FreeBSD® JOURNAL

November/December 2024

Virtualization Issue

Character Device Driver Tutorial (Part 2)

BHYVE for the Windows and Linux Users

Xen and FreeBSD

Wifibox: An Embedded Virtualized Wireless Router

Embedded FreeBSD: Fabric—Baby Steps

Adventures in TCP/IP: Dynamic Goodput Pacing: A New Approach to Packet Pacing

Conference Report: My EuroBSDCon Experience in Dublin

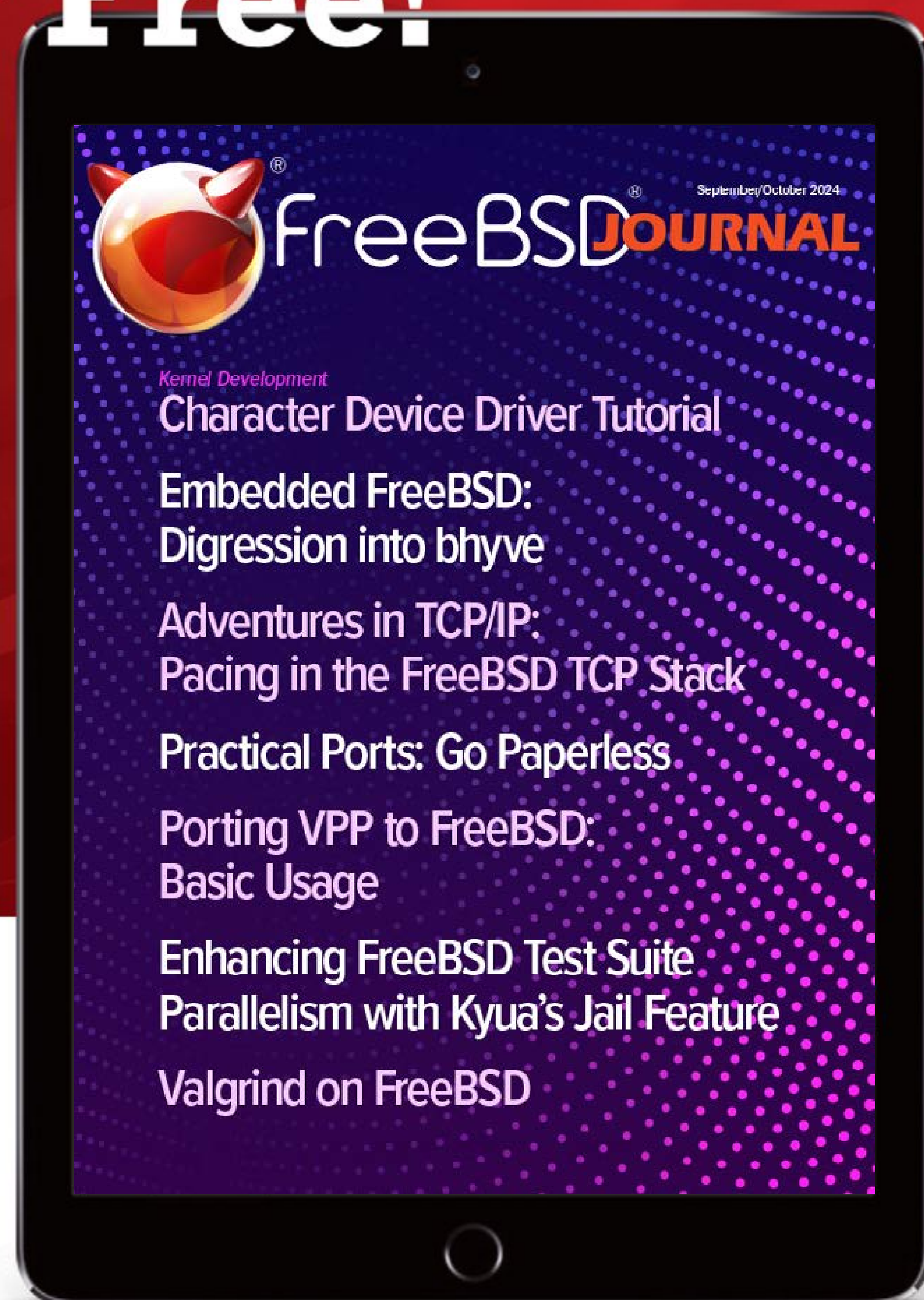
FreeBSD® JOURNAL

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!



2025 Editorial Calendar

- Jan/Feb/March Downstreams
- April/May/June Networking
- July/August/Sept Contributing/Workflow
- Oct/Nov/Dec Embedded

Find out more at: freebsdfoundation/journal

FreeBSD® JOURNAL

Editorial Board

John Baldwin • FreeBSD Developer and Chair of the *FreeBSD Journal* Editorial Board

Tom Jones • FreeBSD Developer, Software Engineer, FreeBSD Foundation

Ed Maste • Senior Director of Technology, FreeBSD Foundation and Member of the FreeBSD Sec Team

Benedict Reuschling • FreeBSD Documentation Committer

Jason Tubnor • BSD Advocate, Senior Security Lead at Latrobe Community Health Service (NFP/NGO), Victoria, Australia

Mariusz Zaborski • FreeBSD Developer

Advisory Board

Anne Dickison • Deputy Director FreeBSD Foundation

Justin Gibbs • Founder of the FreeBSD Foundation, President of the FreeBSD Foundation Board, and a Software Engineer at Facebook

Allan Jude • CTO at Klara Inc., the global FreeBSD Professional Services and Support company

Dru Lavigne • Author of *BSD Hacks* and *The Best of FreeBSD Basics*

Michael W Lucas • Author of more than 40 books including *Absolute FreeBSD*, the *FreeBSD Mastery* series, and *git commit murder*

Kirk McKusick • Lead author of *The Design and Implementation* book series

George Neville-Neil • Past President of the FreeBSD Foundation Board and co-author of the *Design and Implementation of the FreeBSD Operating System*

Hiroki Sato • Chair of AsianBSDCon, Member of the FreeBSD Core Team, and Assistant Professor at Tokyo Institute of Technology

Robert N. M. Watson • Director of the FreeBSD Foundation Board, Founder of the TrustedBSD Project, and University Senior Lecturer at the University of Cambridge

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

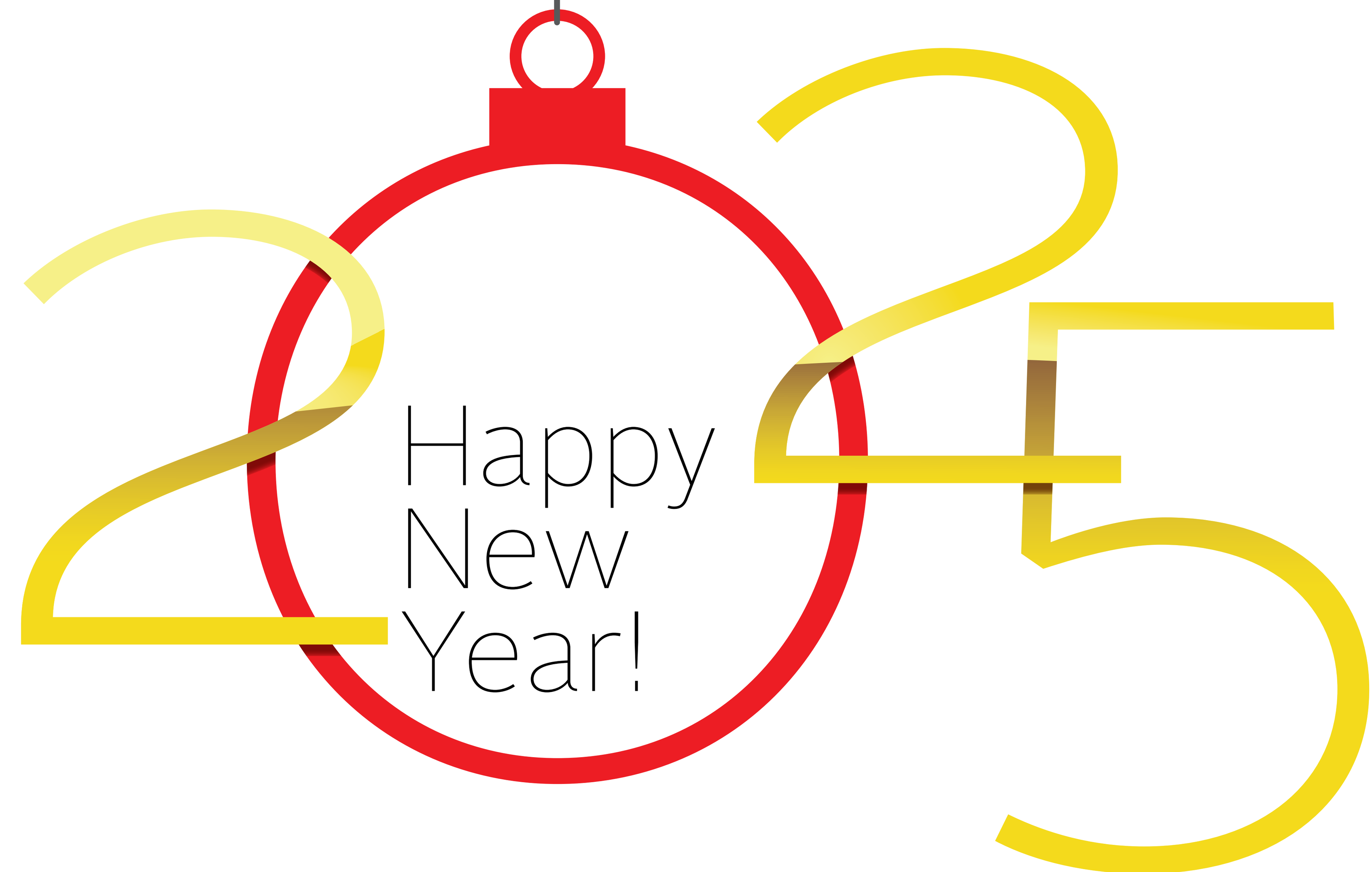
Editor-at-Large • James Maurer
maurer.jim@gmail.com

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

Copyright © 2024 by FreeBSD Foundation. All rights reserved.
This magazine may not be reproduced in whole or in part without written permission from the publisher.

LETTER
from the Foundation



FreeBSD JOURNAL
November/December 2024

Virtualization Issue

8 Character Device Driver Tutorial (Part 2)
By John Baldwin

22 BHYVE for the Windows and Linux Users
By Jason Tubnor

27 Xen and FreeBSD
By Roger Pau Monné

34 Wifibox: An Embedded Virtualized Wireless Router
By Gábor Páli

3 Foundation Letter

5 We Get Letters
By Michael W. Lucas

42 Embedded FreeBSD: Fabric—Baby Steps
By Christopher R. Bowman

46 Adventures in TCP/IP:
Dynamic Goodput Pacing: A New Approach to Packet Pacing
By Randall Stewart

55 Conference Report: My EuroBSDCon Experience in Dublin
By Stefano Marinelli

63 Events Calendar
By Anne Dickison

WeGet letters

by Michael W Lucas

Dear Letters Person,

Every day someone figures out new types of virtualization or ways to complicate it. Full virtualization, light virtualization, containers, ABI compatibility, it just goes on and on. Where does this end? How can I get ahead of this?

—Racing Ahead of Virtualization Every Day

PS: does the *Journal* have a new person for this column yet, or am I going to get a completely unhelpful rant?

Dear RAVED,

Anyone who believes my rants are unhelpful is new to IT. Once your trauma helps you develop triggers, you'll understand. Give it another week.

Business schools teach the importance of "getting ahead" of a trend. That's utterly inapplicable to sysadmins. We not only are the trend, we spend our spare time dreaming up ways to make the trend both steeper and even more trendy.

We continuously delude ourselves that we can make things better, when we can only make things differently bad.

Business schools teach the importance of "getting ahead" of a trend. That's utterly inapplicable to sysadmins.

The server has one hard drive? Better mirror that, or RAID-5 it, or fibre channel it off to the NAS where it becomes someone else's problem. Bloatware runs slowly? Trim the program or add memory or install a fast caching disk. Cosmic ray filesystem corruption? A zraid3 array with copies=99 will fix that! Every "improvement" adds failure modes. Even senior sysadmins, who understand in their marrow Rule of System Administration #16 (*The impulse to improve is the leading cause of failure*) fall prey to this fallacy.

Virtualization is merely another expression of such.

This isn't new. Remember chroots? When the Internet converted from a private educational network to a tangle of private enterprises, much of the core software was revealed to have critical security problems. Nobody on the private Internet wanted to destroy the private network, but once it went public a handful of people out in the world liked setting nice things on fire. (Looking at the modern Internet, I can't help wondering if some of those

early attackers were time travelers come back to save us from ourselves.) If a program suffered from shell escapes, you could run it from a directory that contained only the files the program needed. You can't have a shell escape if you don't have a shell to escape to, after all! This is both a clever hack and a prime example of solving the wrong problem.

FreeBSD's jails were first conceived of as enhanced chroots. What if you took that locked-in directory, gave it an IP address and its own process space? It'd look like a full system but wholly contained within another system! We could even give that super-chroot the ability to run its own child super-chroots! It's beautiful, and elegant, and complex enough to encourage not only failures, but exciting new failure modes you've never previously experienced.

But there's good news, jail fans! Chances are you're using your jail for a specific task. It only needs a small selection of the base system, not a complete userland. OccamBSD is quite usable, and lets you install only the necessary system components. Where jails were an improvement over chroots, chroots are now an improvement over jails!

The impulse to improve.

Or the Linuxulator. You can take a FreeBSD system and have it run Linux programs. Linux is just a kernel, after all. Install your least loathed Linux userland in a directory, chroot your programs into that directory, and run as if they were a Linux system. That'll save you from installing an extra server just to run Linux because (real talk here) who wants to run a Unix without ZFS or PF? Many developers use the Linuxulator as an intermediate step to port Linux software to FreeBSD. An easy win, right?

You can take a FreeBSD system and have it run Linux programs. Linux is just a kernel, after all.

Sure. Linux mode is almost entirely compatible with Linux. "Almost entirely compatible" is like "almost leprosy-free;" good for you, but I ain't touching it. I'm better off licking armadillos.

These aren't enough, though. We want to further optimize our virtualization, so we add in unionfs and base jails. Optimize! Never mind today's endless infinite oceans of disk space. Yes, yes, your big data application needs a storage array, but you can fit thousands of OS installs on an NVMe disk and upgrade them all with a simple script. Or you can have the optimally arranged base jails upgrade the One True Jail with a simple script and have another simple script that runs through all your OS installs and applies the needed upgrades to each. Nobody's willing to address the real problem and design systems that don't need upgrades, because that would put the entire computer industry out of business and who would poison the planet then?

Heavy virtualization? Bhyve, qemu, libvirt, all of those? Mere super-jails that add CPU, process, and filesystem isolation. More secure? There's a reason security professionals say, "another day, another hypervisor escape."

Virtualization allows endless opportunities for not mere failure, but debacle. The tangle of complex interconnected systems creates an exponential climb of interactions, a rising arc that peaks at a number too large for that lump of mildly electrified pudding in your skull to process. You're a sysadmin. You'll sit down and contemplate how these systems might fail. Some failure modes are obvious: fire, flood, famine. Some less so: what if

System V IPC communications leak between two particular jails, leaking company secrets into the world? On second thought, hang onto that excuse; it'll be useful when you decide to blow the whistle on your unethical employer. What, you claim they're ethical? Then how are they making the money to pay you? Don't worry about it, when it becomes impossible to ignore, you're prepared to leak.

