-IOV capable motherboard
- ensure the motherboard's SR-IOV feature is enabled
- create /etc/iov/ixl0.conf and specify how many VFs we want
- reference /etc/iov/ixl0.conf in /etc/rc.conf to persists across boots

And that's it!

To demonstrate that it worked, we allocated one VF to a jail using vnet. And we pre-allocated another VF at boot-time for passthrough to bhyve virtual machines. In both cases, all we had to do was put a few lines in the respective jail/VM config files.

The following section will contrast the FreeBSD approach compared to what you'll find in Linux distributions to give you a feel how the two approaches vary.

SR-IOV in Linux

SR-IOV works really well in Linux. Once you've got it all setup, you likely won't be able to find discernible differences between FreeBSD and Linux. Getting it all setup, however, can be a bit of a journey.

The biggest difference is there is no standard tool like FreeBSD's iovctl for setting up SR-IOV in Linux. There are several ways to achieve a working setup, but they are not so obvious. I'll highlight how I use `udev` to setup a Mellanox card's PF and VFs.

`udev` is a powerful tool that does a lot of stuff. One of the things it can do is enable SR-IOV devices at boot time. The tool itself is excellent, but knowing what data to feed it is where the challenge lies. Getting the attributes you need will likely require a bit of searching on the Internet, but once you have them the resulting `udev` rules are very simple.

```
# DO NOT Probe VFs that will be used for VMs
KERNEL=="0000:05:00.0", SUBSYSTEM=="pci", ATTRS{vendor}=="0x15b3", ATTRS{device}=="0x1015",
ATTR{sriov_drivers_autoprobe}="0", ATTR{sriov_numvfs}="4"

# DO Probe VFs that will be used for LXD
KERNEL=="0000:05:00.1", SUBSYSTEM=="pci", ATTRS{vendor}=="0x15b3", ATTRS{device}=="0x1015",
ATTR{sriov_drivers_autoprobe}="1", ATTR{sriov_numvfs}="16"
```

That essentially says, "match the PCI device 0000:05:00.0 with vendor ID 0x15b3 and device ID 0x1015, and for that device do not try to automatically assign a driver and create 4 VFs" (i.e., reserve for passthrough). The second rule is similar, but targets a different PF, does assign a driver, and creates 16 VFs (i.e., ready for container allocation).

Depending on the card and specific Linux distribution you're using, those may not be all the attributes you need. For example, if you're using Fedora you may need to add **`ENV{NM_UNMANAGED}="1"`** to avoid NetworkManager taking control of your VFs at boot time.

Similar to **`pciconf`**, **`lspci`** will get us much of what we need for the matching parts of those rules, which is the PCI address, vendor and device ID. In this system we can see that we have Mellanox ConnectX-4 Lx card.

```
lspci -nn | grep ConnectX
05:00.0 Ethernet controller [0200]: Mellanox Technologies MT27710 Family [ConnectX-4 Lx] [15b3:1015]
05:00.1 Ethernet controller [0200]: Mellanox Technologies MT27710 Family [ConnectX-4 Lx] [15b3:1015]
```

The attributes set by **`udev`** are visible in **`/sys/bus/pci/devices/0000:05:00.*/`** with many others. Listing the contents of that directory is a good place to go looking for things to tell **udev**.

```
(linux) $ ls -AC /sys/bus/pci/devices/0000:05:00.0/
aer_dev_correctable       device              irq              net              resource0              subsystem
aer_dev_fatal             dma_mask_bits       link             numa_node        resource0_wc           subsystem_device
aer_dev_nonfatal          driver              local_cpulist    pools            revision               subsystem_vendor
ari_enabled               driver_override     local_cpus       power            rom                    uevent
broken_parity_status      enable              max_link_speed   power_state      sriov_drivers_autoprobe vendor
class                     firmware_node       max_link_width   ptp              sriov_numvfs           virtfn0
config                    hwmon               mlx5_core.eth.0  remove           sriov_offset           virtfn1
consistent_dma_mask_bits  infiniband          mlx5_core.rdma.0 rescan           sriov_stride           virtfn2
current_link_speed        infiniband_verbs    modalias         reset            sriov_totalvfs         virtfn3
current_link_width        iommu               msi_bus          reset_method     sriov_vf_device        vpd
d3cold_allowed            iommu_group         msi_irqs         resource         sriov_vf_total_msix
```

In that listing, we see our two udev targets, **`sriov_drivers_autoprobe`** and **`sriov_numvfs`**, which we want to set at boot time. What does everything else do? You'll probably need your favorite search engine to answer that question.

With udev we've accomplished step 1 of 2 major steps. It effectively "turns on" the hardware SR-IOV capability. We need to still need to configure it for networking use, which is major step 2. This varies a great deal depending on whatever we're using to manage networking. For example, if you use systemd-networkd, you'd do something like this.

```
#/etc/systemd/network/21-wired-sriov-p1.network
[Match]
Name=enp5s0f1np1

[SR-IOV]
VirtualFunction=0
Trust=true

[SR-IOV]
VirtualFunction=1
Trust=true
```

Luckily, for systemd-networkd, the documentation isn't so bad and you can find most of what you need. With that, we restart the service and the VFs are ready to use.

But not all documents are great, and aside from the networking software itself, security overlays like AppArmor and selinux can create hard to detect blockers that are technically doing what they're supposed to do, but will very much make the system feel like it's not functioning.

As a specific example of frustration, I was recently using Fedora 39 to run a handful of LXD containers. I found notes to set `ENV{NM_UNMANAGED}="1"` in udev and that did the trick to let LXD manage my VFs. Everything worked fine until I rebooted the containers several times. Suddenly LXD started complaining that there were no VFs.

It turns out that while the `udev` rule stopped NetworkManager from managing VFs at boot time, NetworkManager was intercepting them at runtime when containers were restarting and taking over management of them. I realized something strange was happening because VF device names were changing after restarting containers. For example, what started as `enp5s0f0np0` would become something like `physZqHm0g` once the container it was assigned to restarted.

Eventually, I was able to find a way to tell NetworkManager not to do this. The critical config file I had to create to stop the LXD+NM battle is below, just in case you were wondering.

```
[keyfile]
unmanaged-devices=interface-name:enp5s0f1*,interface-name:phys*
```

This is just one example. Thinking you have everything working only to find out days later things are actually slowly self-destructing is not a good experience. In general, I find all frustrations have the same root cause: no existing or emerging standard way to configure SR-IOV in the Linux ecosystem. Once you get over the not-so-obvious setup hurdles, SR-IOV for networking with Linux works just fine.

Conclusion

SR-IOV is a first class citizen in FreeBSD. Everything mentioned in this article you can find using the OS-provided manual pages. Start with a simple apropos(1) query.

```
(host) $ apropos "SR-IOV"
iovctl(8) - PCI SR-IOV configuration utility
```

The iovctl manual will get you started and the driver pages will give you the specifics for your hardware. When things are apparent and easy to find, system administration doesn't feeI like a chore.

Linux distributions are equally capable, but lacking in terms of cohesion and in-system documentation for SR-IOV. While I rely on Linux for all sorts of things, I truly appreciate the organization of configuration in FreeBSD. It's easy to come back to a system I haven't touched in a year and quickly understand what I've done. I far prefer this over taking detailed notes with obscure URLs to comments on discussion boards where some saint posted the way to make something work.

As with anything, make your own informed choice for what best suits your needs.

MARK McBRIDE works in CAR-T cell therapy in Seattle, Washington where he integrates supply chain, manufacturing, and patient operations solutions in a very new segment of personalized healthcare. In his free time, he enjoys over-engineering his garage homelab and cheering on all the local Seattle sports teams. Connect with him as @markmcb in #freebsd on the Libera IRC server, or via other means listed on his person site, markmcb.com.

Support FreeBSD®

Donate to the Foundation!

You already know that FreeBSD is an internationally recognized leader in providing a high-performance, secure, and stable operating system. It's because of you. Your donations have a direct impact on the Project.

Please consider making a gift to support FreeBSD for the coming year. It's only with your help that we can continue and increase our support to make FreeBSD the high-performance, secure, and reliable OS you know and love!

Your investment will help:

- Funding Projects to Advance FreeBSD
- Increasing Our FreeBSD Advocacy and Marketing Efforts
- Providing Additional Conference Resources and Travel Grants
- Continued Development of the FreeBSD Journal
- Protecting FreeBSD IP and Providing Legal Support to the Project
- Purchasing Hardware to Build and Improve FreeBSD Project Infrastructure

Making a donation is quick and easy.

freebsdfoundation.org/donate

FreeBSD™
FOUNDATION

FreeBSD Interface API (IfAPI)

BY JUSTIN HIBBITS

As some may be aware, Juniper uses its own custom network stack with FreeBSD—forked long ago—so it only superficially resembles the current FreeBSD network stack. There is state in the current FreeBSD stack that doesn't exist in Junos, and vise-versa.