The apparent "problem" is, of course, that modern hardware is ridiculously overpowered for most tasks. I rent a small, dedicated server. It runs jails and bhyve VMs. The most resource-intensive application I run is my own email server. Email takes very few resources, but rspamd takes everything you give it. The hardware is mostly idle, so I pile more functions on it. Because it's *there*. Why not try to make things a little better, and use my resources?

Having too much computing power is not a problem in any reasonable sense of the word. But I did it anyway.

So: getting ahead of the trend? You mean get ahead of this nightmarish tangle of optimizations?

Yes, there is a way to "get ahead of the trend."

Fail faster.

Good luck. I have faith in you.

Have a question for Michael?
Send it to letters@freebsdjournal.org

MICHAEL W LUCAS' latest books include *Dear Abyss* (a collection of these columns), *Run Your Own Mail Server*, and *Apocalypse Moi*. See more at https://mwl.io.

Books that will help you. Or not.

"While we appreciate Mr Lucas' unique contributions to the Journal, we do feel his specific talents are not being fully utilized. Please buy his books, his hours, autographed photos, whatever, so that he is otherwise engaged."

— John Baldwin
FreeBSD Journal Editorial Board Chair

https://mwl.io

Character Device Driver Tutorial (Part 2)

BY JOHN BALDWIN

In the previous article in this three-part series, we built a simple character device driver that permitted I/O operations backed by a fixed buffer. In this article, we will extend this driver to support a FIFO data buffer along with support for non-blocking I/O and event reporting. The full source of each version of the device driver can be found at https://github.com/bsdjhb/cdev_tutorial.

Before moving forward though, we must address an unintended bit of unfinished business from the previous article. Alert reader Virus-V noted that the final version of the echo driver from the article did not destroy the `/dev/echo` device during unload and that accessing the device after unload triggered a kernel panic. One of the first clues to the bug lay in a warning message about leaked memory from the kernel emitted during module unload prior to the panic. As noted in the prior article, this warning is one of the reasons kernel modules should use a dedicated malloc type when possible. The bug lay in the `echodev_create()` function added in the last set of changes. We had failed to return a pointer to the newly allocated softc structure to the caller by storing the value in `*scp`. As a result, the `echo_softc` variable was always NULL and the softc was not destroyed during module unload. The fix was a one line addition to `echodev_create()` to store the pointer to the new softc in `*scp` on success.

The echo driver from the first article used a flat data buffer for I/O operations.

Using a FIFO Data Buffer

The echo driver from the first article used a flat data buffer for I/O operations. Reads and writes could access any region of the data buffer, and the entire range of the data buffer was always valid. These semantics are similar to accessing a file that does not grow when written beyond the end. A character device driver is free to implement a range of semantics, however. For this article, we will alter the echo driver to treat user I/O data like a FIFO stream device similar to a `pipe` or `fifo`. I/O write requests will append data to the tail of a logical data buffer and read requests will read data from the head of this data buffer. File offsets such as those used with `pread(2)` will be ignored. The driver will continue to use an in-kernel buffer to hold a temporary copy of user data. Writes will store data in this buffer and reads will consume data from this buffer. This means that the driver will now need to keep

track of the amount of valid data in the buffer as well as the buffer's length. To simplify the implementation, the start of the data buffer will always be treated as the buffer's head. Read requests that read a subset of the available data will copy the remaining data to the front of the buffer.

This does raise several additional questions, however. First, how should reads that want to read more data from the buffer than is available be handled? Second, how should writes that want to store more data than the buffer can hold be handled? For simplicity, we will start by returning a short read of whatever bytes are available for read requests, and truncating write requests to only store the amount of data for which there is room in the buffer. Third, what should an `ECHODEV_SBUFSIZE` request do that shrinks a buffer smaller than the amount of valid data in the buffer? We have chosen to fail such a request with an error. One could choose to discard some of the data instead, but one would have to decide which data to discard. Listing 1 provides the updated read and write methods. Note that a new `valid` member has been added to the softc to track the amount of valid data in the buffer. Example 1 demonstrates a few scenarios of this updated driver. Initially, the device is empty, but data can be read once input is provided. The last few commands read a series of bytes across two requests.

Listing 1: Read and Write Using a FIFO Data Buffer

```
static int
echo_read(struct cdev *dev, struct uio *uio, int ioflag)
{
	struct echodev_softc *sc = dev->si_drv1;
	size_t todo;
	int error;

	sx_xlock(&sc->lock);
	todo = MIN(uio->uio_resid, sc->valid);
	error = uiomove(sc->buf, todo, uio);
	if (error == 0) {
		sc->valid -= todo;
		memmove(sc->buf, sc->buf + todo, sc->valid);
	}
	sx_xunlock(&sc->lock);
	return (error);
}

static int
echo_write(struct cdev *dev, struct uio *uio, int ioflag)
{
	struct echodev_softc *sc = dev->si_drv1;
	size_t todo;
	int error;

	sx_xlock(&sc->lock);
	todo = MIN(uio->uio_resid, sc->len - sc->valid);
```

```
        error = uiomove(sc->buf + sc->valid, todo, uio);
        if (error == 0)
                sc->valid += todo;
        sx_xunlock(&sc->lock);
        return (error);
}
```

Example 1: Simple FIFO I/O

```
# hd < /dev/echo

# echo "foo" > /dev/echo
# cat /dev/echo
foo
# echo "12345678" > /dev/echo
# dd if=/dev/echo bs=1 count=4 status=none | hd
00000000  31 32 33 34                                       |1234|
00000004
# cat /dev/echo
5678
```

Blocking I/O

While this version of the echo driver does implement a simple data stream, it has some limitations. If a process wants to use this device to share a block of data larger than the data buffer, it has to wait to write an additional buffer's worth of data until a reader has consumed the previous buffer's worth of data. This requires the writing process to either coordinate with the reader process(es) or to use a timer and retry write operations periodically. Neither of these solutions are very practical. Instead, the driver can permit larger writes by sleeping in the write request while the buffer is full until the request is completed. Readers would awaken the waiting writer when space is available permitting the writer to make additional progress. Similarly, readers could block waiting for data to return. To more closely match the semantics of pipes and sockets, we have chosen to make read requests only block at the start of a request and return a short read as soon as data is available. However, for writes, we attempt to drain the entire buffer. To handle blocking, we use the `sx_sleep(9)` function which atomically releases our device's lock while putting the current thread to sleep. Passing `PCATCH` to this function permits signals to interrupt this sleep in which case `sx_sleep()` will return a non-zero error value. Listing 2 shows the updated read method. The write method is similarly updated but with an extra loop to retry until the write has fully completed. Note that in the write method, we do not have to "hide" errors if the write partially completes. The generic write system call handling in the `dofilewrite()` function maps errors from `sx_sleep()` to success if at least some data was written. Some of the ioctl handlers also require updates to awaken sleeping writers when the buffer grows in size or has its contents cleared.

When testing this version of the driver, Example 2 shows some possibly surprising behavior. While the data previously written is returned, `cat(1)` continues to wait for additional data until killed with a signal.

Listing 2: Blocking Read Method

```
static int
echo_read(struct cdev *dev, struct uio *uio, int ioflag)
{
	struct echodev_softc *sc = dev->si_drv1;
	size_t todo;
	int error;

	if (uio->uio_resid == 0)
		return (0);

	sx_xlock(&sc->lock);

	/* Wait for bytes to read. */
	while (sc->valid == 0) {
		error = sx_sleep(sc, &sc->lock, PCATCH, "echord", 0);
		if (error != 0) {
		sx_xunlock(&sc->lock);
		return (error);
		}
	}

	todo = MIN(uio->uio_resid, sc->valid);
	error = uiomove(sc->buf, todo, uio);
	if (error == 0) {
		/* Wakeup any waiting writers. */
		if (sc->valid == sc->len)
		wakeup(sc);

		sc->valid -= todo;
		memmove(sc->buf, sc->buf + todo, sc->valid);
	}
	sx_xunlock(&sc->lock);
	return (error);
}
```

Example 2: Blocking I/O Hangs Forever

```
# echo "12345678" > /dev/echo
# cat /dev/echo
12345678
^C
```

Unloading Sanely

We will get to the surprising behavior from Example 2 shortly. The current driver has another surprise. Unloading the driver while a process is blocked in the read or write methods

will hang the process unloading the module until the first process is killed with a signal. This is not the behavior an administrator expects when unloading a module. Instead, the echo driver should awaken any sleeping threads during device destruction and ensure they will return from the driver methods without sleeping again. To support this, the next change adds a **`dying`** flag to the **`softc`** and the read and write methods fail with an `ENXIO` error rather than blocking if this flag is set. During device destruction, the **`dying`** flag is set and sleeping threads are awakened before calling **`destroy_dev()`**. Listing 3 shows the changed lines in the read method and the updated **`echodev_destroy()`** function.

Listing 3: Waking Threads on Unload

```
static int
echo_read(struct cdev *dev, struct uio *uio, int ioflag)
{
        ...
        /* Wait for bytes to read. */
        while (sc->valid == 0) {
                if (sc->dying)
                        error = ENXIO;
                else
                        error = sx_sleep(sc, &sc->lock, PCATCH, "echord", 0);
                if (error != 0) {
                        sx_xunlock(&sc->lock);
                        return (error);
                }
        }
        ...
}

...

static void
echodev_destroy(struct echodev_softc *sc)
{
        if (sc->dev != NULL) {
                /* Force any sleeping threads to exit the driver. */
                sx_xlock(&sc->lock);
                sc->dying = true;
                wakeup(sc);
                sx_xunlock(&sc->lock);

                destroy_dev(sc->dev);
        }
        free(sc->buf, M_ECHODEV);
        sx_destroy(&sc->lock);
        free(sc, M_ECHODEV);
}
```

Conditional Blocking on Read

In Example 2, it was surprising that `cat(1)` continued to block after reading the available data from the device given our prior examples. However, this behavior does follow naturally from our driver since `cat(1)` just calls [`read(2)`]() in a loop until it receives EOF and the second call to `read(2)` blocks waiting for more data. The semantic used by other stream devices like pipes and fifos is that reads will return EOF instead of blocking if no process has the device open for writing. If there are devices open for writing, reads will block waiting for more data.

We can implement these semantics in the echo driver easily. We add a count of writers to the softc and only block in the read method if this count is non-zero. To detect writers, we add an open method that increments the counter for each open which requests write permission. This can be determined by checking for the `FWRITE` flag in the file flags passed to the open method. A new close method decrements count when a writer closes. By default, the close character device switch method is only called for the last close of a device when no remaining file descriptors remain. Instead, we set the `D_TRACKCLOSE` character device switch flag so that the close method is called each time a file descriptor is closed. The close method awakens any waiting readers if the last writer is closed. Listing 4 shows the new open and close methods as well as the changed line in the read method. Retrying the steps from Example 2 no longer results in the surprising behavior as `cat(1)` now exits after reading the available data.

Listing 4: Tracking Open Writers

```
static int
echo_open(struct cdev *dev, int fflag, int devtype, struct thread *td)
{
	struct echodev_softc *sc = dev->si_drv1;

	if ((fflag & FWRITE) != 0) {
		/* Increase the number of writers. */
		sx_xlock(&sc->lock);
		if (sc->writers == UINT_MAX) {
			sx_xunlock(&sc->lock);
			return (EBUSY);
		}
		sc->writers++;
		sx_xunlock(&sc->lock);
	}
	return (0);
}

static int
echo_close(struct cdev *dev, int fflag, int devtype, struct thread *td)
{
	struct echodev_softc *sc = dev->si_drv1;

	if ((fflag & FWRITE) != 0) {
		sx_xlock(&sc->lock);
		sc->writers--;
```

```
                if (sc->writers == 0) {
                        /* Wakeup any waiting readers. */
                        wakeup(sc);
                }
                sx_xunlock(&sc->lock);
        }
        return (0);
}

static int
echo_read(struct cdev *dev, struct uio *uio, int ioflag)
{
        ...
        /* Wait for bytes to read. */
        while (sc->valid == 0 && sc->writers != 0) {
        ...
}
```

Non-blocking I/O

Now our echo device supports blocking I/O. However, some consumers may wish to use non-blocking I/O. A process can request non-blocking I/O either by passing the `O_NONBLOCK` flag to [open(2)](), or toggling the `O_NONBLOCK` flag on an open file descriptor via [fcntl(2)](). A character device driver can check if non-blocking I/O is enabled by checking for the `O_NONBLOCK` flag in the file flags passed to the read and write methods. If non-blocking I/O is requested, the error `EWOULDBLOCK` should be returned instead of blocking any time the driver would block. For the echo device, this means adding extra checks before blocking in the read and write methods. That alone is sufficient to handle non-blocking I/O requested at open time. However, to support toggling flags via `fcntl(2)`, an additional step is required.

The I/O control handler should return zero if the requested flag setting is supported.

Every attempt to set file flags via the `fcntl(2)` `F_SETFL` operation invokes two I/O control commands on a character device: `FIONBIO` and `FIOASYNC`. Even requests that do not change the state of the associated `O_NONBLOCK` and `O_ASYNC` flags invoke these I/O control commands. If either I/O control command fails, the entire `F_SETFL` operation fails, and the file flags remain unchanged. As a result, a character device driver that wants to support `F_SETFL` must implement support for both I/O control commands.

`FIONBIO` and `FIOASYNC` pass an int as the command argument. This int value is zero if the associated file flag is clear in the new file flags or non-zero if the associated flag is set in the new file flags. The I/O control handler should return zero if the requested flag setting is supported, or an error if the requested setting is not supported. The echo device supports either setting for the `O_NONBLOCK` flag but does not support setting the `O_ASYNC` flag, so the `FIOASYNC` handler for the echo device fails if the int argument is non-zero.

Listing 5 shows the relevant changes to the read and I/O control methods to support non-blocking I/O.

Listing 5: Support for Non-Blocking I/O

```
static int
echo_read(struct cdev *dev, struct uio *uio, int ioflag)
{
	...
	/* Wait for bytes to read. */
	while (sc->valid == 0 && sc->writers != 0) {
		if (sc->dying)
			error = ENXIO;
		else if (ioflag & O_NONBLOCK)
			error = EWOULDBLOCK;
		else
			error = sx_sleep(sc, &sc->lock, PCATCH, "echord", 0);
	...
}

...

static int
echo_ioctl(struct cdev *dev, u_long cmd, caddr_t data, int fflag,
    struct thread *td)
{
	...
	switch (cmd) {
	...
	case FIONBIO:
		/* O_NONBLOCK is supported. */
		error = 0;
		break;
	case FIOASYNC:
		/* O_ASYNC is not supported. */
		if (*(int *)data != 0)
			error = EINVAL;
		else
			error = 0;
		break;
	...
}
```

Polling I/O Status

Applications that use non-blocking I/O often use an event loop to service requests for multiple file descriptors. For each iteration of the loop, the application blocks waiting for

one or more file descriptors to be ready (for example, having data available to read, or room for more data to be written). The application then services each of the ready file descriptors before waiting again. FreeBSD supports two system calls for this type of event loop: `select(2)` and `poll(2)`. In the kernel, `select(2)` and `poll(2)` are implemented using a common framework. Each requested file descriptor is polled individually to determine if it is ready. If no file descriptors are ready, the thread invoking the system call can sleep waiting for at least one of the file descriptors to become ready. If the file descriptor becomes ready while a thread is waiting, the file descriptor must awaken the sleeping thread.

The `selrecord(9)` family of functions manages the sleeping and awakening of threads. A file descriptor that supports polling must create a **`struct selinfo`** object for type of event it supports. The object should be initialized by clearing the entire object with zeroes (for example, using **`memset()`**). If the file descriptor's poll function finds that a file descriptor is not ready, it must call **`selrecord()`** on the associated **`struct selinfo`** object for each requested event. Any time an event occurs that could make a file descriptor ready, **`selwakeup()`** must be called on the **`struct selinfo`** object for that event. Finally, **`seldrain()`** should be used to awaken any remaining threads before destroying a **`struct selinfo`** object.

For the echo device we support both read and write events.

For character devices, the file descriptor polling function invokes the character device poll method. This method accepts a bitmask of `poll(2)` events as a function argument and must return a mask of those events that are currently true. In addition, this function is responsible for calling **`selrecord()`** if none of the requested events are true. Note that character devices do not support different types of priority data via the read and write methods, only normal data.

For the echo device we support both read and write events. We add two **`struct selinfo`** objects to the softc, one for each event. Since the entire softc is zeroed on creation, no further changes are required when initializing the softc. Each of the read and write methods calls **`selwakeup()`** for the *other* event to awaken any threads that might be waiting. A few other places can also make the echo device ready as well and need calls to **`selwakeup()`**. If an I/O control command grows the buffer or clears its contents, that may make the device ready to write. If the last writer closes the device, that can also make the device ready to read. A new poll method determines the current status of the device and calls **`selrecord()`** as needed. Finally, **`seldrain()`** is called for each event when destroying the device. Listing 6 shows the added call to **`selwakeup()`** in the read method as well as the new poll method. Note that for the read method, **`selwakeup()`** is used for the write event.

Listing 7: Device Polling

```
static int
echo_read(struct cdev *dev, struct uio *uio, int ioflag)
{
        ...
        error = uiomove(sc->buf, todo, uio);
        if (error == 0) {
                /* Wakeup any waiting writers. */
```

```
        if (sc->valid == sc->len)
            wakeup(sc);

        sc->valid -= todo;
        memmove(sc->buf, sc->buf + todo, sc->valid);
        selwakeup(&sc->wsel);
    }
    ...
}

...

static int
echo_poll(struct cdev *dev, int events, struct thread *td)
{
    struct echodev_softc *sc = dev->si_drv1;
    int revents;

    revents = 0;
    sx_slock(&sc->lock);
    if (sc->valid != 0 || sc->writers == 0)
        revents |= events & (POLLIN | POLLRDNORM);
    if (sc->valid < sc->len)
        revents |= events & (POLLOUT | POLLWRNORM);
    if (revents == 0) {
        if ((events & (POLLIN | POLLRDNORM)) != 0)
            selrecord(td, &sc->rsel);
        if ((events & (POLLOUT | POLLWRNORM)) != 0)
            selrecord(td, &sc->wsel);
    }
    sx_sunlock(&sc->lock);
    return (revents);
}
```

A pair of generic I/O control commands are also useful for inspecting the status of a file descriptor. `FIONREAD` and `FIONWRITE` return the number of bytes that can be read or written without blocking, respectively. The byte count is returned in a control command argument of type int. Listing 8 shows the support for these I/O control commands in the echo device. Note that the returned value is clamped to `INT_MAX` to avoid overflow.

Listing 8: FIONREAD and FIONWRITE

```
static int
echo_ioctl(struct cdev *dev, u_long cmd, caddr_t data, int fflag,
    struct thread *td)
{
    ...
    switch (cmd) {
```

```
        ...
        case FIONREAD:
                sx_slock(&sc->lock);
                *(int *)data = MIN(INT_MAX, sc->valid);
                sx_sunlock(&sc->lock);
                error = 0;
                break;
        case FIONWRITE:
                sx_slock(&sc->lock);
                *(int *)data = MIN(INT_MAX, sc->len - sc->valid);
                sx_sunlock(&sc->lock);
                error = 0;
                break;
        ...
}
```

To make this functionality easier to demonstrate, we have added a new `poll` command to the echoctl utility. This command uses `poll(2)` to query the echo device's current status. If the device is readable, it uses the `FIONREAD` I/O control command to output the number of bytes available to read. If the device is writable, it uses `FIONWRITE` to output the number of bytes that can be written. Example 3 shows a few invocations of this command along with other operations on the echo device. Note that since there is not another writer in this example, the device is readable even while it is empty.

Example 3: Polling I/O Status

```
# echoctl poll
Returned events: POLLIN|POLLOUT
0 bytes available to read
room to write 64 bytes
# echo "foo" > /dev/echo
# echoctl poll
Returned events: POLLIN|POLLOUT
4 bytes available to read
room to write 60 bytes
# cat /dev/echo
foo
# echoctl poll -r
Returned events: POLLIN
0 bytes available to read
```

I/O Status Reporting via `kqueue(2)`

FreeBSD provides the [`kqueue(2)`] kernel event notification facility as a separate API from `select(2)` and `poll(2)`. With `kqueue(2)`, applications register a persistent note in the kernel for each desired event. The kernel generates a stream of events that the application can consume and act upon. Unlike `select(2)` and `poll(2)`, an application does not need to register all the events it cares about each time it wants to wait for a new event. This reduces overhead in applications while also permitting more efficient tracking of desired events in the kernel.

A kernel event consists of a filter (event type) and identifier. The behavior of some events can be further customized via various flags. For I/O on file descriptors, the two primary filters are `EVFILT_READ` and `EVFILT_WRITE` to determine if a descriptor is readable or writable, respectively. The identifier field for these event filters is the integer file descriptor. In addition, for read and write events the kernel event structure returns the amount of data that can be read or written as a separate field. This avoids the need for separate invocations of the `FIONREAD` and `FIONWRITE` I/O control commands.

Inside the kernel, kernel events are described by a **`struct knote`** object. This structure contains copies of the event fields that are used to generate the events returned to the application. A list of active events is stored in a **`struct knlist`** object. Since I/O events handled by `select(2)` and `poll(2)` are usually associated with a kernel event, **`struct selinfo`** embeds a **`struct knlist`** as its **`si_note`** member. Each knote is also associated with a **`struct filterops`** object pointed to by the **`kn_fop`** member. This structure and the APIs for manipulating knotes and knote lists are described in [`kqueue(9)`].

For character devices, the kqfilter method is responsible for attaching a **`struct filterops`** object to a knote. This includes setting the **`kn_fop`** member and adding the knote onto the correct knote list. As a result, **`struct filterops`** objects for character devices do not use the **`f_attach`** member. The **`kn_hook`** member of **`struct knote`** is an opaque pointer that can be set by the kqfilter method to pass state to **`struct filteropts`** methods similar to the **`si_drv1`** field of `struct cdev`.

For the echo driver, we define two **`struct filteropts`** objects: one for read events and one for write events. Each event includes **`f_detach`** and **`f_event`** methods. We reuse the embedded knote list in the existing read and write **`struct selinfo`** objects from the softc. Since the echo driver uses an sx(9) lock, we define custom locking callbacks for use with **`knlist_init()`** when creating the echo device. The **`f_detach`** methods use `knlist_remove()` to remove a knote from the associated knote list. The **`f_event`** methods set the **`kn_data`** field to the appropriate byte count and mark the event ready if the byte count is non-zero. The **`f_event`** method for the read event also sets `EV_EOF` if there are no writers. A new kqfilter character device method attaches a knote to the new **`struct filterops`** objects for the `EVFILT_READ` and `EVFILT_WRITE` filters. It also sets the **`kn_hook`** member of the knote to the softc pointer. Finally, all the places in the driver that call `selrecord()` to awaken sleeping threads from `poll(2)` or `select(2)` now also call `KNOTE_LOCKED()` to report an event for knotes associated with the read or write event. Listing 9 shows the **`struct filterops`** object and its associated methods for the read filter. Listing 10 shows the new kqfilter character device method.

Listing 9: EVFILT_READ Filter

```
static struct filterops echo_read_filterops = {
	.f_isfd =	1,
	.f_detach = echo_kqread_detach,
	.f_event =	echo_kqread_event
};

...

static void
echo_kqread_detach(struct knote *kn)
```

```
{
	struct echodev_softc *sc = kn->kn_hook;

	knlist_remove(&sc->rsel.si_note, kn, 0);
}

static int
echo_kqread_event(struct knote *kn, long hint)
{
	struct echodev_softc *sc = kn->kn_hook;

	kn->kn_data = sc->valid;
	if (sc->writers == 0) {
		kn->kn_flags |= EV_EOF;
		return (1);
	}
	kn->kn_flags &= ~EV_EOF;
	return (kn->kn_data > 0);
}
```

Listing 10: kqfilter Device Method

```
static int
echo_kqfilter(struct cdev *dev, struct knote *kn)
{
	struct echodev_softc *sc = dev->si_drv1;

	switch (kn->kn_filter) {
	case EVFILT_READ:
		kn->kn_fop = &echo_read_filterops;
		kn->kn_hook = sc;
		knlist_add(&sc->rsel.si_note, kn, 0);
		return (0);
	case EVFILT_WRITE:
		kn->kn_fop = &echo_write_filterops;
		kn->kn_hook = sc;
		knlist_add(&sc->wsel.si_note, kn, 0);
		return (0);
	default:
		return (EINVAL);
	}
}
```

As with the `poll(2)` support, we have extended the echoctl utility with another command to demonstrate the `kevent(2)` support. The new events command registers read and write events for the echo device and outputs a line for each event that is received. Since read and write events are level-triggered by default, echoctl sets the `EV_CLEAR` flag when registering events for the echo device. This instead only reports events when the device

state changes triggering a call to `KNOTE_LOCKED()` inside the driver. Example 4 shows the output of the events command across a series of actions. The first two events are reported as the initial state when the echo device is idle without any open readers or writers. In another shell we execute the command `"jot -c -s "" 80 48 > /dev/echo"` to write 81 bytes of data to the echo device. Since the default buffer size is 64 bytes, this command blocks in the `write(2)` system call after writing 64 bytes. The write of 64 bytes triggers the next `EVFILT_READ` event reporting 64 bytes available to read. Finally, in a third shell we execute the command "`cat /dev/echo`" to read all the data from the echo device. The first `read(2)` system call from `cat(1)` reads the 64 bytes of output and triggers an `EVFILT_WRITE` event. However, before the echoctl process can query the echo device's state, the `jot(1)` process has awakened and written the remaining 17 bytes of data to the buffer leaving room for 47 bytes. This accounts for the first `EVFILT_WRITE` event reported in the third block of events. The write of the remaining 17 bytes also triggered an `EVFILT_READ` event. However, by the time this event is reported, the `jot(1)` process has exited and `cat(1)` has read the remaining 17 bytes, so the `EVFILT_READ` event reports `EV_EOF` with zero bytes to read. The read of 17 bytes by `cat(1)` also triggered an `EVFILT_WRITE` event that is reported as the next to last event. Finally, `cat(1)` calls `read(2)` a third time which returns 0 to signal EOF. This read also triggers an `EVFILT_WRITE` event which is reported as the last event. This last sequence of events is not deterministic and may appear in a different order or with slightly different values across different runs (for example, the first `EVFILT_WRITE` may report 64 bytes available to write if `jot(1)` hasn't yet written the remaining 17 bytes).

Example 4: I/O Status via Kernel Events

```
# echoctl events -W
EVFILT_READ: EV_EOF 0 bytes
EVFILT_WRITE: 64 bytes
...
EVFILT_READ: 64 bytes
...
EVFILT_WRITE: 47 bytes
EVFILT_READ: EV_EOF 0 bytes
EVFILT_WRITE: 64 bytes
EVFILT_WRITE: 64 bytes
```

Conclusion

In this article we extended the echo device to support a FIFO data buffer with both blocking and non-blocking I/O. We also added support for querying device state via `poll(2)` and `kevent(2)`. The final article in this series will describe how character devices can provide a backing store for memory mappings created via `mmap(2)`.

JOHN BALDWIN is a systems software developer. He has directly committed changes to the FreeBSD operating system for over twenty years across various parts of the kernel (including x86 platform support, SMP, various device drivers, and the virtual memory subsystem) and userspace programs. In addition to writing code, John has served on the FreeBSD core and release engineering teams. He has also contributed to the GDB debugger. John lives in Ashland, Virginia with his wife, Kimberly, and three children: Janelle, Evan, and Bella.

BHYVE FOR THE LINUX AND WINDOWS USERS

BY JASON TUBNOR

The FreeBSD bhyve hypervisor was announced to the world in May 2011 by Neel Natu and Peter Grehan and then gifted to FreeBSD from NetApp. This finally gave FreeBSD something to compete against the Linux KVM hypervisor. However, there were further benefits, it is small and robust as well as being performant, leaning heavily on CPU instruction sets rather than dealing with interpretation.

The initial implementation was only suitable for FreeBSD guests, and it was some time before we saw bhvye able to run other operating systems.

First up, there was Linux, and then there was a way to repackage Windows 8 or Windows Server 2012 to get them to install. This was too much for a regular user to manage and it wasn't until the arrival of the bhyve UEFI boot feature that things really took off.

UEFI boot was the killer feature that bhyve had been waiting for. This allowed for a wide range of operating systems to be installed and run on FreeBSD bhyve. When FreeBSD 11 was released, we finally had a virtualization component on par with other operating systems.

While UEFI booting was the killer feature for bhyve, the killer app for bhyve was Windows Server 2016. This was the turning point when enterprises could take bhyve and Windows in a vanilla format and have a reliable enterprise hypervisor to run business workloads in a stable fashion.

Suddenly, businesses were able to deploy equipment far and wide with a solution that was 2-clause BSD licensed and be able to tune — either via hardware or software — the hypervisor to solve their problems.

Suddenly, businesses were able to deploy equipment far and wide.

There was still a problem, however, because Windows required numerous drivers to be installed either in-image or after installation to avoid performance issues. In July 2018, this was partially solved by the implementation of the PCI-NVMe storage emulation, eventually giving bhyve the edge over KVM in storage performance for general workloads.

Today, Windows running on bhyve still requires at least the VirtIO-net drivers from RedHat to allow for network transfers to be reliable and exceed 1Gb/s. There are other drivers within the applicable Windows MSI package that is available from RedHat, and these are recommended to be loaded prior to production implementations. For Linux, most distributions have all applicable drivers, including AlmaLinux, which we are using in this article. It is possible and recommended to use NVMe emulated backed storage for Linux installations, however, it is quite difficult to configure KVM to use an emulated NVMe storage type and if you plan to move Linux workloads between bhyve and KVM, it is recommended that you set your guest to simply use VirtIO-blk storage.

The following will work for a standard FreeBSD workstation in a typical type-2 hypervisor configuration or for a dedicated FreeBSD server that is only hosting guest workloads with the applicable storage and network associated with the guests for a type-1 hypervisor.

Preparation

Typically, all modern processors from the last ten years will be suitable to use with bhyve virtualization.

Ensure that your hardware is configured with the virtualization technology enabled along with VT-d support enabled. The use of PCI pass-through is out of scope for this article, but it is recommended to enable VT-d so that it can be used when needed. After this is configured in your machine's BIOS/firmware, you can check that it visible to FreeBSD by looking for POPCNT in the Features2 of the CPU:

```
# dmesg | grep Features2
  Features2=0x7ffafbff<SSE3,PCLMULQDQ,DTES64,MON,DS_CPL,VMX,SMX,EST,TM2,SSSE3,SDBG,FMA,
  CX16,xTPR,PDCM,PCID,SSE4.1,SSE4.2,x2APIC,MOVBE,POPCNT,TSCDLT,AESNI,XSAVE,OSXSAVE,AVX,
  F16C,RDRAND>
```

Now we have confirmed that our CPU is ready, we need to install a few packages to make it easy to create and manage guest operating systems:

```
pkg install openntpd vm-bhyve bhyve-firmware
```

Briefly, OpenNTPD is a simple time daemon from the OpenBSD project. This keeps the host time from skewing. When a hypervisor is under extreme pressure from guest workloads, this can cause the regular system time to quickly get out of sync. OpenNTPD keeps time in check while ensuring your upstream time source is reporting the correct time using constraints over the HTTPS protocol. bhyve-firmware is the meta package that will load the most recently supported EDK2 Firmware for bhyve from packages. Finally, vm-bhyve is a management system for bhyve written in shell, avoiding the need for complex dependencies.