The Why and How

Junos is big, very big. Updating FreeBSD is a monumental task. To make it easier, Juniper split the FreeBSD component out into its own separate repository, leaving Juniper enhancements in its separate repository. This causes a dilemma—how can we keep drivers in FreeBSD, but the netstack elsewhere? As part of the FreeBSD split project, the original DrvAPI was born. With this, drivers could exist in the FreeBSD repository, while keeping the Junos netstack separate.

But what is the netstack? Where do we draw the line between netstack and the rest? The initial approach was "all directories in sys/ prefixed **`net`**" which works well. However, recently the **`netlink`** component was added, which really isn't part of the network stack conceptually, so that was crossed out. Now the netstack consists of **`net`**, **`net80211`**, **`netgraph`**, **`netinet`**, **`netinet6`**, and **`netpfil`**. Keeping the details to only the network stack also hides the details from the core kernel. Some changes were needed for other parts of the kernel to accommodate the IfAPI, including NFS rootless (boot) and mbuf handling.

Juniper split the FreeBSD component out into its own separate repository.

Design

The current design of the IfAPI is simply accessors and iterators. This was chosen as the most expedient way to convert drivers and hide the **`struct ifnet`**, though it is far from the best way. Conversion was largely mechanical, and Juniper provided a shell script to handle the bulk of the conversion in **`tools/ifnet/convert_ifapi.sh`**. Obviously, this may miss some conversions, such as those where **`ifp`** is a member of another structure or is named something else like **`foo_ifp`**, but it does handle most cases.

As for iterators, the initial implementation was based upon Gleb Smirnoff's **`if_foreach_lladdr()`**, using callbacks when iterating over a given type. This was applied to both if_addr and if_t, where iterating over the interfaces only iterates over the current VNET. More re-

cently, a new iterator API was added, allowing iterating with a more traditional loop; the requirement being you must call `if_iter_finish()` or `ifa_iter_finish()` to clean up any state that set up for the iteration, including freeing any memory the implementation may have allocated (not done with the FreeBSD network stack, but could be done by other stacks).

Benefits

Decoupling device drivers from the network stack details brings benefits beyond Juniper's source code management. With a stable ABI, a single device driver can be used with multiple different network stacks. For instance, one image for all computers in a data center could select–at boot time—a different network stack based on the execution profile; a high-performance limited stack for some devices and a full stack for others, all using the same network drivers.

Another, smaller benefit is that driver changes and netstack changes can occur simultaneously without affecting one another. Before the IfAPI, any changes made to the `struct ifnet` required rebuilding all device drivers. Going forward, with the `struct ifnet` being completely private, any change to the structure requires rebuilding only the files that directly reference it, resulting in a shorter debug cycle.

With a stable ABI, a single device driver can be used with multiple different network stacks.

Where Next?

IfAPI is just step 1, there is still more to do to properly abstract away the network stack. Gleb Smirnoff proposed using KOBJ interfaces to allow a more pluggable netstack and fully decouple the netstack from the rest. This would even allow replacing the netstack at runtime (`kldunload`/`kldload`). Taking this further, we could potentially allow multiple netstacks, assigning different devices to different stacks. With that, there could even be the possibility of moving interfaces between netstacks dynamically.

Conclusion

The IfAPI is only phase one in an effort to decouple network drivers from the inner workings of the network stack. With further work, multiple network stacks could be in use—and even reloadable network stacks.

JUSTIN HIBBITS was foolishly entrusted with a FreeBSD commit bit in 2011 for his obsession with PowerPC. Since then, he's focused mostly on PowerPC and other embedded architectures. He currently works for Juniper Networks, working on all things FreeBSD kernel related, and continues his passion for low-level development and exotic architectures.

BATMAN
THE BETTER APPROACH TO MOBILE AD-HOC NETWORKS

BY AYMERIC WIBO

In the expansive realm of network protocols, one stands out as a versatile and resilient contender: BATMAN, the Better Approach to Mobile Ad-hoc Networks. Through the airwaves of large cities, BATMAN allows devices to seamlessly communicate over a mesh network, without any one device requiring knowledge of the wider network topology.

Over the summer, I participated in Google Summer of Code (GSoC) where I ported the `batman-adv` kernel module—which provides support for the BATMAN protocol on Linux—to FreeBSD. This GSoC thing is a program where students are awarded a stipend by Google for working on an open source project over the summer, overseen by a mentor, who, in my case, was the one and only Mahdi (or Mehdi, depending on which day of the week you ask him) Mokhtari, aka mmokhi@. He's a classy guy, and I'm very grateful to have had him as a guiding hand throughout my GSoC journey!

Currently with `batman_adv` (FreeBSD's equivalent to `batman-adv`), you can create and participate in meshes and send/receive packets over Ethernet. This all also works from within the Linuxulator. The porting effort mostly consisted of work on the LinuxKPI (notably backing `struct net_device` by `struct ifnet` and `mbuf` by `sk_buff`), which should hopefully ease porting other networking-related drivers from Linux in the future.

> Through the airwaves of large cities, BATMAN allows devices to seamlessly communicate over a mesh network.

What Did the Porting Process Look Like?

Though I'm far from an expert in the ways of porting kernel components (this is the first time I've ported anything remotely as large as BATMAN), here's a high-level overview of what the process looked like for me—on the off-chance some insight may be gleaned from my little adventure.

The first step was—naturally—to pull in the code for `batman-adv` from Linux into `sys/contrib/dev/batman_adv` and to create the Makefile to build it, which contains a list of all the source files to use and which compile-time options to set, such as telling it to include the LinuxKPI stuff. As expected, things didn't compile on the first try; `batman-adv` calls a lot of functions and uses a lot of structures which only exist or make sense in the context of the Linux kernel. This is the raison d'être of the LinuxKPI; it provides a compatibility layer that implements a subset of the Linux kernel by calling to equivalent (or sometimes not-so-equivalent) functions in the FreeBSD kernel.

So, the next natural thing was to get things to compile by writing stubs for all the missing functions and structures in the LinuxKPI, filling in the missing fields as they're used (we can't copy the structures from Linux as-is, because Linux is GPL-licensed). These stubs just contain debug print statements which let us know when they're being used.

With everything compiled, I could load the kernel module, which immediately panicked the kernel. Then it was just the process of going through all the stubs which are being called, looking up and understanding their implementation in Linux, and implementing an equivalent for FreeBSD in the LinuxKPI, until the kernel not-panicked. This was maybe 70% of the work.

Once every operation (loading the module, creating interfaces, sending stuff over it, and so forth) works and doesn't blow up anything, it is time to close the curtains, open up the ol' Wireshark on a second monitor, and lock myself in my kot (== Belgian dorm room) for a week straight getting a) all the devices on the mesh to recognize each other, and b) data to actually go from device A to device C passing through device B without getting mangled or lost in the process. A lot of this time was spent revising the (sometimes insufficient) implementations made during the previous step. This was maybe 30% of the work, but it felt like 90%, with most of my time spent staring directly at Wireshark and it staring back at me.

With a stable ABI, a single device driver can be used with multiple different network stacks.

Finally, BATMAN worked on FreeBSD, and all that was left to do was make userland tools support manipulating BATMAN interfaces, write a few lines of documentation, and open the curtains.

Upstreaming batman_adv

At EuroBSDCon last year, I spoke to some members of core@ about the possibility of upstreaming `batman_adv`, which they'd like to avoid, as BATMAN is GPL-licensed. So it'll probably remain as a port forever because there isn't really a use case where BATMAN is necessary to get a working FreeBSD system; if you need to use BATMAN for something, you'll likely be in a position where it's easy to fetch and build the port yourself anyway (as opposed to NIC drivers e.g.).

What's Left to Do?

The big limitation of BATMAN on FreeBSD right now is that it can't participate in wireless networks. I fully intend on getting at least wireless working in the following year, as that's where BATMAN's major use case lies. Hopefully I'll get that done in time for BSDCan 2024 :)

I wholeheartedly recommend participation in GSoC to anyone who meets the requirements to apply. I went from not knowing much about FreeBSD's network stack and kernel to, well, still not knowing all that much about it in the grand scheme of things, but certainly knowing a lot more than when I started. It especially helped me become a lot more comfortable navigating through the source tree and debugging kernel panics, even not-net-work-code-related.

Further Reading

Here's a link to my GSoC project wiki page where all the specifics, code, and a small demo video are:
https://wiki.freebsd.org/SummerOfCode2023Projects/CallingTheBatmanFreeNetworksOnFreeBSD

And, you might find this link to the batman-adv overview interesting, as it goes more in detail on the BATMAN implementation on Linux (and thus also on FreeBSD):
https://www.open-mesh.org/projects/batman-adv/wiki/Wiki

AYMERIC WIBO is a CS student at UCLouvain in Belgium and has been using and developing projects based on FreeBSD since high school. His primary interests lie in graphics and networking.

Make Your Own VPN—FreeBSD, Wireguard, IPv6 and Ad-blocking Included

BY STEFANO MARINELLI

Note: This article assumes a setup based on FreeBSD. If you prefer a version based on OpenBSD, it is available here.