Boot strapping a machine ready for use with vm-bhyve is quite simple but attention is required for some of the ZFS options to ensure that guests remain performant on underlying storage for general workloads:

```
# zfs create -o mountpoint=/vm -o recordsize=64k zroot/vm
# cat <<EOF >> /etc/rc.conf
vm_enable="YES"
vm_dir="zfs:zroot/vm"
vm_list=""
vm_delay="30"
EOF
# vm init
```

Before we get too far into this, we should download the ISOs that we will be using later so they are ready for use by the vm-bhyve installer. To download an ISO to the vm-bhyve ISO store, use the **`vm iso`** command:

```
# vm iso https://files.bsd.engineer/Windows11-bhyve.iso
```

(sha256 - 46c6e0128d1123d5c682dfc698670e43081f6b48fcb230681512edda216d3325)

```
# vm iso https://repo.almalinux.org/almalinux/9.5/isos/x86_64/AlmaLinux-9.5-x86_64-dvd.iso
```

(sha256 - 3947accd140a2a1833b1ef2c811f8c0d48cd27624cad343992f86cfabd2474c9)

These will be downloaded into the /vm/.iso directory. Note: The AlmaLinux ISO is directly downloaded from the project and checksums can be verified upstream. The Windows11-bhyve ISO was downloaded from Microsoft and has been modified to ensure that it will install on hardware that Microsoft deems `unsupported` and has been provided to assist with this article. As such, this ISO should only be used in a lab environment. It has had the CPU and TPM requirements removed along with not needing to create a Microsoft account.

Networking

By default, vm-bhyve uses bridges to connect the systems physical interface with the tap interfaces that are assigned to each guest. When adding a physical interface to a bridge, certain features such as TCP Segment Offload (TSO) and Large Receive Offload (LRO) do not get disabled but need to be disabled for networking functions of guests to work correctly. If the host has an em(4) interface, this can be disabled by:

```
# ifconfig em0 -tso -lro
```

To avoid having to disable these after each reboot, add them to the system's /etc/rc.conf file:

```
# ifconfig_em0_ipv6="inet6 2403:5812:73e6:3::9:0 prefixlen 64 -tso -lro"
```

The above may not be required in every situation depending on the network card being used but if you experience guest network performance issues, this is what the problem will be.

To configure a vSwitch (bridge) the `switch` vm sub-command is used:

```
# vm switch create public
# vm switch add public em0
```

This creates a vSwitch called `public` and then attaches the em0 physical interface to the vSwitch.

Templates

Templates are required to assist with setting up guest configuration with the correct virtual hardware and other settings needed for them to function correctly. Using root, add the following to the templates repository:

```
# cat <<EOF > /vm/.templates/linux-uefi.conf
loader="uefi"
graphics="yes"
cpu=2
memory=1G
disk0_type="virtio-blk"
disk0_name="disk0.img"
disk0_dev="file"
graphics_listen="[::]"
graphics_res="1024x768"
```

```
xhci_mouse="yes"
utctime="yes"
virt_random="yes"
EOF

# cat <<EOF > /vm/.templates/windows-uefi.conf
loader="uefi"
graphics="yes"
cpu=2
memory=4G
disk0_type="nvme"
disk0_name="disk0.img"
disk0_dev="file"
graphics_listen="[::]"
graphics_res="1024x768"
xhci_mouse="yes"
utctime="no"
virt_random="yes"
EOF
```

Creating Guests

With storage, network, installers and templates prepared, guests can now be created.

```
# vm create -t windows-uefi -s 100G windows-guest
# vm add -d network -s public windows-guest

# vm create -t linux-uefi -s 100G linux-guest
# vm add -d network -s public linux-guest
```

The above creates both windows and linux guests with 100GB of storage allocated (using file backed storage) using their applicable templates and adds a network interface to each that is connected to the `public` vSwitch.

The windows guest needs a slight adjustment in its configuration to enable it to have network access during installation until the VirtIO drivers are installed. Edit the guests configuration and change the network interface from virtio-net to e1000:

```
# vm configure windows-guest
```

network0_type="e1000"
Revert to "virtio-net" once the RedHat VirtIO drivers have been installed.

Installing and Using Guests

To install each guest:

```
# vm install windows-guest Windows11-bhyve.iso
# vm install linux-guest AlmaLinux-9.5-x86_64-dvd.iso
```

Once the installation has been initiated, bhyve will be in a wait state. It will not commence the ISO boot process until a connection has been made to the VNC console port. To deter-

mine which guest console is running on a corresponding VNC port, use the **list** sub-command:

```
# vm list
NAME           DATASTORE  LOADER  CPU  MEMORY  VNC         AUTO  STATE
linux-guest    default    uefi    2    1G      [::]:5901   No    Locked (host)
windows-guest  default    uefi    2    8G      [::]:5900   No    Locked (host)
```

Using a VNC viewer like TigerVNC or TightVNC, connect the the IPv6 address and port for each of the guests to commence installation:

```
[2001:db8:1::a]:5900 # if the host is remote
[::1]:5900 # if it is on your local machine using localhost
```

After the Windows guest has been installed, the VirtIO drivers can be loaded. The drivers can be found at:
https://fedorapeople.org/groups/virt/virtio-win/direct-downloads/archive-virtio/virtio-win-0.1.266-1/virtio-win-gt-x64.msi
(sha256 – 37b9ee622cff30ad6e58dea42fd15b3acfc05fbb4158cf58d3c792b98dd61272)

Navigate to the above URL with the Edge Browser once Windows has finished installing to download and install these drivers. Once installed, shut down the host, switch the network interface in the guest configuration file back to virtio-net and the system can be started as normal.

We now have installed guests ready for use but there needs to be control over these so they can be started and stopped when required. The following commands will perform basic operations on your guests, such as starting, stopping or immediately powering off respectively:

```
# vm start linux-guest
# vm stop windows-guest
# vm poweroff windows-guest
```

The difference between stop and power off is that **stop** issues an ACPI shutdown request to the guest where **poweroff** immediately kills the bhyve process and won't shut down the guest cleanly.

Summary

This article is a brief insight into controlling bhyve and installing common operating systems with tools that are available directly from the FreeBSD package repository. vm-bhyve can do so much more than what was described here and is comprehensively detailed in the vm(8) man page.

JASON TUBNOR has over 28 years of IT industry experience in a vast range of disciplines and is currently the ICT Senior Security Lead at Latrobe Community Health Service (Victoria, Australia). Discovering Linux and Open Source in the mid 1990s, then being introduced to OpenBSD in 2000, Jason has used these tools to solve various problems in organizations that cover different industries. Jason is also a co-host on the BSDNow Podcast.

Xen and FreeBSD

BY ROGER PAU MONNÉ

The Xen Hypervisor began at the University of Cambridge Computer Laboratory in the late 1990s under the project name Xenoservers. At that time, Xenoservers aimed to provide "a new distributed computing paradigm, termed 'global public computing,' which would allow any user to run any code anywhere. Such platforms price computing resources, and ultimately charge users for resources consumed".

Using a hypervisor allows for sharing the hardware resources of a physical machine among several OSes in a secure way. The hypervisor is the piece of software that manages all those OSes (usually called guests or virtual machines) and provides separation and isolation between them. First released in 2003 as an open-source hypervisor under the GPLv2, Xen's design is OS agnostic, which makes it easy to add Xen support into new OSes. Since its first release more than 20 years ago, Xen has received broad support from a large community of individual developers and corporate contributors.

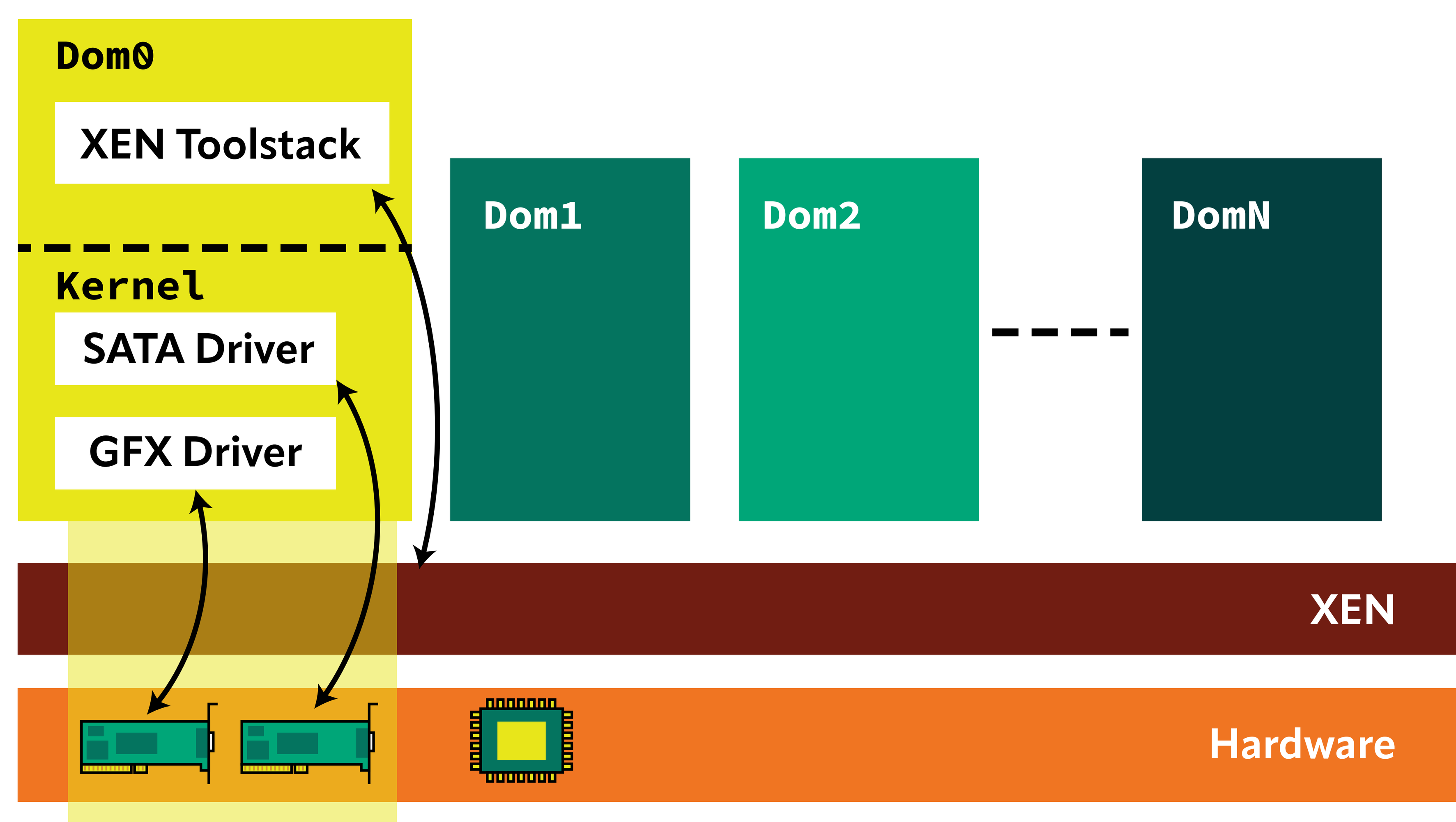
The Architecture

Hypervisors can be divided into two categories:

- **Type 1:** those that run directly on bare metal and are in direct control of the hardware.
- **Type 2:** hypervisors that are part of an operating system.

Common Type 1 hypervisors are VMware ESX/ESXi and Microsoft Hyper-V, while VMware Workstation and VirtualBox are clear examples of Type 2 hypervisors. Xen is a Type 1 hypervisor with a twist — its design resembles a microkernel in many ways. Xen itself only takes control of the CPUs, the local and IO APICs, the MMU, the IOMMU, and a timer. The rest is taken care of by the control domain (Dom0), a specialized guest granted elevated privileges by the hypervisor. This allows Dom0 to manage all other hardware in the system, as well as all other guests running on the hypervisor. It is also important to realize that Xen contains almost no hardware drivers, preventing code duplication with the drivers already present in OSes.

Architecture



When Xen was initially designed there were no hardware virtualization extensions on x86; options for virtualization either involved full software emulation or binary translation. Both options are very expensive in terms of performance, so Xen took a different approach. Instead of intending to emulate the current x86 interfaces, a new interface was provided to guests. The purpose of such a new interface was to avoid the overhead of having to deal with the emulation of hardware interfaces in the hypervisor and, instead, use a new interface between the guest and Xen that's more natural to implement for both. This virtualization approach is also known as Ring Deprivileging.

However, this requires the guest to be aware it's running under Xen, and to use a different set of interfaces compared to running natively. That set of interfaces was designated as ParaVirtualized, and hence the guests that used those interfaces were usually referred to as PV guests. The following interfaces are replaced with PV equivalents on PV guests:

- Disk and network.
- Interrupts and timers.
- Kernel entry point.
- Page tables.
- Privileged instructions.

The main limitation with such an approach is that it requires extensive changes to core parts of the guests kernel OSes. Currently, the only OSes that still have x86 Xen PV support are Linux and NetBSD. There was an initial port of Windows to run as a PV guest that was never published, plus Solaris also had PV support.

With the addition of hardware virtualization extensions to x86 CPUs, Xen also gained support to run unmodified (non-PV) guests. Such guests rely on the usage of hardware virtualization plus emulation of hardware devices. On a Xen system, such emulation is either done by the hypervisor itself (for performance critical devices) or offloaded to an external emulator running in user-space, by default QEMU. This hardware virtualized guests that emulates a full PC-compliant environment is called HVM in Xen terminology.

Instead of intending to emulate the current x86 interfaces, a new interface was provided to guests.

So now we have gone over two very different types of guests, on one side we have PV guests that use PV interfaces to avoid emulation, and on the other side, we have HVM guests that rely on hardware support and software emulation in order to run unmodified guests.

Emulated IO devices used by HVM guests, such as disks or network cards, don't perform very well due to the amount of logic required to handle data transfers and the overhead of emulating legacy interfaces. To avoid this penalty, Xen HVM guests also get the option to use PV interfaces for IO. Some other PV interfaces are available to HVM guests (like a one-shot PV timer) to reduce the possible overhead of using emulated devices.

While HVM allows every possible unmodified x86 guest to run, it also has a wide attack surface due to emulating all devices required for a PC compatible environment. To reduce the amount of interfaces (and thus the surface of attack) exposed to guests, a slightly modified version of HVM guests was created, named PVH. This is a slimmed down version of HVM, where a lot of emulated devices that would be present on HVM guests are not available. For example, a PVH guests only gets an emulated local APIC and maybe an emulat-

ed IO APIC, but there's no emulated HPET, PIT or legacy PIC (8259). PVH mode, however, might require modifications in the guest OS kernel so it's aware it's running under Xen and some devices are not available. PVH mode also uses a specific kernel entry point that allows direct booting into the guest kernel, without relying on an emulated firmware (SeaBIOS or OVMF), thus greatly speeding up the boot process. Note, however, OVMF can also be run in PVH mode to chain load OS-specific bootloaders when startup speed is not of great concern and ease of use is preferred. See the table below for a brief comparison of the different guest modes on x86.

	PV	PVH	HVM
I/O devices	PV (xenbus)	PV (xenbus)	emulated + PV
Legacy devices	NO	NO	YES
Privileged instructions	PV	hardware virtualized	hardware virtualized
System configuration	PV (xenbus)	ACPI + PV (xenbus)	ACPI + PV (xenbus)
Kernel entry point	PV	PV + native*	native

* It's possible for PVH guests to re-use the native entry point when booted with firmware, but that requires adding logic to the native entry point to detect when booting in a PVH environment. Not all OSes support this.