VPNs are a fundamental tool for securely connecting to your own servers and devices. Many people use commercial VPNs for various reasons, ranging from not trusting their provider (especially when connecting from a public hotspot) to wanting to "go out" on the Internet with a different IP address, perhaps from another country. Here, I want to highlight some of the new features that have been brought into the base stack—many of which are enabled by default, and some of which may need to be specifically activated. Each feature will be described with details that may help improve the networking experience.

Whatever the reason, solutions are not lacking. I have always set up management VPNs to allow servers and/or clients to communicate with each other using secure channels. Lately, I have been activating IPv6 connectivity on all my devices (both desktop/servers and mobile devices) and I needed to quickly create a node that concentrated some networks and allowed them to go out on the network in IPv6. The tools I used and will describe are:

I want to highlight some of the new features that have been brought into the base stack.

- VPS – in this case, I used a basic Hetzner Cloud VPS, but any provider that provides IPv6 connectivity will do – if you want IPv6, of course.
- FreeBSD – a versatile, stable, and secure operating system.
- Wireguard – lightweight, secure, and at the same time, not very "chatty," so it is also gentle on mobile device batteries. When there is no traffic, it simply does not transmit/receive anything. Well supported by all major desktop and server operating systems as well as Android and iOS devices.
- Unbound – can make DNS queries directly to root servers, not through forwarders. It also allows you to insert block-lists and have a result similar to that of Pi-Hole (i.e., ad-blocking).
- SpamHaus lists – to immediately stop connections to and from users on blacklists.

The first step is to activate a VPS and install FreeBSD. On the Hetzner Cloud console, there might not be a pre-built FreeBSD image, but only a selection of Linux distributions. Don't worry, just choose any of them and create the VPS. Once done, the FreeBSD ISO image will be available among the "ISO Images." Just insert the virtual CD, restart the VPS, and the FreeBSD installation will appear in the console.

I won't go into detail, the operation is simple and straightforward. The only precaution (in the case of a Hetzner Cloud VPS) is to use "DHCP" for IPv4 but, for now, do not configure IPv6. It will be configured later.

Install all FreeBSD updates (using the **`freebsd-update fetch install`** command) and reboot.

Wireguard, on FreeBSD, is now available as a kernel module and the userland can be installed using the **`pkg install wireguard-tools`** package manager. This means you can easily keep it updated alongside other software on the system.

The first step is to configure IPv6 on the VPS. In the case of Hetzner, unfortunately, they only provide a /64, so it will be necessary to segment the assigned network. In this example, it will be divided into /72 subnetworks - to find valid subclasses, it will be possible to use a calculator.

The **`/etc/rc.conf`** file should have entries similar to:

```
ifconfig_vtnet0="DHCP"
ifconfig_vtnet0_ipv6="inet6 2a01:4f8:cafe:cafe::1 prefixlen 72"
ipv6_defaultrouter="fe80::1%vtnet0"
```

In short, keep the base address assigned by Hetzner, but change the prefix length to 72 - thus giving the possibility of having other networks available.

It is now necessary to enable forwarding for IPv4 and IPv6. Add these lines to the **`/etc/sysctl.conf`** file:

```
net.inet.ip.forwarding=1
net.inet6.ip6.forwarding=1
```

After reboot, test it:

```
ping6 google.com
```

If everything has been configured correctly, the ping will be executed and google.com will reply.

To configure Wireguard, a few steps will be necessary. First of all, the private key will need to be created:

```
wg genkey | tee /dev/stderr | wg pubkey  | grep --label PUBLIC -H .
```

You will get a private key and a public key. Take note of the public key — it will be needed to configure the clients.

Now create a new file called **`/usr/local/etc/wireguard/wg0.conf`**:

```
[Interface]
Address = 172.14.0.1/24,2a01:4f8:cafe:cafe:100::1/72
ListenPort = 51820
PrivateKey = YUkS6cNTyPbXmtVf/23ppVW3gX2hZIBzlHtXNFRp80w=
```

A new Wireguard interface called **`wg0`** is being created. Start the Wireguard interface:

```
service wireguard enable
sysrc wireguard_interfaces="wg0"
service wireguard start
```

If everything has been entered correctly, the interface should come up. Check its status:

```
wg
```

As for the firewall, FreeBSD comes with no `pf` configuration. In my setups, I tend to block what is not needed and be permissive with what may be useful. However, I like to keep out the "bad guys," so I use blacklists. **pf** allows elements to be inserted and removed from tables at runtime, so the firewall can be configured accordingly.

To download and apply the Spamhaus lists, I use a simple but effective script found on the Internet, but for OpenBSD.

For the Spamhaus lists, continue with the FreeBSD script creation.
Create the script in **/usr/local/sbin/spamhaus.sh**:

```
#!/bin/sh
#
#this is normally run once per day via cron.
#
echo updating Spamhaus DROP lists:
(
  { fetch -o - https://www.spamhaus.org/drop/drop.txt && \
    fetch -o - https://www.spamhaus.org/drop/edrop.txt && \
    fetch -o - https://www.spamhaus.org/drop/dropv6.txt ; \
  } 2>/dev/null | sed "s/;/#/" > /var/db/drop.txt
)
pfctl -t spamhaus -T replace -f /var/db/drop.txt
```

Make it executable and run it. Pf isn't enabled, so you'll get an error — but this will create the **/var/db/drop.txt** file:

```
chmod a+rx /usr/local/sbin/spamhaus.sh
/usr/local/sbin/spamhaus.sh
```

There are many possibilities to configure **pf** on FreeBSD. A fairly simple example could be this:

```
ext_if="vtnet0"
wg0_if="wg0"
wg0_networks="172.14.0.0/24"

set skip on lo

nat on $ext_if from { $wg0_networks } to any -> ($ext_if)

# Spamhaus DROP list:
table <spamhaus> persist file "/var/db/drop.txt"

block drop log quick from <spamhaus>

# Pass ICMP on ipv6
```

```
pass quick proto ipv6-icmp
# Block from ipv6 to wg0 network
block in quick on $ext_if inet6 to { 2a01:4f8:cafe:cafe:100::/72 }
# Pass Wireguard traffic - in and out
pass quick on $wg0_if

# default deny
block in
block out


pass in on $ext_if proto tcp to port ssh
pass in on $ext_if proto udp to port 51820

pass out on $ext_if
```

This is a very simple configuration: it blocks everything that is present in the list downloaded from Spamhaus, allows NAT from the Wireguard network to the public interface, allows ICMP traffic in IPv6 (necessary for the network to function properly) while blocking incoming traffic to the Wireguard IPv6 LAN (remember that the IPs will be public and directly reachable, so we don't want to expose our devices by default). All traffic on the Wireguard interface will be allowed to pass. Then everything will be blocked and exceptions will be specified, i.e., allowing SSH and Wireguard connections (of course). Authorization will also be granted to allow traffic to exit from the public network interface.

Save this configuration to **`/etc/pf.conf`**.

Enable and start **`pf`**:

```
service pf enable
service pf start
```

You will probably be kicked out of the system. Don't worry, just reconnect. pf is doing its job.

If everything went correctly, the firewall should have loaded the new rules.

To obtain caching of DNS queries and the related ad-block, it is now time to configure Unbound. Let's install it with:

```
pkg install unbound
```

A while ago, I found a script which I slightly adapted. I don't remember where I got it, so I'll paste it here without citing the original creator.

Create a script to update the unbound ad-block, in **`/usr/local/sbin/unbound-ad-hosts.sh`**:

```
#!/bin/sh
#
# Using blacklist from pi-hole project https://github.com/pi-hole/
# to enable AD blocking in unbound(8)
#
PATH="/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/local/sbin"
```

```
# Available blocklists - comment line to disable blocklist
_disconad="https://s3.amazonaws.com/lists.disconnect.me/simple_ad.txt"
_discontrack="https://s3.amazonaws.com/lists.disconnect.me/simple_tracking.
txt"
_stevenblack="https://raw.githubusercontent.com/StevenBlack/hosts/master/
hosts"

# Global variables
_tmpfile="$(mktemp)" && echo '' > $_tmpfile
_unboundconf="/usr/local/etc/unbound/unbound-adhosts.conf"

# Remove comments from blocklist
simpleParse() {
  fetch -o - $1 | \
  sed -e ‘s/#.*$//’ -e ‘/^[[:space:]]*$/d’ >> $2
}

# Parse DisconTrack
[ -n "${_discontrack}" ] && simpleParse $_discontrack $_tmpfile

# Parse DisconAD
[ -n "${_disconad}" ] && simpleParse $_disconad $_tmpfile

# Parse StevenBlack
[ -n "${_stevenblack}" ] && \
  fetch -o - $_stevenblack | \
  sed -n '/Start/,$p' | \
  sed -e 's/#.*$//' -e '/^[[:space:]]*$/d' | \
  awk '/^0.0.0.0/ { print $2 }' >> $_tmpfile

# Create unbound(8) local zone file
sort -fu $_tmpfile | grep -v "^[[:space:]]*$" | \
awk '{
  print "local-zone: \"" $1 "\" redirect"
  print "local-data: \"" $1 " A 0.0.0.0\""
}' > $_unboundconf && rm -f $_tmpfile

service unbound reload 1>/dev/null

exit 0
```