The PVH approach has also been adopted by other virtualization technologies like Firecracker from AWS. While Firecracker is based on KVM, it re-uses the Xen PVH entry point and applies the same attack surface reduction by not exposing (and thus emulating) legacy x86 devices.

Speaking about ARM architecture, the fact that the Xen port was developed once ARM already had support for hardware virtualization extensions led to a different approach when compared to x86. ARM has only one guest type, and it would be the equivalent of PVH on x86. The focus is also to attempt to not expose an excess of emulated devices to reduce the complexity and the attack surface.

It's quite likely that the upcoming RISC-V and PowerPC ports will take the same approach of supporting only one guest type, more akin to HVM or PVH on x86. Those platforms also have hardware virtualization extensions that forego the need for something like classic PV support.

Usage and Unique Features

The first commercial uses of Xen were strictly focused on server virtualization, either on premise usage of Xen-based products or through cloud offerings. However, due to its versatility, Xen has now also extended into the client and embedded space. Xen's small footprint and security focus makes it suitable for a wide range of environments.

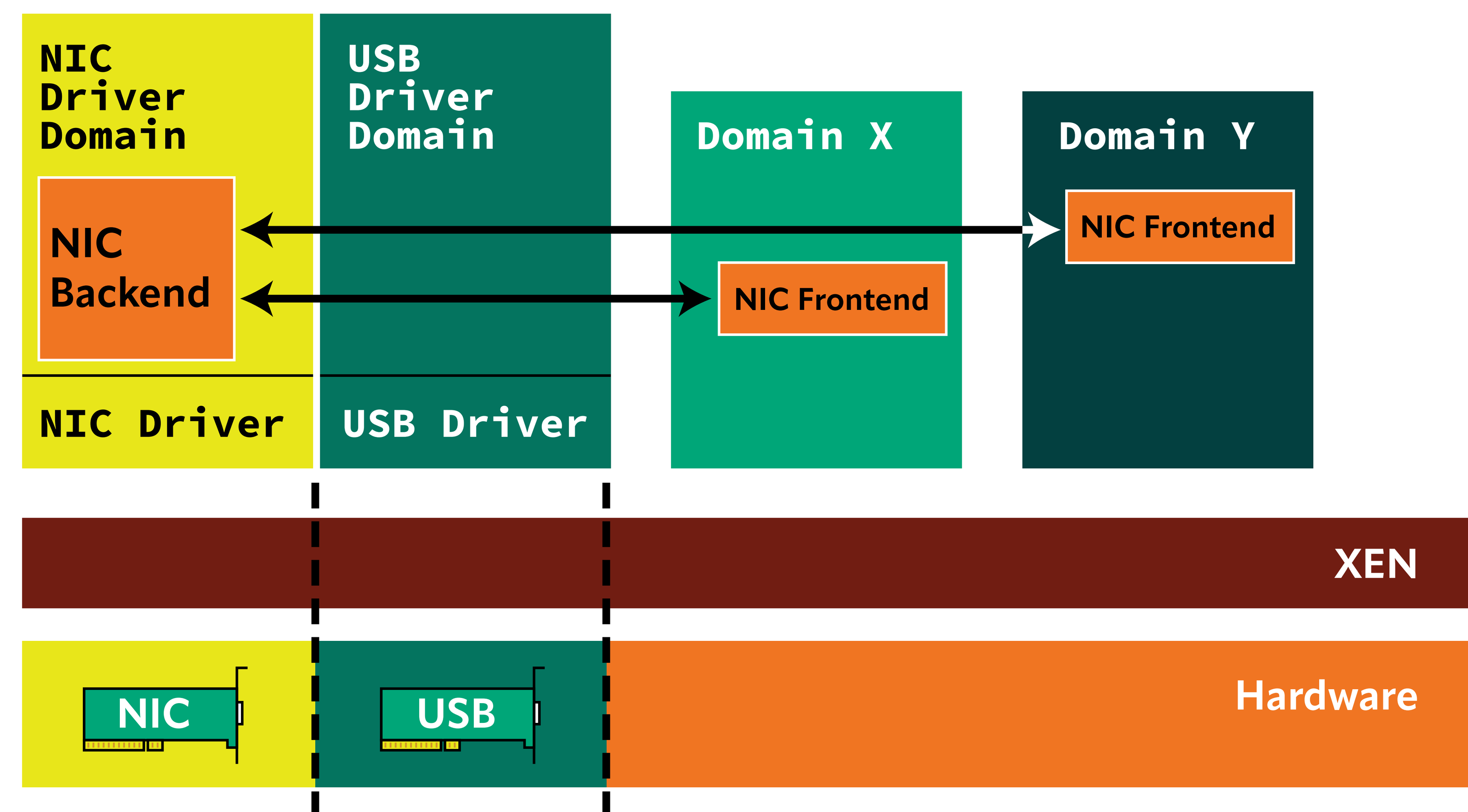
A great example of a client (desktop) usage of Xen is QubesOS, a Linux-based OS that's focused on security through isolation of different processes in virtual machines, all running on top of the Xen hypervisor and even supporting the usage of Windows applications. QubesOS relies heavily in some key Xen-specific features:

- Driver domains: network cards and USB drivers are run in separate VMs, so that security issues from the usage of those devices cannot compromise the entire system. See the diagram about driver domains.
- Stub domains: the QEMU instance that handles the emulation for each HVM guests is not run in dom0, but rather in a separate PV or PVH domain. This isolation prevents security issues in QEMU from compromising the entire system.

- Limited memory sharing: by using the grant sharing interfaces, a domain can decide what pages of memory are shared to which domains, thus preventing other domains (even semi-privileged ones) from being able to access all guest memory.

Similarly to QubesOS there's also OpenXT: a Xen-based Linux distribution focused on client security used by governments.

Driver Domains



A couple more of unique Xen x86 features that are used by diverse products:

- Introspection: allows external monitors (usually running in user-space on a different VM) to request notifications for actions performed by a guest. Such monitoring includes, for example, access to a certain register, MSR, or changes in execution privilege level. A practical application of this technology is DRAKVUF, a malware analysis tool that doesn't require any monitor to be installed in the guest OS.
- VM-fork: much like process forking, Xen allows forking of running VMs. Such a feature still doesn't create a fully functional fork, but it's enough to be used for kernel fuzzing. The KF/x fuzzing project puts the kernel into a very specific state, and then starts fuzzing by creating forks of the guest. All forks start execution at the same instruction, but with different inputs. Being able to fork a VM in a very specific state extremely fast and in parallel is key to achieving a very high rate of iterations per minute.

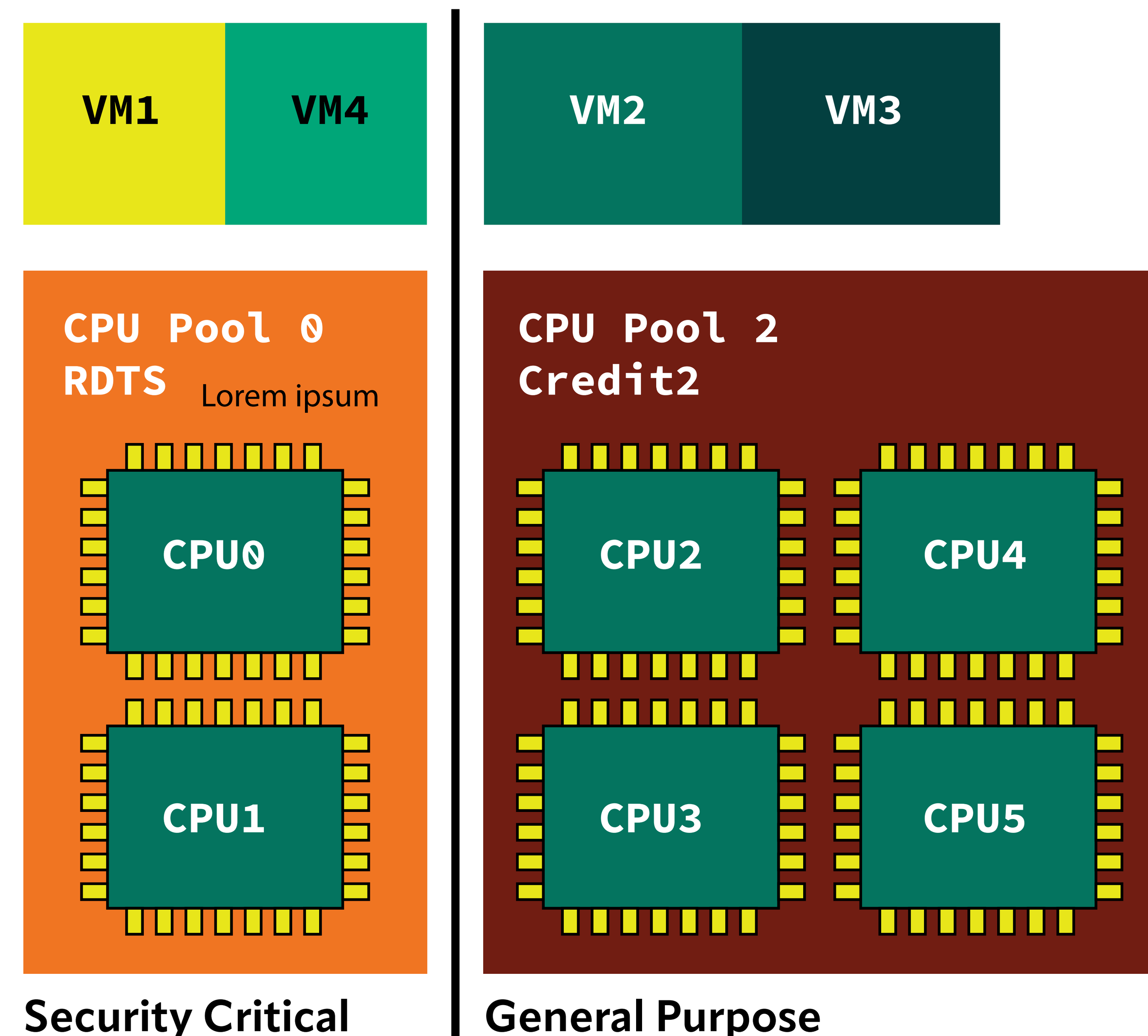
Since the addition of the ARM port, there's been a wide interest in using Xen on embedded deployments, from industrial to automotive. Apart from the small footprint and security focus, there are some key features of Xen that make it appealing for such usages. First, the amount of code in Xen is quite limited when compared to Type-2 hypervisors, so it's conceivable to attempt to safety-certify it. There's currently an effort upstream to attempt to comply with the applicable parts of the MISRA C standard so Xen can be safety certified.

Some unique features that make it very appealing to embedded uses include:

- Small code base: makes it possible to audit and safety certify, also the code base is being adapted to comply with the MISRA C standard.
- cpupools: Xen has the ability to partition the CPUs into different groups and assign a different scheduler to each group. Guests can then be assigned to those groups, allowing for a set of guests that run using a real-time scheduler, like RTDS or ARINC653, while a different set of guests can run using a general-purpose scheduler like credit2. See CPU Pools diagram.
- CPU pinning: it's also possible to apply restrictions on which host CPUs get to schedule which guest CPUs, so, for example, a guest CPU can be exclusively given a host CPU when running latency sensitive workloads.

- Deterministic interrupt latency: significant efforts have been put into Xen to ensure interrupt latency remains both low and deterministic, even in the presence of cache pressure caused by noisy neighbors. There's a patch series currently in review that adds cache coloring support to Xen. Additionally, Xen is being ported to run on Arm-v8R MPU (memory protection unit) based systems. This is a quite significant change in Xen's architecture, as it has always been supported on Memory Management Unit (MMU) based systems. With MPU, there is flat mapping between VA and PA and thus one can achieve real-time effect since there is no translation involved. There are a limited number of memory protection regions that can be created by Xen to enforce memory type and access restrictions on different memory ranges.
- dom0less/hyperlaunch: a feature that originated in ARM and is currently also being implemented for x86 allows multiple guests to be created statically at boot time. This is very useful for static partitioned systems, where the number of guests is fixed and known ahead of time. In such a setup the presence of an initial (privileged) domain is optional, as some setups don't require further operations against the initially created guests.

CPU Pools



FreeBSD Xen Support

FreeBSD Xen support was added quite late compared to other OSes. For instance, NetBSD was the first OS to formally commit Xen PV support because Linux patches for full PV support didn't get merged until Linux 3.0 (around 2012).

FreeBSD had some initial support for PV, but that port was 32bit only and not fully functional. Development on it stopped, and it ended up being deleted from the tree once PVH support was implemented. In early 2010, FreeBSD saw the addition of PV optimizations when running as an HVM guest, which allowed FreeBSD to make use of PV devices for I/O together with the usage of some additional PV interfaces for speedups like the PV timer.

In early 2014, FreeBSD gained support to run as a PVHv1 guest, and shortly after, as a PVHv1 initial domain. Sadly, the first implementation of PVH (also known as PVHv1) was wrongly designed, and had backed in too many PV-related limitations. PVHv1 was designed as an attempt to move a classic PV guest to run inside of an Intel VMX container. This was fairly limiting, as the guest still had a bunch of restrictions inherited from classic PV, and it was also limited to Intel hardware only.

After finding out about those design limitations, work started on moving to a different implementation of PVH. The new approach started with an HVM guest and stripped as

much emulation as possible, including all emulation done by QEMU. Most of this work was, in fact, developed with FreeBSD, as that's my main development platform, and I did extensive work in order to implement what was later called PVHv2 and is now plain PVH.

FreeBSD x86 runs as both an HVM and PVH guest and supports running as a PVH dom0 (initial domain). In fact, x86 PVH support was merged earlier in FreeBSD than Linux. Running in PVH mode, however, still has some missing features compared to a classic PV dom0. The biggest one is the lack of PCI passthrough support, which, however, requires changes in both FreeBSD and Xen to be implemented. There's an ongoing effort in Xen upstream to add PCI passthrough support for PVH dom0, however, that's still being worked on, and when finished, will require changes to FreeBSD for the feature to be usable.

On the ARM side, work is underway to get FreeBSD to run as an Aarch64 Xen guest. That required splitting the Xen code in FreeBSD to separate the architecture specific bits from the generic ones. Further work is being done to integrate Xen interrupt multiplexing with the native interrupt handling done in ARM.

Recent Developments in the Xen Community

Apart from the ongoing effort mentioned before that attempts to bring feature parity between a PV and PVH dom0 on x86, there's a lot more going on in upstream Xen. Since the last Xen release (4.19), PVH dom0 has been a supported mode of operation, albeit with caveats due to some key features still missing.

The RISC-V and PowerPC ports are making progress to reach a functional state, hopefully in a couple of releases we might have them reach a state where the initial domain can be booted and guests can be created.

There's a lot more going on in upstream Xen.

At least on x86, a lot of time in recent years has been spent on mitigating the flurry of hardware security vulnerabilities. Since the original Meltdown and Spectre attacks released in early 2018, the amount of hardware vulnerabilities has been increasing steadily. This requires a lot of work and attention on the Xen side. The hypervisor itself needs to be fixed so as not to be vulnerable, but it's also quite likely some new controls need exposure to the guests so they can protect themselves. To mitigate the impact that future hardware vulnerabilities have on Xen, we are working on a new feature called Address Space Isolation (which has also been known as Secret Free Xen), that aims to remove the direct map plus all sensitive mappings from being permanently mapped in the hypervisor address space. This would make Xen not vulnerable to speculative execution attacks, thus allowing the removal of a lot of the mitigations applied on entry points into the hypervisor, and possibly the need to apply more mitigations for any future speculative issues.