After saving the script, make it executable and run it:

```
chmod a+rx /usr/local/sbin/unbound-adhosts.sh
/usr/local/sbin/unbound-adhosts.sh
```

Now, the Unbound configuration file in **/usr/local/etc/unbound/unbound.conf** can be modified as follows:

```
server:
        verbosity: 1
        log-queries: no
        num-threads: 4
        num-queries-per-thread: 1024
        interface: 127.0.0.1
        interface: 172.14.0.1
        interface: 2a01:4f8:cafe:cafe:100::1
        interface: ::1
        outgoing-range: 64
        chroot: ""

        access-control: 0.0.0.0/0 refuse
        access-control: 127.0.0.0/8 allow
        access-control: ::0/0 refuse
        access-control: ::1 allow
        access-control: 172.14.0.0/24 allow
        access-control: 2a01:4f8:cafe:cafe:100::/72 allow

        hide-identity: yes
        hide-version: yes
        auto-trust-anchor-file: "/usr/local/etc/unbound/root.key"
        val-log-level: 2
        aggressive-nsec: yes
        prefetch: yes
        username: "unbound"
        directory: "/usr/local/etc/unbound"
        logfile: "/var/log/unbound.log"
        use-syslog: no
        pidfile: "/var/run/unbound.pid"
        include: /usr/local/etc/unbound/unbound-adhosts.conf

remote-control:
        control-enable: yes
        control-interface: /var/run/unbound.sock
```

Now, enable and start unbound:

```
service unbound enable
service unbound start
```

If everything has been set up correctly, unbound will be able to respond to DNS requests made on **172.14.0.1** and **2a01:4f8:cafe:cafe:100::1**.

Now it is possible to configure the Wireguard client. Create a new configuration by inserting "172.14.0.2/32, 2a01:4f8:cafe:cafe:100::2/128" (the ones that will later be entered in the peer configuration of the server) in the local IP addresses. Set the DNS server address to

"172.14.0.1" and/or its corresponding IPv6 address (in the example, 2a01:4f8:cafe:cafe:100::1 - yours will be different). In the peer section, insert the server's data, including its public key, IP address:port (in the example, the port is 51820), and allowed addresses (setting "0.0.0.0/0, ::0/0" means "all connections will be sent via Wireguard" — all the traffic will pass through the VPN for both IPv4 and IPv6).Each implementation has its own procedure (Android, iOS, MikroTik, Linux, etc.) but essentially it is sufficient to create the right configuration both on the server and on the client.

Reopen the Wireguard configuration file **/usr/local/etc/wireguard/wg0.conf** and add:

```
[Interface]
Address = 172.14.0.1/24,2a01:4f8:cafe:cafe:100::1/72
ListenPort = 51820
PrivateKey = YUkS6cNTyPbXmtVf/23ppVW3gX2hZIBzlHtXNFRp80w=

[Peer]
PublicKey = *client's public key*
AllowedIPs = 172.14.0.2/32, 2a01:4f8:cafe:cafe:100::2/128
```

The client's public key will be shown by the client itself.
Reload the Wireguard configuration:

```
service wireguard restart
```

It is also possible to use the VPN only as an ad-blocker, by only routing DNS traffic through it. To achieve this result, configure the client so that the only allowed address is the one of the just-configured unbound (in this example, 172.14.0.1 and/or 2a01:4f8:cafe:cafe:100::1) — DNS resolution will occur via VPN, but browsing will continue to work through the main provider.

To automatically update the spamhaus and ad-block lists, we will use cron.First, create a script, for example, **/usr/local/sbin/update-blocklists.sh**:

```
#!/bin/sh

/usr/local/sbin/unbound-adhosts.sh
/usr/local/sbin/spamhaus.sh
```

Make it executable:

```
chmod +x /usr/local/sbin/update-blocklists.sh
```

Then, add it to the crontab to run daily:

```
echo "@daily /usr/local/sbin/update-blocklists.sh" >> /etc/crontab
```

This approach benefits from both update management and security perspectives.

STEFANO MARINELLI is an IT Consultant with over two decades of experience in the realms of IT consulting, training, research, and publishing. His expertise spans across operating systems, with a special emphasis on *BSD systems — FreeBSD, NetBSD, OpenBSD, DragonFlyBSD - and Linux. Stefano is also the barista at BSD Cafe, a vibrant community hub for *BSD enthusiasts, and has led the FreeOsZoo project at the University of Bologna, making open-source operating system images accessible for virtual machines.

PRACTICAL PORTS

Monitor Your Hosts with Zabbix

BY BENEDICT REUSCHLING

I'd like to know what's going on, especially on the servers and machines that I am responsible for. Monitoring those systems has become a good practice for me. There are simply too many to check on a daily basis, and most of the time, there's nothing crazy going on. But when there is something going on, I want to know about it, even when it happened while I was asleep or otherwise away from a terminal. Monitoring also gives me an overview of my fleet of machines. I sometimes discover a host that has served its purpose but has not been recycled yet. I've even had cases where I could move new services to an underutilized machine, rather than spin up another host or jail.

Graphs serve me well in displaying what the machine has been doing over a period of time. Was last week's system load unusual or did lab groups in the university start? That disk is slowly filling up, I better do something about it. Questions like these come up and graphs over the collected machine metrics help answer them.

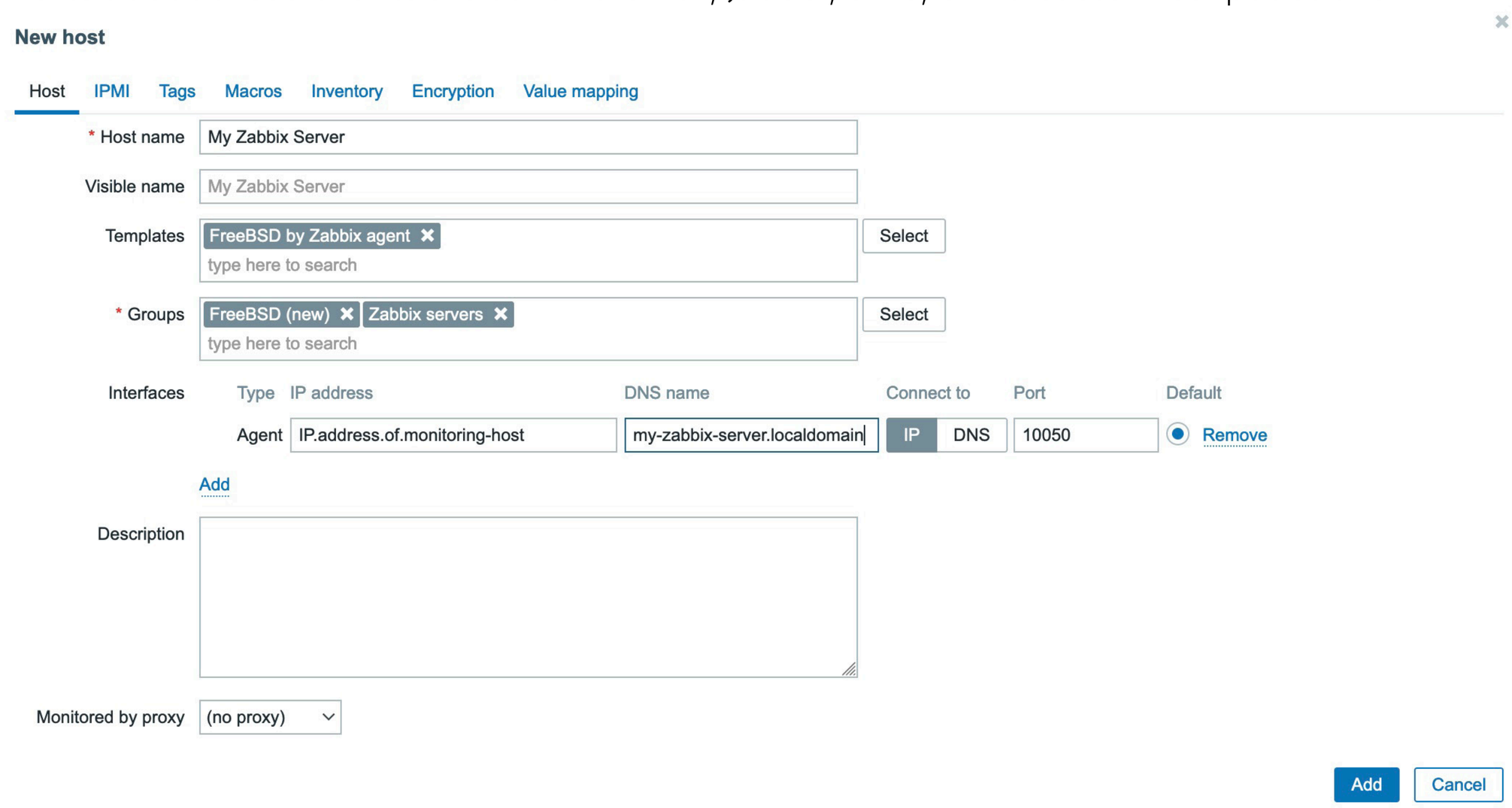
Any good sysadmin will sleep better knowing that their systems are up and running. Monitoring software can tell you they are, and you can even give management an uptime report for each individual machine to fill PowerPoint slides.

Any good sysadmin will sleep better knowing that their systems are up and running.

A monitoring system that can do all these things is called Zabbix. It is an open source solution to monitor your IT infrastructure, developed by a company that can even provide professional support if needed. The fully featured solution I am using is running on a long-term support cycle and I found the installation to be well documented. A central server collects the data from the systems running an agent. The server provides a web UI with dashboards, graphs, alerts about certain events, and much more. Another bonus for me was that FreeBSD is not an unknown operating system, and Zabbix has machine templates available to monitor important metrics like CPU, RAM, and even ZFS.

I run Zabbix from a jail and do not have any major problems with it. The PHP-based solution requires a database for storing the metrics. Postgres, MySQL/MariaDB, SQLite and even commercial DB2 from IBM and Oracle database servers can be used. The monitoring itself happens via SNMP/IPMI, ssh or a simple ping to check availability. For active monitoring (col-

lecting live machine metrics), an agent needs to be installed on those hosts. There are even functionalities in place to monitor whole subnets for new hosts and add them to monitoring as they appear There are even functionalities in place to monitor whole subnets for new hosts and automatically add them to monitoring when they first appear. Triggers set on certain hosts for specific situations (for example, disk full) are configurable via the web interface. Alerts about these events are sent via email, Jabber, SMS, or via custom script actions.



Zabbix Setup

Let's see how we can install this monitoring solution. I'm starting out with a fresh FreeBSD 14.0 jail that is connected to the network that I'm monitoring and that downloads required packages from the web or a separate Poudriere server. Note that I configured the port from using the default MySQL to use PostgreSQL by running "make config" in the net/mgmt/zabbix64-server ports directory.

These are the packages that we need:

```
# pkg install zabbix64-server zabbix64-frontend-php82 postgresql15-server nginx
```

In an earlier attempt, I was using zabbix6-server and frontend, because I thought version 6.4 would eventually become version 6.5. But the release schedule for Zabbix is a bit different. The long-term support versions are the major versions (6 in this case), whereas the minor release versions (6.4) have a shorter support cycle. I switched to the long-term support version, as I didn't want to be on the bleeding edge of monitoring and can run with the current feature set for a while. Stability is what I want as a primary goal. You don't want to fix your monitoring every time a new version comes out, when you rely on monitoring your critical systems with it.

Activate services to run when the monitoring host (or jail) starts using sysrc:

```
# sysrc zabbix_server_enable=yes
# sysrc zabbix_agentd_enable=yes
# sysrc postgresql_enable=yes
# sysrc nginx_enable=yes
# sysrc php_fpm_enable=yes
```

I don't use SNMP in my setup (that may change in the future), but the pkg-message has details about how to enable the SNMP daemon if you do. Our first task after the package installation is to set up the zabbix database. I love Postgres, so I use that in my setup here. As mentioned, other databases are supported, so pick your favorite data storage solution here.

Database Setup

First, I change to the postgres user, which was installed as part of the package.

```
# su postgres
```

After switching to **/var/db/postgres** (which is the postgres home directory), I run the initdb command to initialize the database cluster. Then I start the database using the pg_ctl command since we need a running database to import the Zabbix base tables.

```
$ cd /var/db/postgres
$ initdb data
$ pg_ctl -D ./data start
```

Once the database is running, I switch to

```
/usr/local/share/zabbix64/server/database/postgresql/
```

where the database templates for the tables, triggers, and everything else resides.

```
$ cd /usr/local/share/zabbix64/server/database/postgresql/
```

Then I run PostgreSQL's interactive shell called psql. In there, I create a new database for zabbix, a user with the same name and give it a password. After granting that user permissions on the zabbix database, I need to log out from psql to switch to the zabbix user we just created.

```
psql -d template1
psql> create database zabbix;
psql> CREATE USER zabbix WITH password 'yourZabbixPassword';
psql> GRANT ALL PRIVILEGES ON DATABASE zabbix to zabbix;
psql> exit
```

Use psql again to log in as the new zabbix user into the empty zabbix database. We load three files that contain table definitions for zabbix (and some other database objects) in that order: **schema.sql**, **images.sql**, and finally **data.sql** from the local directory we changed into earlier. Once we're done, we can log out of the database again.

```
$ psql -U zabbix zabbix
psql> \i schema.sql
psql> \i images.sql
psql> \i data.sql
psql> exit
```

That is all that is needed for the database. Let's move on to configuring Zabbix itself.

Zabbix Configuration

Zabbix needs to know which database to use to store metrics, machines to monitor, and a whole lot of other information. The main Zabbix configuration is located in **/usr/local/etc/zabbix64/zabbix_server.conf** as a straightforward key = value file. Comments de-

scribe what the values do and most of them are commented out. After I made the necessary changes, my file looks like this:

```
SourceIP=IP.address.of.monitoring-host
LogFile=/var/log/zabbix/zabbix_server.log
DBHost=
DBName=zabbix
DBUser=zabbix
DBPassword=yourZabbixPassword
Timeout=4
LogSlowQueries=3000
StatsAllowedIP=127.0.0.1
```

The first thing that I defined is the **SourceIP** address of the central Zabbix server that collects the metrics. Note that the frontend (the web UI) does not have to reside on the same host, but I keep things central here. I define where Zabbix should log any events that happen during its runtime (**LogFile**). Create that path and file if it does not exist as yet and set the owner to the Zabbix user (this is the one that came with the package installation).

No, I did not forget to set a value for the **DBHost** parameter. When running on Postgres, this needs to be empty. One day, when you have to define your own config file, you may think back about this and may do things differently. Less typing for me, and remember, other databases may require a value here, so adapt your setup if you don't use Postgres. **DBName**, **DBUser**, and **DBPassword** (in clear text) are the ones that we created earlier in the database setup. This connects Zabbix to the database instance to store and retrieve values during monitoring. Both **Timeout** and **LogSlowQueries** are default values I kept. I had no need to adjust them yet, but I can imagine resource constraints would be a good reason to do just that. Internal statistics for Zabbix itself are collected as well and the **StatsAllowedIP** parameter defines from where to accept those. Localhost is totally fine for my use case. Another Zabbix instance may want to access those as well, which is why you can define multiple addresses here if you have several Zabbix servers monitoring each other.

That's it for the server configuration file, at least for the basics. I'm using Agent monitoring for both the clients and the server itself. When using SNMP, there may be additional values required. Check the documentation for Zabbix for details.

Zabbix Agent Configuration

The gent is aptly named Zabbix Agentd (good old Unix daemons) and it should run when the server jail starts. The entry for **/etc/rc.conf** is done using sysrc and looks like this:

```
# sysrc zabbix_agentd_enable=yes
```

Located next to the server configuration file is the one for the agent, too. The filename is **zabbix_agentd.conf** in **/usr/local/etc/zabbix64/zabbix_agentd.conf**. My own changes make this file look like the following:

```
LogFile=/var/log/zabbix/zabbix_agentd.log
SourceIP=IP.address.of.host-to-monitor
Server=IP.address.of.monitoring-host
ServerActive=127.0.0.1
Hostname=Zabbix server
```

We find familiar lines in this file. The **LogFile** should be a different one to distinguish messages from the server and the client, especially for the server. The machines that are monitored typically don't run the server parts, so there is no mixing them up on those systems. The directory should exist by now, but the file needs a manual nudge via touch(1) to come into existence. **SourceIP** is exactly the same as the monitoring host sends its own metrics to itself. For other clients, set this to the IP or DNS address of that host in question. **ServerIP** stays the same on all systems, as we are using a central system to collect the metrics. **ServerActive** has been discussed above and **Hostname** is an identifier that Zabbix uses internally to keep systems distinct.

Zabbix Frontend

The Zabbix frontend requires a running webserver with PHP enabled and its own configuration file. The front-end needs to know how to get metrics and other data from the back-end, which is why another configuration file requires the database credentials. The file in question is located in **/usr/local/www/zabbix64/conf/zabbix.conf.php**, right where the rest of the website files reside that make up Zabbix. A few comments will guide you to the lines that need to be filled with the proper values. Mine look like this:

```
<?php
  // Zabbix GUI configuration file.
  $DB[‘TYPE’]                             = 'POSTGRESQL’;
  $DB[‘SERVER’]                           = 'localhost';
  $DB[‘PORT’]                             = '0';
  $DB[‘DATABASE’]                         = 'zabbix';
  $DB[‘USER’]                             = 'zabbix';
  $DB[‘PASSWORD’]                         = 'yourZabbixPassword';
  $DB[‘SCHEMA’]                           = '';
```

Additional settings about TLS encryption are left as an exercise to the reader. Definitely consider implementing that or someone listening on the wire may be able to determine a lot of information about the hosts sending their metrics to the central server on a regular basis. I omitted it here to keep the tutorial focused on the most important parts to make Zabbix run in the first place.