Since the beginning of 2021, all Xen commits have been tested for builds on FreeBSD using the Cirrus CI testing system. This has been a massive help to keep Xen building on FreeBSD, as the usage of Clang plus the LLVM toolchain sometimes created or displayed issues that wouldn't manifest when using the GNU toolchain. We currently test that Xen builds on all the supported FreeBSD stable branches, plus the HEAD development branch. Xen recently retired its custom testing system called osstest, and now solely relies on Gitlab CI, Cirrus CI and Github actions to perform testing. This allows for a more open and well documented testing infrastructure, where it's easier for newcomers to contribute and add

tests. Future work in that area should include runtime testing on FreeBSD, even if initially using QEMU instead of a real hardware platform.

Recent releases also added toolstack support for exposing VirtIO devices to Xen guests. Both Linux and QEMU currently support using VirtIO devices with grants instead of guest memory addresses as the basis for memory sharing between the VirtIO frontends and backends. This addition hasn't required a VirtIO protocol change, since it's, instead, implemented as a new transport layer. There are also efforts to introduce a transport layer not based on memory sharing, as this is a requirement for some security environments. Going forward, this would allow Xen to use VirtIO devices while keeping the security and isolation that's guaranteed when using the native Xen PV IO devices. The overall goal is to be able to reuse the VirtIO drivers as first-class interfaces on Xen deployments.

Safety certification and the adoption of MISRA C rules has also been one of the main tasks for the past releases. The last Xen release (4.19) has been extended to support 7 directives and 113 rules of a total of 18 directives and 182 rules that conform to the MISRA C specification. Adoption is being done progressively, so that each rule or directive can be debated and agreed upon before being adopted. Given that the Xen code base wasn't designed with MISRA compliance in mind, some of the rules will require either global or local per-instance deviations. Also, as part of the Safety Certification initiative work, it has started adding safety requirements and assumptions of use. Safety requirements provide a detailed description of all the expected behaviors of the software (Xen), enabling independent testing and validation of these behaviors.

The Future of Xen

Looking back at when x86 PVH support was first added on FreeBSD, it's been a long and not always easy road. FreeBSD was an early adopter of PVH for dom0 mode, and a lot of Xen development has been done while using a FreeBSD PVH dom0. It's also notable how FreeBSD has become a first-class Xen citizen in the recent years, as now there is build testing of Xen on FreeBSD for each commit that goes into the Xen repository.

The port of FreeBSD to run as a Xen Aarch64 guest has also gained some traction recently and is certainly a feature to look forward to given the increasing presence of ARM based platforms both on the server, the client, and the embedded environments.

It's good to see Xen being used in so many different use-cases, and so different from its inception design purpose of being focused on server side (cloud) virtualization. I can only hope to see which new deployments and use-cases of Xen will be used in the future.

How to Reach

The Xen community does all the code review on the xen-devel mailing list. For more informal communications and discussions we also run a couple of Matrix rooms free for everyone to access. For FreeBSD/Xen specific questions there's also the freebsd-xen mailing list, and of course the FreeBSD Bugzilla can be used to report any bugs against FreeBSD/Xen.

ROGER PAU MONNÉ is a Software Engineer at Cloud Software Group and a FreeBSD committer. His roles in the Xen community include being a x86 maintainer, part of the Xen Security Team and also a Xen committer. He has done extensive work on the x86 PVH implementation in both Xen and FreeBSD, and now spends most of his time working on security-related features or chasing down bugs.

Wifibox: An Embedded Virtualized Wireless Router

BY GÁBOR PÁLI

Due to changes in life priorities, I drifted away from FreeBSD for a few years around 2017. Later I returned and started building a new FreeBSD-based workstation for myself, a Lenovo ThinkPad X220. I noticed that although it was working, the wireless support was still far from optimal, the `iwm` driver was neither stable nor peformant enough for daily use.

I realized that in the wireless networking area, FreeBSD is still struggling to match the performance of Linux-based systems due to lack of up-to-date hardware support. This is happening for a reason: FreeBSD is often not considered a first-class citizen, hence it is not a target of such developments, and the respective portions of its networking subsystem need to be elevated to meet the latest requirements. This is not a trivial issue to fix, and the FreeBSD Foundation has been sponsoring a long-term project that aims to bring updates to the stack and establish a framework to facilitate re-using the wireless network card drivers from Linux.

> In the wireless networking area, FreeBSD is still struggling to match the performance of Linux-based systems.

This began to bug me and I could not just wait patiently for the problem to get resolved. I wanted to be part of the development of FreeBSD again because I not only enjoyed using it but also learning about it. However, I did not have the luxury of investing time into mastering both the networking and driver code development in my free time, so I had to look for other opportunities.

Prototype

I was excited when another approach was brought to my attention (thank you to Gábor Zahemszky for that!). It was the idea of leveraging the PCI pass-through capabilities of **`bhyve`** to run a Linux guest, for which it becomes possible to talk to the real hardware, set up the wireless connection, and share the network with the FreeBSD host. I discovered David Schlachter's excellent blog post where the whole process is described in detail, and was able to build my own prototype with the help of that.

While experimenting with the whole process, I studied it from at least two perspectives. First, whether this is something that would be sustainable in the long run with regard to system upgrades, and second, whether this is something that users without a deep understanding of the solution could easily install and remove. As a former ports developer, I knew that I would have to be able to maintain the components and keep them isolated from the base system and somehow integrate with other third-parties. So why not exploit the existing ports framework for that purpose? And then the concept of the **`net/wifibox`** port was conceived in April 2021.

Initially, I used the **sysutils/vm-bhyve** port to build and manage a virtual machine that was based on Alpine Linux. Alpine is a lightweight Linux distribution that adopts OpenRC as the init system, uses the **musl** standard C library to make it possible to create small applications, and integrates BusyBox for the most commonly utilized command-line tools. Originally, it was created as an embedded-first distribution. I learned about it when I was working with Docker container images and remembered it for the small footprint and ease of use. It is actively maintained and provides a large number of packages which are managed through its "aports" system. In retrospect, the whole system bears resemblance to FreeBSD in many aspects, and I grew fond of it.

Although **vm-bhyve** is an excellent tool, I felt that it was too much for this specific use case. Instead, I used it as a model for the basic user interface, such as providing a console for the user to interact with the virtual machine hosted inside, and the elementary orchestration routines. Since these routines required interaction with the command-line **bhyve** tools, I decided to stick with the shell-based approach. I would probably not have had a better experience if I tried to implement all the plumbing in some other, higher-level language such as Python, as it would unnecessarily increase the build times and introduce a dependency on other third-party packages.

In the end, the user interface of Wifibox was composed of the **start**, **stop**, **restart**, **status**, and **resume** commands. The resume operation had to be handled specially because it was known that on suspending the notebook, the virtual machine loses its connection to the virtualized PCI device which has to be recovered somehow. After some experimentation, I noticed that this could be mitigated by stopping the virtual machine, reloading the **vmm** kernel module, and starting the VM again. There was recently a solution proposed by Joshua Rogers to fix the issue on the kernel level, but this has not yet been added to the base system.

Alpine is a lightweight Linux distribution that adopts OpenRC as the init system.

The VM's interaction with the PCI device has often proven to be a weak point, which often limits the usability of Wifibox itself. As an enhancement to this workaround, the repertoire of recovery methods have been expanded. Certain hardware configurations react differently to how the device is shut down and restarted. For example, thanks to Joshua's work, it was discovered that it matters if the guest itself shuts down the device properly during its own shutdown sequence. It was also learned that the **ath11k_pci** Linux kernel module does not well tolerate run in a virtualized environment, because it assumes that the location of the Message Signaled Interrupts (MSI) table matches with that of the host. This could only be handled if the FreeBSD host somehow supported injecting the host physical MSI information for the guest or disabling the MSI virtualization.

VirtFS/9P Support

One of the primary design principles of Wifibox was that users should not know about the underlying virtual machine, but be able to run it directly on the host as a local application. To create the illusion of that, the recently completed work around the VirtFS/9P file system pass-through support of **bhyve** was explored. By mounting the appropriate directo-

ries on the host for the **bhyve** guest, the required configuration files could be imported and the log files could be exported. This way, the user would not have to move or keep files in sync manually between the virtual machine and the FreeBSD host.

The VirtFS/9P support was made available beginning with FreeBSD 13, but I wanted to extend it to the older versions at that time, 12 and 11, to broaden its userbase. Fortunately, this feature is contained in a single module, and I was able to create another port, called **sysutils/bhyve+**, to automatically patch the **bhyve** sources in the base system to have this module included. With the help of that, there was no need to wait for the original authors to backport the feature, but this extra dependency could be pulled in for the **net/wifibox** port when needed. Besides the addition of **virtfs-9p**, the **bhyve+** port included many other fixes that made it possible for Wifibox to run. Basically the goal was to put together a version of **bhyve** that could be the same for every major FreeBSD version and minimize the differences. This would have been based on the version in the 13.x line, but the related architectural changes made it non-trivial and the idea was later dropped. Over the years, its relevance has slowly faded away and it eventually became obsolete.

The creation of the disk image for the guest had to face many challenges. The primary concern was that in the beginning, the image itself was a pre-installed Alpine system. It was maintained locally on my workstation, it was hard to track what it contained, and it kept changing due to writes to the various temporary and work files. From the user's perspective, it raised the valid question of trusting "somebody else's VM." The initial versions were around 640 MBs, which looked gigantic compared to the ones that are typical for embedded systems. This size partially resulted from the image containing all possibly useful tools and files, so it was a logical next step make it smaller and more modular.

Version 1.0

For version 1.0 in May 2022, a lot of effort went into reworking how the image was created. The prime directive was to make the whole process reproducible and lean. Technically, the complexity of installing the system components from scratch was translated to the port's **Makefile**. The image has gained its own sub-port, **net/wifibox-alpine**, while the orchestrator script was split into **net/wifibox-core**, and **net/wifibox** has become a metaport. Through constant experimentation with the Alpine installation files, the root file system package, and its package manager, the **apk** tool, they were adapted to run atop FreeBSD with the help of the Linuxulator. In addition to the creation of a **Makefile** to drive the automated installation of the system to a designated directory on the host optionally extended with extra files, Alpine packages are downloaded and installed there. The packages themselves offered a way to modularize the construction of the image and make it possible for the user to select between them through the various port options. For example, the firmware files for each of the major wireless card brands could be separately installed and FreeBSD package flavors could be created for them.

The creation of the disk image for the guest had to face many challenges.

The package-based approach lent itself to the creation of additional packages and the modification of the existing ones. Unfortunately, many of the upstream Alpine packages turned out to carry some extra weight, such as documentation or additional binaries, so

they had to be removed. But it also allowed porting applications such as `mDNSResponder` to this platform and allowed them to run as part of the solution. The package for the Linux kernel itself had to be heavily edited to lose all the unused components, reduce its resource consumption, and shrink its attack surface. Wifibox does not need the standard `initrd`-based boot process, therefore the initial temporary root file system is completely removed and the boot happens directly with its root file system. Configuration files and patches for architectures other than AMD64 were removed, as Wifibox only supports that specific one.

The virtual machine image is compressed by SquashFS and keeps the overall size down to the ballpark of 15 MB. This approach also comes with a read-only root file system that prevents even the `root` user from tampering. It is expanded with a memory-backed temporary file system that is mounted under `/tmp` to manage the run-time file writes besides the VirtFS/9P mounts for the reading the application configuration files from the host. The boot process is run through GRUB, hence the Linux kernel (without its modules) is not part of this image, but pre-loaded with `sysutils/grub2-bhyve`.

Package Framework

To roll out the required set of Wifibox packages in addition to the ones imported from upstream, the package framework of Alpine Linux has been adopted. For transparency and reproducibility, every customized package has its own `APKBUILD` file and extra files version-controlled in `git`. The packages are built on a clean, dedicated Alpine Linux `bhyve` virtual machine, often dubbed `wifibox-dev`, which is re-created for every minor Alpine release. The resulting packages are uploaded to GitHub for the user's convenience. In the past, there were experiments to build the packages in a Linux `chroot` environment, but FreeBSD's native Linux emulation support did not prove sufficient enough for this purpose. The results were similar in case of cross compilation, which is why I ended up with using `bhyve` for this as well. Per Bernhard Fröhlich's suggestion, I am currently looking into utilizing GitHub Actions to build the Wifibox packages automatically and independently in a native Linux environment.

The resulting packages are uploaded to GitHub for the user's convenience..

Linux-based Wireless Stack

A regular Linux-based wireless stack is operated inside the VM. First of all, the Linux kernel is used to detect the PCI wireless device and make it run through one of its drivers and the corresponding firmware, when necessary. The `wlan0` wireless networking interface is brought up by the standard OpenRC services. Then either WPA Supplicant or `hostapd` is hooked up on that to finalize the configuration. Next to the wireless interface, a virtual `eth0` Ethernet interface is exposed by `bhyve`. On the host, there is a bridge interface, `wifibox0` defined, which is joined with `eth0` in the guest through a `tap` software tunnel. Using `iptables`, there is Network Address Translation (NAT) and packet forwarding applied to make the traffic flow bidirectionally between `wlan0` and `eth0`. The IP addresses are obtained with the help of Busybox's built-in `udhcpd` for the host (over `eth0`) and either `dhcpcd` or `udhcpd`

in the guest (over `wlan0`). The IP address range for the host can be controlled in the Wifibox configuration and adjusted according to the user's needs.

Due to the introduction of NAT, note that Wifibox uses a different range of IP address for **`eth0`** and `wlan0`. As a consequence, certain applications may not work properly out of the box, and the deployment of additional tools and extra configuration, e.g., port forwarding is required. This can be considered a benefit from the perspective of security, since there we have a firewall installed automatically. But this is equally a drawback, because it breaks the end-to-end connectivity, a core principle of the Internet. To overcome this, there were experiments to push down the packet forwarding to the level of Ethernet. For example, there is WLAN Kabel that implements moving packets between Ethernet and wireless interfaces. It was used with moderate success, since the traffic was flowing, but DHCP communication could not be made to pass through the barrier, and the observed performance was low. Nevertheless, that is a curious approach which is worth exploring further in the future.

An advantage of using Wifibox is the so-called "Unix domain socket pass-through", which helps tools like `wpa_cli` or `wpa_gui` communicate directly with the WPA Supplicant running inside the guest so that the user does not have to run them there and interact with the VM. That is extra functionality because Unix domain sockets on the guest file system are not exported by VirtFS/9P so the host will not see them. This is overcome by running a dedicated **`uds_passthru`** process in the guest, which runs a stripped-down version of **`socat`** in the background to convert the sockets to TCP ports. It has a pair on the side of the host, that communicates on those ports and translates the data back to a socket locally. With that in place, Wifibox can emulate the presence of that socket to implement smooth communication. In collaboration with the guest, the orchestrator script automatically manages the socket pass-through through a specific configuration file.

Wifibox as a Product

Although the Wifibox guest OS is mostly based on Alpine Linux, it often includes additional patches. For example, it imports a number of patches and packages from Arch Linux, because Arch has a better support for wireless devices. But it was also discovered that the driver for older Broadcom cards simply did not have support for MSI, which is a must for

drivers that want to run in a virtualized environment. Slow initialization lead time of certain drivers can make the WPA Supplicant unable to find the **wlan0** interface on boot, so an exponential back off mechanism was implemented to enhance its resilience. As experience shows, a small operating system distribution dedicated to solving these issues is definitely warranted.

The orchestrator script makes it possible to combine **bhyve** with other tools to shape its behavior further. Thanks to Anton Saietskii's observations, **nice** was connected to assist with controlling the priority of the process that is responsible for running the virtual machine and avoid overloading the host. Similarly, a layer with **daemon** was added to monitor the status and revive the machine if it crashed or it was deliberately restarted.

As of the time of writing, Wifibox is actively maintained and it has been receiving semi-annual updates in March and September. With the help of Ashish Shukla, these releases are published to the FreeBSD Ports Collection, therefore they are available for installing with the **pkg** tool. Note that Wifibox is not featured on the installation media for the FreeBSD releases, which can make it harder to take advantage of it when one tries to install FreeBSD over a wireless network. However, it is possible to add all the required binary packages to the installation media and use them to initialize the network connection before starting the installation procedure.