My webserver of choice is nginx, but any other will do just fine. The main **nginx.conf** after the changes had these lines in it:

```
worker_processes  1;
events {
  worker_connections  1024;
}
http {
  include             mime.types;
  default_type        application/octet-stream;
  sendfile            on;
  keepalive_timeout   65;
  server {
    listen            80;
    server_name       localhost;
    root /usr/local/www/zabbix64;
    index index.php index.html index.htm;
```

```
location / {
  try_files $uri $uri/ =404;
}
location ~ .php$ {
  fastcgi_split_path_info ^(.+\.php)(/.+)$;
  fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
  fastcgi_param PATH_INFO $fastcgi_path_info;
  fastcgi_param REMOTE_USER $remote_user;
  fastcgi_pass unix:/var/run/php-fpm.sock;
  fastcgi_index index.php;
    include fastcgi_params;
  }
  error_page   500 502 503 504  /50x.html;
  location = /50x.html {
    root   /usr/local/www/nginx-dist;
  }
}
```

Note that this also misses the SSL/TLS setup that is now ubiquitous on the web. I was using an SSL proxy for the requests, but a Let's encrypt solution is cheap and easy to implement. Again, I focus on Zabbix here.

For PHP itself, the configuration is a few lines away from production use. Edit **/usr/local/etc/php-fpm.d/www.conf** and change these settings to the values shown:

```
listen = "/var/run/php-fpm.sock"
```

Uncomment the following:

```
listen.owner
listen.group
listen.mode
```

The file **/usr/local/etc/php.ini** should use production values, so copy and rename **/usr/local/etc/php.ini-production** to that filename. We are not quite done yet, as the setup will check for certain PHP values later and complain about lower (or outright wrong) values. Change these to have a smooth experience (and change the time zone to yours):

```
date.timezone = Europe/Berlin
post_max_size = 16M
max_execution_time = 300
max_input_time = 300
```

I saved those files, ran **sysrc nginx_enable=yes** followed by starting all services that are part of the Zabbix stack:

```
service postgresql restart
service php-fpm start
service nginx start
zabbix_server start
zabbix_agentd start
```

Website Woes

That went fine and I saw a couple of lines appear in the log file for Zabbix (both the server and agent). Excited as I was, I went to my browser of choice and entered the URL for Zabbix. Nothing happened. A white screen with nothing on it greeted me. I restarted the services, checked the logs, and re-checked the configuration files. No effect, still the same white page. I tried a different browser, just to rule out any peculiarities. The white page was staring in my face, just the same.

Whenever you meet me at a BSD conference (which is a likely event), you'll notice how little hair is left on my scalp. I could blame bad genes or environmental factors, but working in IT is a big contributor to why I pull my hair out in situations like this. This is supposed to work according to the documentation, so why is there nothing to see on the website?

This article would reach an unsatisfying end here if luck had not pointed me in the right direction. I had opened the Zabbix FAQ to read about other things when my eyes caught this section:

```
If "opcache" is enabled in the PHP 7.3 configuration, Zabbix frontend may show a blank
screen when loaded for the first time. This is a registered PHP bug. To work around
this, please set the "opcache.optimization_level" parameter to 0x7FFFBFDF in the PHP
configuration (/usr/local/etc/php.ini file).
https://www.zabbix.com/documentation/current/en/manual/installation/known_issues
```

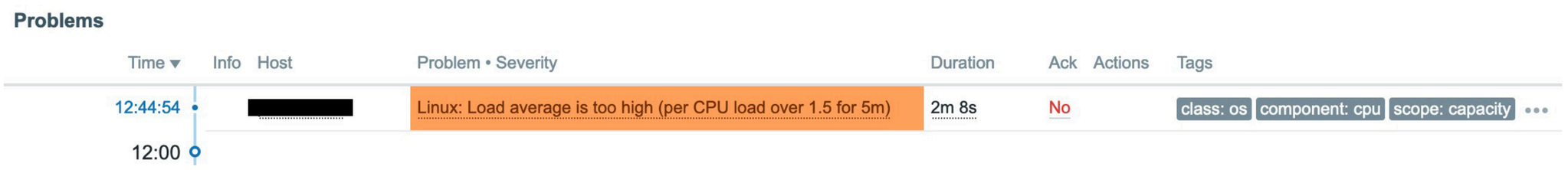
Solution

OK, so I went back to **`/usr/local/etc/php.ini`** and set the **`opcache.optimization_level`** like this:

```
opcache.optimization_level=0x7FFFBFDF
```

Obscure value as it may be, this setting may or may not be required when you, dear reader implements this. I kept it in there to avoid another round of hair pulling. Restarting the services one more time, I was greeted by the Zabbix web configuration. The setup itself is main confirming the PHP settings and values for the backend database. Set a password for the web frontend and then log in for the first time.

You probably do not see a host on a fresh installation. To see at least your own monitoring server, click on the left side (big eye icon) on Monitoring, and then Hosts. On the new page, click the "Create Host" button in the upper right corner. A form will appear where you can enter information about the host like name, IP address, and what kind of monitoring to use (zabbix agent in our case). The templates field holds presets for certain kinds of hosts, and we will find one for FreeBSD in there when we start typing that operating system name. Groups logically combine hosts that have similar characteristics (you can create as many groups as you need). This makes filtering easier, and if an outage occurs, may tell you what other systems need looking after. In the interfaces section, click "Add" and select Agent. New input fields will appear where you enter the IP address and/or DNS name. The port 10050 is the default for the agent, so check if any firewall rules prevent access. A description is optional, but with the growing number of servers, it's good to remind yourself about a systems purpose. Click the blue "Add" button to create the host. It will appear in your host list, and if all went well, should have data collected by the agent as well as some graphs show up soon after. The dashboard will display any problems detected over the monitoring period.



Check the Zabbix documentation for monitoring a whole subnet, how to use SNMP monitoring, and how to automatically add hosts without having to click through each one of them in the Zabbix documentation. Explore the rest of the Zabbix UI for cool features that you never knew you missed but definitely want to regularly revisit to see if something is out of the ordinary. Happy monitoring!

BENEDICT REUSCHLING is a documentation committer in the FreeBSD project and member of the documentation engineering team. In the past, he served on the FreeBSD core team for two terms. He administers a big data cluster at the University of Applied Sciences, Darmstadt, Germany. He's also teaching a course "Unix for Developers" for undergraduates. Benedict is one of the hosts of the weekly bsdnow.tv podcast.

10 Years of the *FreeBSD Journal*

A CROSSWORD BY TOM JONES

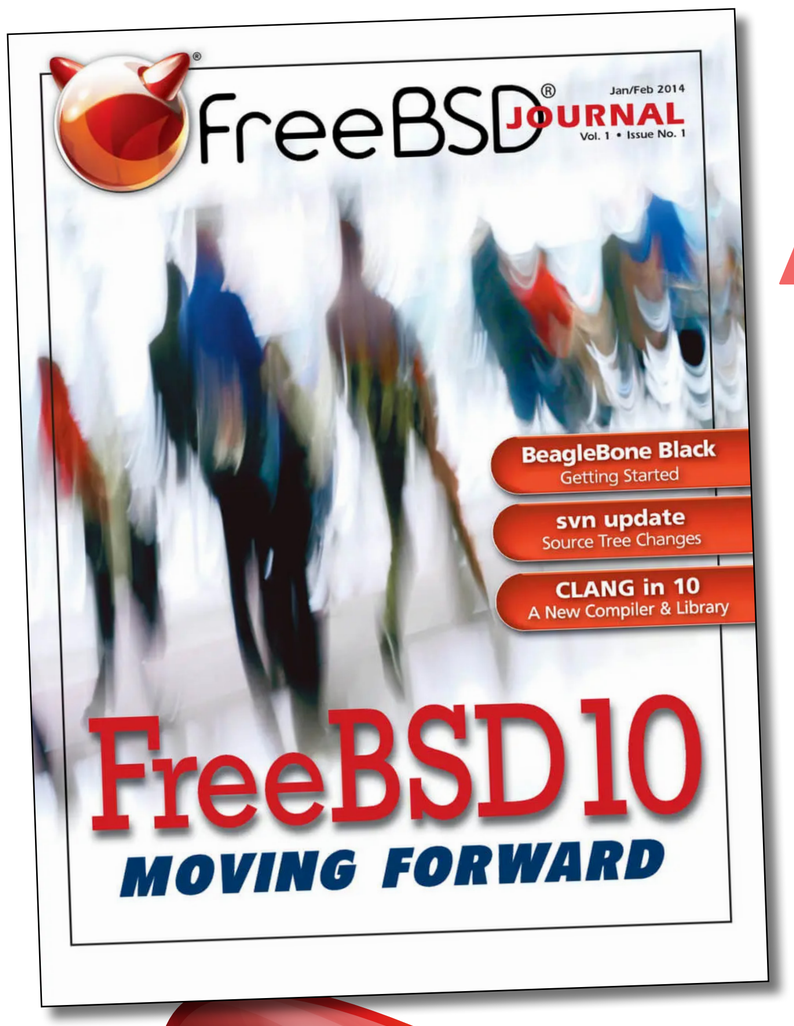
ACROSS

1. Makes your stackoverflow useless
4. It's Hammer time
5. Progentitor OS
7. Frequently phonetically confused conference
8. Without them there would be nothing about which to write
11. Knows what your functions are thinking
12. King of the photocopies
13. Waiting for Godot events