At the project's home page, both the source code and pointers to the respective GitHub repositories can be found, tickets can be opened, and discussions can be started. There is a separate repository for the ports themselves where the published development versions present an opportunity to take a peek into what is brewing next and test out fixes for issues.

Wifibox is actively maintained and it has been receiving semi-annual updates.

Documentation

Wifibox is bundled with a lot of documentation, so I encourage the reader to explore it further. And I would like to emphasize an implicit but important organizing principle with regards to documentation. Wifibox follows a phased, "read as you go" model. This means that it has no extensive online documentation, the README file in the main GitHub repository covers an introduction, the basic installation instructions, and the list of hardware configurations that are known to be compatible. The user then has to install Wifibox to get access to the manual page, which has further details on how to use the tool and where the configuration files are. And then in the configuration files, additional instructions are provided on how to work ourselves through the related steps, together having a validation in place with helpful error messages to guide the user. The virtual machine image has its own dedicated manual page. Thanks to John Grafton, Warner Losh, and many other users for giving me feedback on how to improve on these.

Summary

This all is an interesting mix of product of curiosity, the desire to provide a quick remedy for the challenges in FreeBSD's wireless journey, and inventing another use of the technical advantages that are provided by PCI pass-through in virtualization and the design of **bhyve**.

Wifibox still has its own drawbacks, and it is nowhere near a drop-in replacement for the native solution. Hence, it is a called an embedded virtualized wireless router which removes the need to buy dedicated hardware and creatively presents the CPU's existing virtualization capabilities as an imitation of that. However, I believe this approach still has ideas to chase, such as using `bhyve` to run the Linux kernel and its drivers only in the virtual machine and expose that directly as a wireless networking device. This would bring the approach one step closer to the native solution, but it is not yet known if it is feasible and, yes, how much work it would require. In the meantime, I hope that Wifibox can alleviate the pressure around making the native solution production-ready and reassure users that FreeBSD is still a great choice nowadays and they do not have to necessarily give up on getting good speeds and reliable connections over wireless connections.

GÁBOR PÁLI has been a happy and committed FreeBSD user for many decades and he also has had the joy of being a documentation and ports developer. He lives on the edge of the beautiful Hungarian town of Esztergom with his wife. He endorses the adoption of functional programming in the industry and nowadays he contributes to Apache CouchDB where he can write Erlang and Scala code.

The FreeBSD Project is looking for

- Programmers
- Testers
- Researchers
- Tech writers
- Anyone who wants to get involved

Find out more by

Checking out our website
freebsd.org/projects/newbies.html

Downloading the Software
freebsd.org/where.html

We're a welcoming community looking for people like you to help continue developing this robust operating system. Join us!

Already involved?

Don't forget to check out the latest grant opportunities at **freebsdfoundation.org**

Help Create the Future. Join the FreeBSD Project!

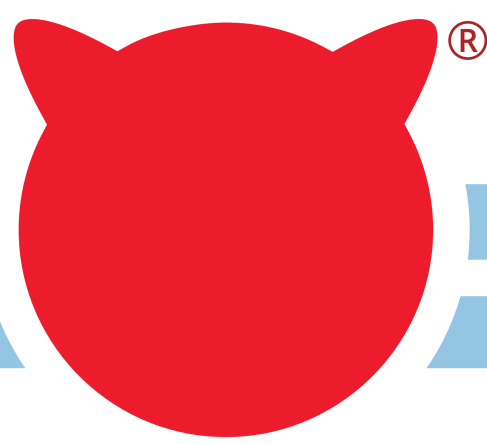
FreeBSD is internationally recognized as an innovative leader in providing a high-performance, secure, and stable operating system.

Not only is FreeBSD easy to install, but it runs a huge number of applications, offers powerful solutions, and cutting edge features. The best part? It's FREE of charge and comes with full source code.

Did you know that working with a mature, open source project is an excellent way to gain new skills, network with other professionals, and differentiate yourself in a competitive job market? Don't miss this opportunity to work with a diverse and committed community bringing about a better world powered by FreeBSD.

The FreeBSD Community is proudly supported by

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.

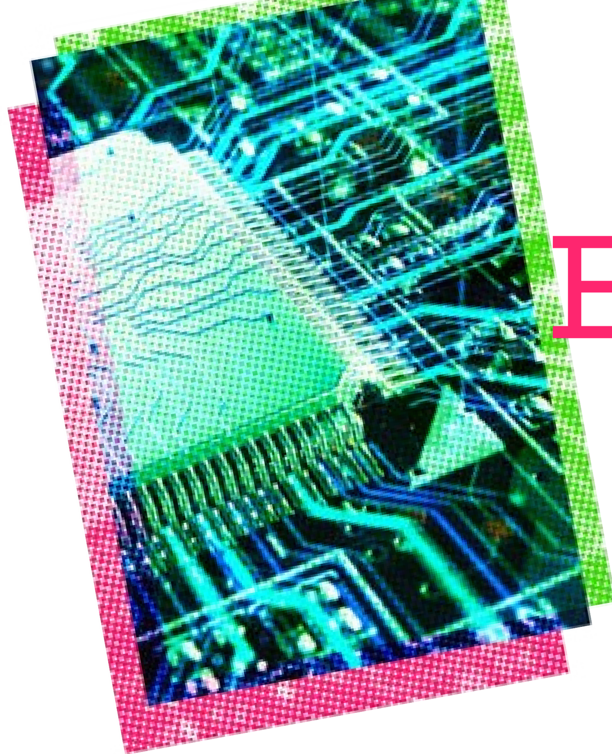
Please consider making a gift to support FreeBSD for the coming year. It's only with your help that we can continue and increase our support to make FreeBSD the high-performance, secure, and reliable OS you know and love!

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



Embedded FreeBSD

Fabric – Baby Steps

BY CHRISTOPHER R. BOWMAN

In previous columns, we took a basic look at the Zynq chip and mentioned it's fabric. Since then, we haven't really mentioned it much. But, in the last column, we got bhyve running a CentOS image, and so now it's time to look at our first fabric circuit. This column will be all circuit and no FreeBSD, but with the next couple columns, we will start to look at systems that combine the two.

In addition to the Zynq-7000 SoC Technical Reference Manual which documents the Zynq chip that forms the bulk of the Arty Z7-20 functionality, Digilent's Arty Z7 Reference Manual contains a wealth of valuable information on how the Zynq chip is wired and connected on the board. If we look at section 12, we can see that the board has 4 LEDs that attach directly to the ZYNQ chip (R14, P14, N16, M14). If we can set these pins to a logic 1 or high-voltage level, then the pins will source current that will flow through the LEDs and then through the current-limiting resistor and to ground causing the LEDs to turn on.

Since we're just getting started, let's try to build the smallest and simplest circuit to turn on these pins. The simplest circuit would be to statically tie these pins high in the fabric. Great, how do we do this? We're going to have to introduce Verilog to do this.

> Over the last decade, two languages have really come to dominate chip design: VHDL and Verilog.

Much like in the early days of programming, where programs were written in machine code and then assembler and eventually in high-level languages like C, ICs were originally hand drawn or laid out with mylar tape. As circuits became larger and CAD programs were developed, circuits were designed using programs — EDA (Electronic Design Automation). Schematic capture programs allowed one to design circuits by connecting devices — originally transistors, and later gates — graphically using a GUI. Eventually, languages were created which allowed the description of circuits textually. I used an early language called SFL for two of my first 3 chips back in the 1.2-micron days. But over the last decade, two languages have really come to dominate chip design: VHDL and Verilog. These languages are much like Ada and C in many conceptual ways. Ada and VHDL are very verbose and have strong type checking. They were both designated as the preferred language for US DOD work, and while both are still present, they're about as popular in their respective fields today. Verilog on the other hand, like C, has become the most popular language in circuit design. It's not as strongly typed as VHDL is, just as C isn't

as strongly typed as Ada. In any case, these days, if you want to do digital (not analog) circuit design of anything more than a handful of gates, you're using VHDL or Verilog, with most new designs use Verilog.

Circuit designs in Verilog get turned into circuits using a synthesis tool just as we use compilers to turn C into a running program. There is a bit more to it. For instance, in addition to turning the Verilog into a set of connected gates implementing the design, when designing a computer chip, you also need to run a tool to place the gates and route the interconnect wires. Fortunately for us, all that functionality is present in the Xilinx/AMD tool called Vivado. Not only does Xilinx/AMD offer Vivado, but it's a free download, and you can get a license to unlock quite a bit of the functionality of the Zynq chips free of charge. If you want to do anything more than write programs for your Arty Z7 board, you'll need to download and install this software on a Linux or windows machine. We covered setting up a bhyve instance running Linux in the last column so that you can run Vivado on a Linux VM under bhyve on your FreeBSD machine.

Circuit designs in Verilog get turned into circuits using a synthesis tool just as we use compilers to turn C into a running program.

Now that we're oriented, let's turn towards our first and simplest circuit to turn on the LEDs on our board. First, we will need a Verilog description of our circuit. I can't possibly teach you Verilog in one small column, so I will just present the design and try to describe what it's doing.

```
module top(
    output [3:0]led
);

wire [3:0]led;

assign led = {1'b1, 1'b0, 1'b1, 1'b0};

endmodule
```

First, designs are captured in modules and this one has a 4-bit bus of wires coming out of it called `led`. We are driving this bus with a set of 4 concatenated constants. Those constants are 1-bit signals, half of which are logic "hi" or 1, and half are logic "low" or 0. I picked an alternating set of values so that I can tell which way the LEDs are wired: msb to lsb or lsb to msb. This way, I don't have to work it out from the markings on the board and the documentation.

The next thing we need to do is provide a constraints file. Constraints files have two major functions: they convey timing information, and in the FPGA world, they also convey some placement information. For our first experiment, we need to tell the tool what pins on the chip are connected to the wires of the `led` bus, and we also need to tell the tool how to setup the IO pins we wish to use. Digilent, quite helpfully, has a master XDC file that has this information for this board and many others in a github repo. Unfortunately, they don't include a copyright notice in the file or readme and so I can't include it in my project. The few

relevant lines are provided here, and if you use the make file included in my static_leds repo for this article, it will download the file from github and uncomment the relevant lines. You'll need GNU make and wget installed on your system.

```
set_property -dict { PACKAGE_PIN R14  IOSTANDARD LVCMOS33 } \
     [get_ports { led[0] }]; #IO_L6N_T0_VREF_34 Sch=LED0
set_property -dict { PACKAGE_PIN P14  IOSTANDARD LVCMOS33 } \
     [get_ports { led[1] }]; #IO_L6P_T0_34 Sch=LED1
set_property -dict { PACKAGE_PIN N16  IOSTANDARD LVCMOS33 } \
     [get_ports { led[2] }]; #IO_L21N_T3_DQS_AD14N_35 Sch=LED2
set_property -dict { PACKAGE_PIN M14  IOSTANDARD LVCMOS33 } \
     [get_ports { led[3] }]; #IO_L23P_T3_35 Sch=LED3
```

Finally, we need to run Vivado to convert this into a BIT file which is a representation of the circuit we've designed. Sadly, it's not as easy as just passing these two files to Vivado. Circuit design is a complex process with many options. Vivado, like many EDA (Electronic Design Automation) tools, is a TCL-based tool that needs a script to operate. In my repo, I've created the simplest **`GNUMakefile`** I can to automatically download the XDC file, patch it, and run Vivado with a TCL script. I encourage you to look through it, but if you just want to get on with it, update the path variables and then a simple **`make`** under Linux should do everything and leave you with a file **`implementation/static.bit`** which is the circuit file we need to load into the Zynq chip.

So, I have my circuit file, now what? U-boot contains an FPGA bitstream loader, so we will start with that. Copy the **`FPGA.bit`** file to the MSDOS partition of your SD card, insert the card into the board, and push the reset button. At the U-boot prompt, interrupt the boot process and run the following commands:

```
Zynq> fatload mmc 0 0x4000000 static.bit
4045663 bytes read in 249 ms (15.5 MiB/s)
Zynq> fpga loadb 0 0x4000000 4045663
```

The first command loads the **`static.bit`** file from the FAT partition on the SD card into memory. The second tells U-boot to program the FPGA with the file contents now in memory at `0x4000000`. At this point, you should see two of the four LEDs on the board turn bright red. Congratulations! You've built and loaded your first FPGA design!

We could stop here, but let's do two more things before we call it quits. Let's make our LEDs blink instead of just turning on, and let's see how we can load the FPGA from under FreeBSD. That will start to lay the ground work for cool things.

To make the LEDs blink, we need to change our circuit by changing the Verilog. There is a new repo for this new circuit, but I'll summarize the changes here. First our new Verilog:

```
module top(
    input clk,
    output [3:0] led
);

localparam cycles_per_second = 125000000;

reg [3:0]leds;
reg [31:0]counter;
```

```
always @ (posedge clk)
begin
  if (counter == 0) begin
    leds <= leds + 1;
    counter <= cycles_per_second;
  end else counter <= counter - 1;
end

assign led = leds;

endmodule
```

We've now added a clock input that will drive a 31-bit counter, and we've made that and a 4-bit counter. The 31-bit counter loads with the value 123,000,000 and counts down to zero. When it hits 0, the 4-bit **leds** counter increments. We choose 125,000,000 because, looking at the Arty reference manual section 11, we see the ethernet phy supplies a 125MHz clock on pin H16.

Next, we need to tell Vivado about the clock on pin H16:

```
set_property -dict { PACKAGE_PIN H16 \
     IOSTANDARD LVCMOS33 } \
     [get_ports { clk }]; #IO_L13P_T2_MRCC_35 Sch=SYSCLK
create_clock -add -name sys_clk_pin -period 8.00 \
     -waveform {0 4} [get_ports { clk }];#set
```

Again, a simple make should build a file **implementation/blinky.bit** which can be transferred to the SD card and loaded into the FPGA as above.

Now you should the see the LEDs blink counting in binary.

Ok, before we wrap it up for this column, let's talk about one last thing. Let's see how we can program the FGPA from inside FreeBSD. Turns out this is simple. There is a **/dev/devcfg** device that was originally intended for you to simply **cat** a bit file to this device, but I don't think the work on it was quite completed. There is a simple C program **xbin2bit** which has its own git repo. if you have root permissions (by default **/dev/devcfg** is owned by root) you can simply run and pass it your bit file:

```
# xbin2bit blinky.bit
```

Did you run that and see your FreeBSD system halt, but the LEDs keep blinking? Yep, turns out our Verilog designs need to be a little more sophisticated so that the processor doesn't stop. We'll explore this more in the next column.

If you've got questions, comments, feedback, or flames on any of this I'd love to hear from you. You can contact me at articles@ChrisBowman.com.

CHRISTOPHER R. BOWMAN first used BSD back in 1989 on a VAX 11/785 while working 2 floors below ground level at the Johns Hopkins University Applied Physics Laboratory. He later used FreeBSD in the mid 90's to design his first 2 Micron CMOS chip at the University of Maryland. He's been a FreeBSD user ever since and is interested in hardware design and the software that drives it. He has worked in the semiconductor design automation industry for the last 20 years.

Adventures in TCP/IP

Dynamic Goodput Pacing: A New Approach to Packet Pacing

BY RANDALL STEWART

The previous column in this series focused on the FreeBSD infrastructure that supports pacing for TCP stacks. This column continues exploring pacing in FreeBSD by discussing a pacing methodology that is available in the RACK stack today in the developer version of FreeBSD. This pacing methodology is called Dynamic Goodput Pacing (DGP) and represents a new form of pacing that can provide good performance and yet still be fair in the network. To understand DGP, we first will need to discuss congestion control, since DGP works by combining two forms of congestion control that traditionally have not been used together. Consequently, this column will first discuss what congestion control is as well as two kinds of congestion control that DGP combines into a seamless pacing regime.