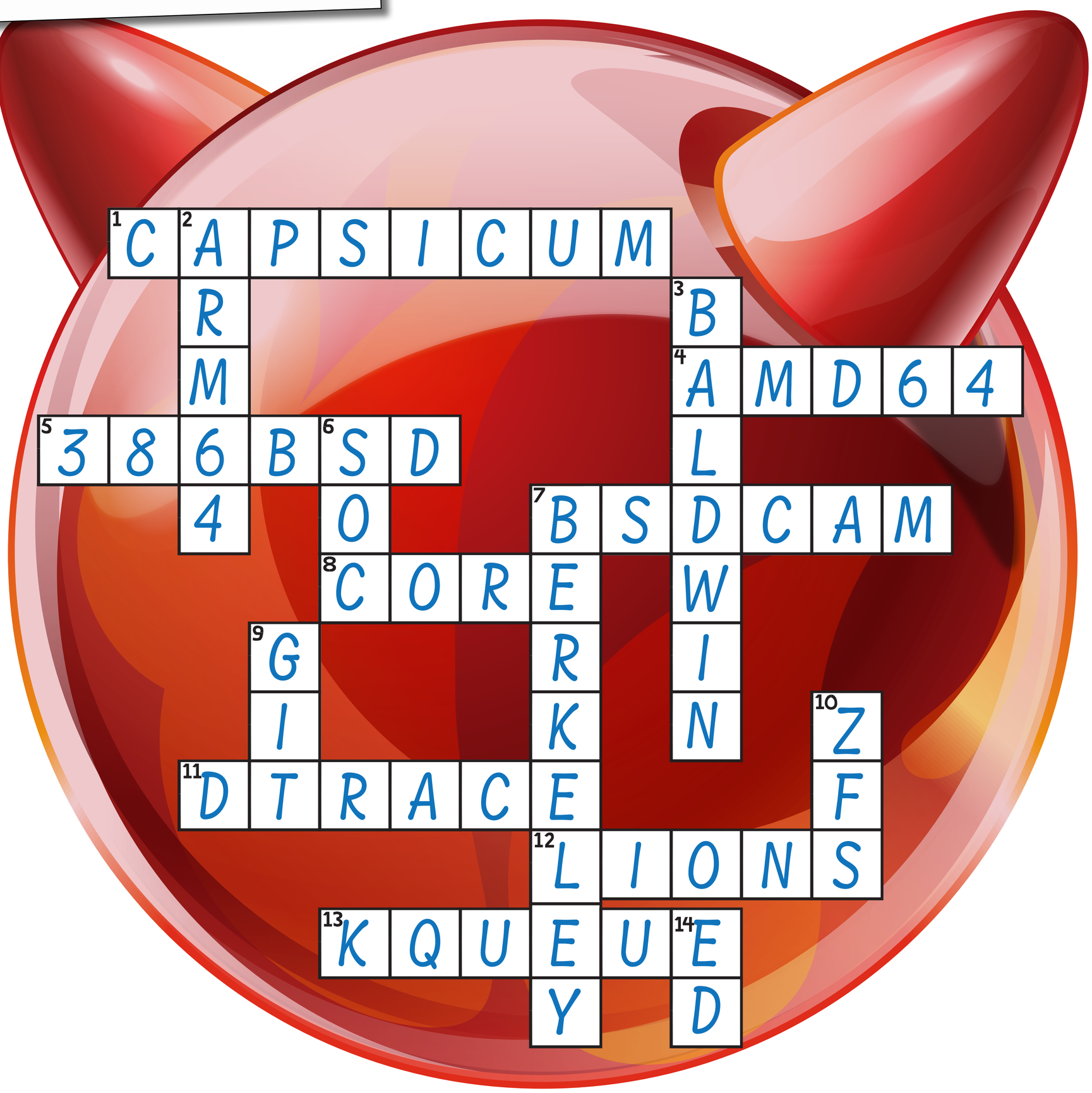
DOWN

2. Futuristic Tier 1
3. The *Journal's* thread safe Chair
6. Complete Packaging
7. Universal home of open source
9. Friendly Version Control
10. Zed's dead baby
14. The standard editor

Answers on next page



A decade ago, we unveiled the inaugural issue, spotlighting the release of FreeBSD 10.0 and featuring discussions on pivotal topics such as the transition to clang as the base system C and C++ compiler, and ZFS. Over the past ten years, the Journal has showcased a wealth of content, with more than 250 articles, accompanied by numerous book reviews and engaging trip reports. None of this would have been possible without the invaluable contributions from authors across the FreeBSD community, both past and present members of the Editorial Board, and, of course, our dedicated readers. A heartfelt thank you to all!



FreeBSD Foundation 2023 Recap

BY DEB GOODKIN

I am sitting here amazed at how quickly 2023 flew by, but I'm also looking forward to kicking off the new year with rejuvenated energy to take on an exciting array of opportunities for FreeBSD. I hope you had a chance to slow down a bit over the holiday season, spend time with friends and family, and pursue activities that bring you joy. If you're like me, you probably thought about what you would like to do differently in the new year: Be more active, eat better, get out more, spend more time using and contributing to your favorite operating system. I hope all of those, but especially that last one. Our sole purpose is to support the FreeBSD Project and community, and if you are reading this, you are part of our amazing community!

As I reflect on what we did to support FreeBSD in 2023, I would like to share some of the highlights while pointing out areas we will focus on in 2024. We've set our sights high with some lofty goals, including:

- Improving the desktop experience, to make it easier for those wanting to use FreeBSD as their daily driver or even those who want to try out FreeBSD.
- Supporting specific features and technologies to ensure FreeBSD is the operating system of choice when organizations are looking at options
- Increasing the visibility of FreeBSD to individuals and organizations, not only to inform them that FreeBSD is an option, but why they should choose FreeBSD.

As I reflect on what we did to support FreeBSD in 2023, I would like to share some of the highlights while pointing out areas we will focus on in 2024.

I'd like to start out by expressing my gratitude to everyone in this remarkable community for their support and contributions. Whether you write code, produce informative and engaging FreeBSD videos on YouTube, improve the documentation, support our infrastructure, submit bug reports or fix bugs, provide answers on Reddit, give FreeBSD presentations or represent FreeBSD at conferences, you are playing a crucial role in advancing FreeBSD. Thank you all.

At the end of last year, we published blog posts highlighting the main areas we support. In my recap, I'm summarizing those highlights for you while providing links so you can read in more depth the topics you are more interested in.

Grab some coffee, tea, Kombucha or a beer, sit back, and continue reading to see some of the highlights of how we supported FreeBSD this past year. I included links within each area to more detailed year-end reports.

Software Development Work to Improve FreeBSD

Did you know that over 60% of our budget is dedicated to funding software development work to improve the operating system? This includes keeping a small staff of software developers who can quickly step in to fix bugs, review changes, and implement features and

functionality in FreeBSD. We also have a handful of contractors working on specific funded projects that align with our long-term goals and longer-term projects like supporting wifi and specific architectures.

Key Highlights in this area include:

- Sponsored 1082 of the 7060 commits to the src repository.
- We heard you! We added the necessary resources to accelerate the wifi efforts by continuing to fund Bjoern and bringing on two more contractors to focus solely on this area.
- Imported OpenSSL v3 into our base system in preparation for FreeBSD 14.0 RELEASE
- Hired six students for summer internships! They had opportunities to make impactful contributions to FreeBSD, including areas like wifi, DTrace, Capsicum, and more. Our internship program highlights the incredible opportunity for students interested in systems programming to gain real-world skills while making significant contributions to FreeBSD.

Go here to learn more about our software development work and accomplishments: https://freebsdfoundation.org/blog/2023-in-review-software-development/

Advocating for FreeBSD and the Community

Part of our effort in supporting FreeBSD is ensuring the Project gets the visibility it needs and deserves. We are here to be a lighthouse for this 30+ year-old project that continues to grow and innovate while recruiting others to do the same. We do this by promoting FreeBSD at computing and open source conferences worldwide, writing blogs and articles highlighting the benefits and features of FreeBSD, and producing the professionally published FreeBSD Journal filled with informative and interesting technical articles.

Key highlights in this area include:

- Organizing, presenting, and sponsoring 16 events, including running Getting Started with FreeBSD workshops, resuming our Bay Area Vendor Summit at an amazing venue (provided by NetApp) with some incredible talks and organizing the BSDCan FreeBSD Developer Summit that included celebrating 30 years of FreeBSD (yes, there was cake!).
- Expanding the coverage of FreeBSD in the media by participating in interviews, writing articles, and working with a public relations firm.
- Getting your input on FreeBSD. You have ideas, and we're helping to get your voices heard. In partnership with the Core team, we produced an in-depth community survey to get your feedback and help inform how the Core Team and FreeBSD Foundation can better support you and FreeBSD going forward.
- Producing the FreeBSD Journal for 10 years! This is a major collaborative effort between the Foundation and FreeBSD community members who volunteer their time to help produce high-quality articles highlighting FreeBSD features and technologies.
- Creating How-to guides to help newer folks quickly get started with FreeBSD.
- Expanding our What is FreeBSD page so that it can better be used by individuals and

Part of our effort in supporting FreeBSD is ensuring the Project gets the visibility it needs and deserves.

companies as a resource to sell to their own leadership, vendors, and customers on the benefits of FreeBSD. We're still fine tuning the content, but you can check it out here: https://freebsdfoundation.org/freebsd-project/what-is-freebsd/

Go here to learn more about how we're advocating for FreeBSD and helping with community engagement: https://freebsdfoundation.org/blog/2023-in-review-advocacy/

Building and Strengthening Partnerships

This section could be called fundraising, but it's so much more than that! Yes, we rely 100% on donations, sponsorships, and grants to fund our work. However, we created a new role to work with our current and prospective partners. The role is not only about securing funding, but also educating them on the importance of contributing back and engaging with the community. It's about reminding them of why FreeBSD is or continues to be the right choice for their organization, and about the importance of shining a light (being a lighthouse) on their use of FreeBSD. We hired Greg Wallace in April to take on this role, bringing many years of open source, business, and marketing experience to concentrate on building and strengthening partnerships

Greg stepped into this role running, gently nudging us to accelerate our efforts in hiring more developers, producing more content, and connecting with more companies to understand their use cases and challenges.

Key highlights in this area include:

- Identifying many dark users – companies using FreeBSD that we weren't aware of
- Engagement by the numbers:
  - 50 - Number of companies reached out to
  - 33 - Number of companies talked to
  - 05 - Number of companies with outstanding partnership proposals
  - 11 - Total corporate partners
  - 03 - Total first-time corporate partners
- Identifying and applying for new grant opportunities to fund security efforts
- Participating and identifying government security guidelines and mandates to ensure our voice is included in these discussions, plus doing the due diligence that FreeBSD will incorporate these requirements to keep FreeBSD secure for the Project and partners.
- Developing consistent messaging and pitches that are helping to inform companies why they should invest in and use FreeBSD - Just look at the # of companies with outstanding partnership proposals above!
- Organizing and running the new Enterprise Working Group that has helped bring the community and enterprise customers together, identifying gaps and opportunities in FreeBSD, while providing opportunities for participants to get more involved in some of these areas.

And, Research too!

This new role also consists of a research component, which includes researching and identifying key markets for FreeBSD, emerging markets like AI and how FreeBSD fits in, and where we should invest to make the biggest impacts in FreeBSD.

Go here to learn more about what we did in partnerships and research and what some of our plans for 2024 are: https://freebsdfoundation.org/blog/2023-in-review-partnerships-and-research/

FreeBSD Infrastructure

We approved over $100,000 for a cluster refresh that began in 2023 and will carry over into this year by purchasing and shipping 15 new servers to a new NYI Chicago facility. We're excited to collaborate with the clusteradm team to provide the funding, resources, and coordination efforts to stand up the new hardware, taking advantage of NYI's generous donation of 4 racks in their newer colocation facility.

We have one staff member on the clusteradm team and another who helps facilitate the efforts between the team and the Foundation. Go here to learn more about how we've been supporting the Project's infrastructure: https://freebsdfoundation.org/blog/2023-in-review-infrastructure/

Continuous Integration

One full-time staff member is dedicated to improving the Project's continuous integration system and test infrastructure.

Some of the highlights include:

- Adding more testing jobs for ARM64 architectures like testing with Kernel Address Sanitizer and building tests for non-standard compilers like GCC 12 and 13.
- Making great progress running the workflow working group, designing and implementing systems to support the pull-request based workflow.
- Implementing Pre-commit CI changes that will be available soon!
- Publishing git hooks used for the Project's git repository and now producing semi-official release snapshots from the CI system.
- Updating the "Tinderbox View" (https://tinderbox.freebsd.org) of the CI result dashboard, which now provides more details of the test results and the possible breakage point.

We approved over $100,000 for a cluster refresh that began in 2023.

Cloud Support

The Foundation also supports our full-time staff member's efforts to work with the engineers from Microsoft to help them implement support for new features in Azure and provide more FreeBSD features in Azure. This includes ARM64 VM support, Gen2 VM support and ZFS images provided. All these changes are included in 14.0-RELEASE and published to the Azure Marketplace.

Go here to learn more about how we've been supporting the Project's testing and CI efforts: https://freebsdfoundation.org/blog/continuous-integration-and-workflow-improvement/

Legal

The Foundation owns the FreeBSD trademarks, and it is our responsibility to protect them. We review a handful of trademark permission requests per month and make sure our registrations haven't expired. Go here https://freebsdfoundation.org/legal/trademark-usage-terms-and-conditions/ to find out more information about the FreeBSD trademarks.

We also provide legal support for the core team to investigate questions that arise. We engaged in multiple NDAs this year, which allows us to understand the commercial user's FreeBSD needs.

Going Forward

We are excited about the future of FreeBSD and the increased interest in individuals and companies using our favorite open source operating system! We've identified areas within the Project, where we can make impactful investments to ensure FreeBSD is the operating system of choice going forward.

We are finalizing our 2024 plans, and will share them soon. But, here's a little preview of what we are planning:

- Improving the desktop and developer experience
- Strengthening and increasing our partnerships
- Increasing the visibility of FreeBSD by providing more content and articles on the benefits and features of FreeBSD, working with more companies to share their stories and use cases, and increasing media exposure. In a nutshell, getting the word out there about FreeBSD and why everyone should use it!
- Implementing key features to keep FreeBSD innovative and the platform of choice for commercial entities.

In a nutshell, we will be implementing soon-to-be-announced features and technologies in the operating system, and increasing the visibility of the Project, while continuing to investigate key markets and opportunities for FreeBSD. We're also going to increase our advocacy efforts, especially in the area of technical content, so watch for an announcement about that soon!

We're looking forward to the opportunities ahead for FreeBSD and will continue and increase our support. If you are a software developer looking for full-time or contract work, keep an eye on our jobs page https://freebsdfoundation.org/open-positions/.

With the number of start-ups using FreeBSD, and companies switching to FreeBSD, the future truly looks bright for us! We will be here supporting the Project, community, and users to ensure FreeBSD stands out as a compelling choice for individuals and businesses.

DEB GOODKIN is the Executive Director of the FreeBSD Foundation. She's thrilled to be in her 19th year at the Foundation and is proud of her hardworking and dedicated team. She spent over 20 years in the data storage industry in engineering development, applications engineering, and technical marketing. When not working, you'll find her road or trail running, playing with her dogs, cycling the backroads of Colorado, or reading a good book.

2024 Events Calendar

BSD Events taking place through May 2024

BY ANNE DICKISON

Please send details of any FreeBSD related events or events that are of interest for FreeBSD users which are not listed here to freebsd-doc@FreeBSD.org.

SCALE 21X

March 14-17, 2024
Pasadena, CA
https://www.socallinuxexpo.org/scale/21x

SCaLE is the largest community-run open-source and free software conference in North America. It is held annually in the greater Los Angeles area. Drew Gurkowski will also be hosting a FreeBSD workshop during the conference.

AsiaBSDCon 2024

March 21-24, 2024
Taipei, Taiwan
https://2024.asiabsdcon.org/

AsiaBSDCon is for anyone developing, deploying and using systems based on FreeBSD, NetBSD, OpenBSD, DragonFlyBSD, Darwin and MacOS X. It is a technical conference and aims to collect the best technical papers and presentations available to ensure that the latest developments in our open source community are shared with the widest possible audience.

Save the Date: May 2024 FreeBSD Developer Summit

May 29-30, 2024
Ottawa, Canada

Join us for the May 2024 FreeBSD Developer Summit, co-located with BSDCan 2024, which will take place in Ottawa, Canada. The two-day event takes place May 29-30, 2023, and will consist of developer discussion sessions, vendor talks, and working groups. More information will be available in March 2024.

BSDCan 2024

May 29 - June 1, 2024
Ottawa, Canada
https://www.bsdcan.org/2024/

BSDCan is a technical conference for people working on and with BSD operating systems and related projects. It is a developers conference focusing on emerging technologies, research projects, and works in progress. It also features Userland infrastructure projects and invites contributions from both free software developers and those from commercial vendors.