Congestion Control

When TCP was first introduced to the budding Internet, it did not contain anything called congestion control. It had flow control, i.e., making sure that a sender did not overrun a receiver, but there was no regard at all to what TCP was doing to the network. This caused a series of outages that have since been termed "congestion collapse" and brought about changes to TCP to have a "network aware" component to try to assure that actions by TCP would not cause problems on the Internet. This "network aware" component is called congestion control.

When TCP was first introduced to the budding Internet, it did not contain anything called congestion control.

Loss Based

The very first congestion control introduced to the Internet was loss-based congestion control. Today it is one of the most widely deployed forms of congestion control though it does have its downsides. There are two main algorithms (though others do exist) used in loss-based congestion — one called New Reno[1] and the other called Cubic[2]. New Reno and Cubic both share one fundamental design, Additive Increase and Multiplicative Decrease (AIMD). We will look a bit more closely at New Reno, since it is simpler to understand.

TCP will start with a number of basic variables set to preset defaults:

- **Congestion Window (cwnd)** — How much data that can be sent into the network without causing congestion. This is initialized to the value of the Initial Window.

- **Slow Start Threshold (ssthresh)** — When the cwnd reaches this point the increase mechanism is slowed down from something called "Slow Start" to "Congestion Avoidance". The ssthresh value is ostensibly set to infinity initially but will get set to ½ the current cwnd whenever a loss is detected.
- **Flight Size (FS)** — The number of data bytes in flight to the peer that has not been acknowledged. This of course starts at zero and is incremented every time data is sent and subtracted from when data is cumulatively acknowledged.
- **Initial Window (IW)** — This is the initial value to be set into the cwnd, in most implementations it is set to 10 segments (10 x 1460), but it may be more or less (initially TCP had this value set to 1 segment).
- **Algorithm – "Slow Start" (SS)** — The slow start algorithm is one of the algorithms used for the additive increase part. In slow start every time an acknowledgment arrives the cwnd is increased by the amount of data acknowledged.
- **Algorithm – "Congestion Avoidance" (CA)** — The congestion avoidance algorithm will increase the cwnd 1 packet every time a full congestion windows worth of data has been acknowledged.

So initially the cwnd is set to the IW, the increase algorithm is set to SS, and the flight size is set to 0 bytes. The implementation, assuming an infinite amount of data to send, will send out the IW worth of data towards its peer moving FS to the IW size as well. The peer will send back acknowledgments for every other packet. (Some implementations such as macOS may change to every eighth packet or every single packet.) This means that with each arriving acknowledgement the FS will go down by two packets and the cwnd will be increased by two packets, which means we can send out four more packets (if the flight size before the acknowledgment arrived was at its maximum value i.e. the cwnd). This sequence will continue until a loss is detected.

The implementation will send out the IW worth of data towards its peer moving FS to the IW size as well.

There are two ways in which we can detect loss: via indications of loss in the returning acknowledgment (where the cumulative acknowledgment point does not advance) or via a timeout. In the former case we cut the cwnd in ½ and store this new value in the ssthresh variable. If it's via the latter, we set the cwnd to 1 packet and again set ssthresh to ½ the old cwnd value before the loss.

In either case we start retransmitting the lost data and once all the lost data has been recovered, we start sending new data with a new lower cwnd value and an updated ssthresh. Note that whenever the cwnd rises above the ssthresh point we will change the algorithm used for increasing cwnd to congestion avoidance. This means that once a whole cwnd of data has been acknowledged we increase cwnd by one packet.

Now in briefly summarizing how loss-based congestion control works I have skipped over some finer points (more details on how to recognize loss for example) and some other nuances. But I wanted to give you an idea as to how it is working so we could then shift our attention to routers on the Internet to focus on what happens because of these loss-based mechanisms.

Routers typically have buffers associated with their links. This way, if a burst of packets

arrives (which happens often), they do not have to discard any packets but can forward the packets to the next hop through whatever link that leads there. Especially when the link speeds vary between incoming versus outgoing. Let's look at Figure 1 below.

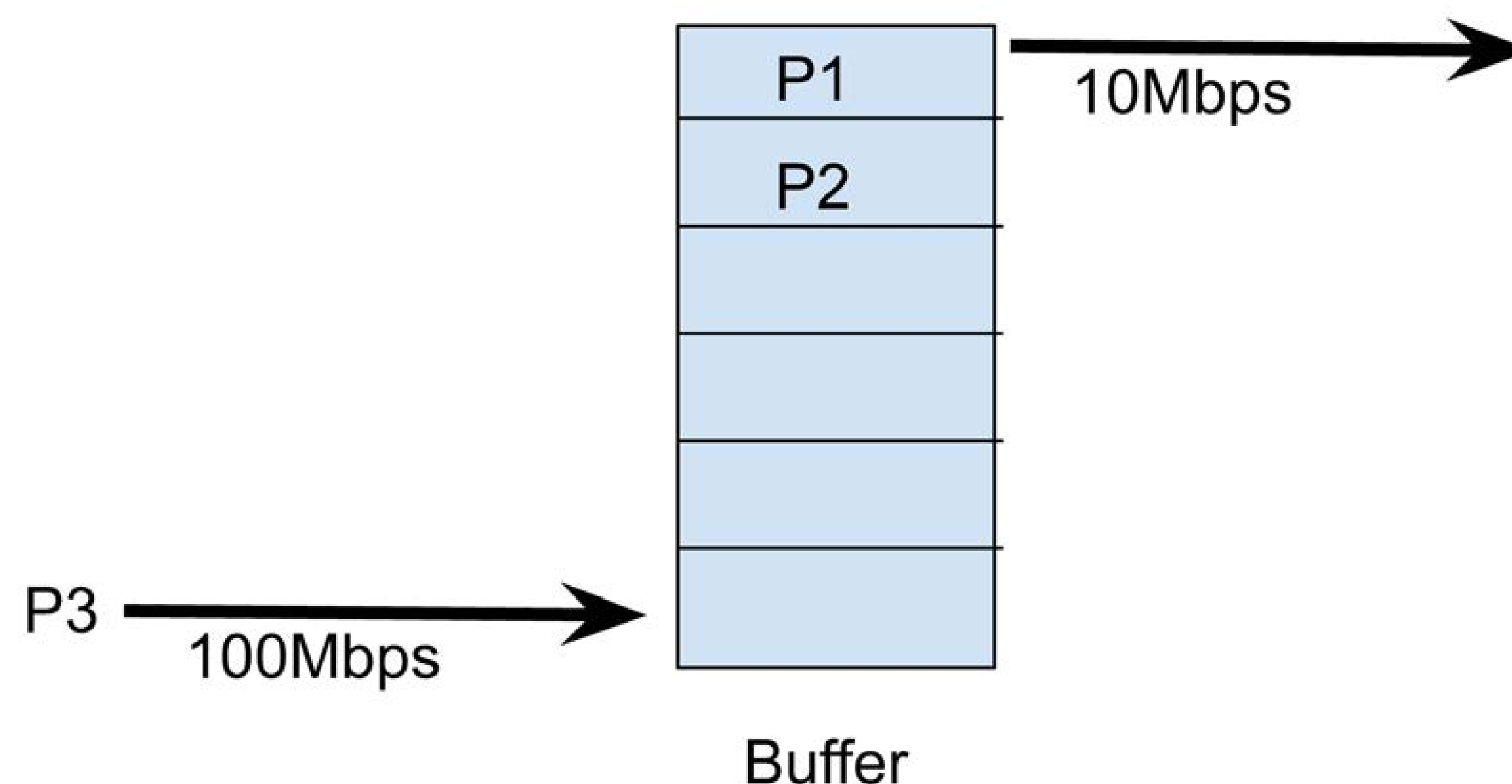


Figure 1: A bottleneck router with a buffer

Here we see P3 arriving at 100Mbps and it will get placed into the third slot in the router's buffer. P1 is currently being transmitted onto the 10Mbps link and P2 is waiting for its turn to be transmitted. Assuming a 1500-byte packet P3 will take approximately 120 microseconds to transmit across the 100Mbps network. When it is its turn to go out onto the 10Mbps destination network it will take 10 times that or 1200 microseconds to be sent. This means that every packet in the router's buffer will cause a 1200 microseconds of additional delay to be added to the packet just arriving.

Now let's step back and consider what our congestion control algorithm is going to optimize for. It will send packets as fast as it can until it loses a packet. If we are the only ones sending on the network this means that we will have to completely fill the routers buffer before a loss occurs. This means we are optimizing the router to always have a full buffer. And since memory is cheap, routers have grown quite large buffers. This means that we end up with long delays when a large transfer is happening via a loss-based congestion control algorithm. In Figure 1 we see only 6 slots for packets but in real routers there can be 100's or 1000's of packets in a routers buffer waiting to be sent. This means that the round-trip time seen by a TCP connection might vary from just a few milliseconds (when no packets are in queue) and then spike up to seconds due to buffering by the routers and TCP's AIMD congestion control algorithms always wanting to keep the buffer completely full.

If we are the only ones sending on the network this means that we will have to completely fill the routers buffer before a loss occurs.

You may have heard the term "buffer bloat" which impacts any real time applications (video calls, audio calls or games), this is directly caused by loss-based congestion control and is what we have just described.

Delay Based

For quite a long time, researchers and developers have known about the tendencies of loss-based congestion control to fill buffers. Long before all the talk of buffer bloat alternative congestion controls had been proposed to solve this issue. One of the first such proposals was TCP Vegas[3]. The basic idea in TCP Vegas is that the stack keeps track of the

lowest RTT it has seen, called the "Base RTT". It uses this information during Congestion Avoidance to determine an expected bandwidth i.e.:

Expected = cwnd / BaseRTT

The actual bandwidth is also calculated as well i.e.:

Actual = cwnd / CurrentRTT

Then a simple subtraction is done to determine the difference i.e.:

Diff = Expected – Actual

The difference *Diff* is then used to determine if the cwnd should be advanced or reduced based on two thresholds $\alpha < \beta$. These thresholds help to define how much data should be in the buffer of the bottleneck. If the difference is smaller than α then the cwnd is increased and when the difference is larger than β then the cwnd is decreased. Whenever the difference is between the two then no change is made to the cwnd. This clever formula with low values (usually 1 and 3) keeps the buffer at the bottleneck to a very small value optimizing the connection to keep the buffer just full enough to achieve optimal throughput for the connection.

During Slow Start, TCP Vegas modifies the way the increase works by alternating every other RTT. The first RTT Slow Start increases as New Reno or other loss-based congestion control mechanisms would. However, on the next RTT, TCP Vegas does not increase the cwnd but measures the difference using the cwnd to again calculate if the router buffer has been saturated. When the actual rate falls below the expected rate by one router buffer, slow start is exited.

Testing with TCP Vegas shows improvements to both RTT and throughput.

Perils of Mixing the Two

Testing with TCP Vegas shows improvements to both RTT and throughput. So why did we not fully deploy TCP Vegas gaining all its benefits?

The answer to that is contained within what happens when a loss-based congestion-controlled traffic competes against a delay based one. Imagine your TCP Vegas connection faithfully tuning the connection to keep only 1 or 2 packets in the bottleneck routers buffer. The RTT is low, and your throughput is at your maximum share. Then a loss-based flow begins, it will of course fill the router buffer until it experiences a loss, which is the only way it learns to slow down. To the TCP Vegas flow a signal that it is going too fast is received repeatedly, getting it to continue to cut its cwnd until it is getting almost no throughput. In the meantime, the loss-based flow gets all the bandwidth. Basically, the two types of congestion control, when mixed, always end up turning out poorly for the delay-based mechanism. Since loss-based congestion control was and is widely deployed on the Internet this then provided a huge dis-incentive for deploying a delay-based congestion control.

Mixing Loss and Delay Based Approaches with DGP

DGP attempts to integrate both loss-based and delay-based approaches in choosing its pacing rate. For the delay-based component Timely[4] was chosen (with some adaptation for the Internet) though arguably any delay-based approach (including TCP Vegas) could have been adapted for this purpose. Timely uses a delay gradient to calculate a multiplier which is combined with the current loss-based congestion controls calculations (either New Reno or Cubic) to derive an overall pacing rate using the following formula:

*Bw = max(GPest, LTbw) * TimelyMultiplier*
FillCwBw = cwnd / CurrentRTT
PaceRate = max (Bw, ((FCC == 0) ? FillCwBw : min(FillCwBw, FCC)))

We will discuss each part of the above formula in the following subsections to give you an idea of how DGP works. For the deep details on Timely we recommend you read the paper[4].

Goodput (GPest)

One of the foundational measurements that DGP keeps track of is the goodput. This is like BBR's[5] delivery rate but different in a subtle way. The delivery rate calculates the arriving rate of all data at a TCP receiver. When there is no loss the delivery rate and the DGP goodput are identical. But in cases of loss, the DGP rate lessens. This is because the goodput is measured strictly on advances to the cumulative acknowledgment (cum-ack), when a loss happens the cum-ack stops advancing. All the time it takes to recover a lost packet is thus folded into the goodput estimate lowering the GPest value.

To measure the goodput initially the IW is allowed to be sent in a burst, this starts the very first measurement window. The goodput is usually measured over 1 - 2 round trips worth of data and is calculated based on the advancement of the cum-ack over that period. During the measurement period a separate RTT is also calculated over that period i.e. the curGpRTT (which will be used later as input to Timely). Once the IW is acknowledged we have a seed of the first measurement. For the next three measurements the estimate is averaged. Once a fourth measurement is made future estimates use an apportioned weighted moving average to update the current GPest. Every time a new GPest is started the curGpRTT is saved into the prevGpRTT and a new weighted moving average of RTT is also begun which will become our new curGpRTT (note this RTT is a separate measurement from the smoothed round trip that TCP continues to make as well). The GPest measurement is continually made by the sender when data is in transit to the receiver. Any time that the sender becomes application limited the current measurement is ended. Note that an implementation becoming congestion window limited does not stop the current measurement. This description has been rather brief and may warrant a future article on how the RACK stack measures the goodput.

To measure the goodput initially the IW is allowed to be sent in a burst, this starts the very first measurement window.

Long Term Bandwidth (LTbw)

DGP also tracks another bandwidth measurement termed the LTbw. The LTbw is the total sum of all bytes cumulatively acknowledged divided by the total time that the data was outstanding. This value is almost always lesser than the current goodput value but in cases of sharp decline in the bandwidth measurement it can provide a stability to the current bandwidth estimate.

Delay Gradient with Timely (TimelyMultiplier)

Timely provides a multiplier that generally ranges somewhere between 50% - 130% of the estimated bandwidth. Timely uses the following formula (from the paper):

Algorithm 1: TIMELY congestion control.

Data: new_rtt
Result: Enforced rate
new_rtt_diff = new_rtt - prev_rtt ;
prev_rtt = new_rtt ;
rtt_diff = (1 - α) · rtt_diff + α · new_rtt_diff ;
    ▷ α: EWMA weight parameter
normalized_gradient = rtt_diff / minRTT ;
if *new_rtt* < T_{low} **then**
    rate ← rate + δ ;
        ▷ δ: additive increment step
    return;
if *new_rtt* > T_{high} **then**
    rate ← rate · $\left(1 - \beta \cdot \left(1 - \frac{T_{high}}{new_rtt}\right)\right)$;
        ▷ β: multiplicative decrement factor
    return;
if *normalized_gradient* ≤ *0* **then**
    rate ← rate + N · δ ;
        ▷ N = 5 if gradient<0 for five completion events
    (HAI mode); otherwise N = 1
else
    rate ← rate · (1 - β · normalized_gradient)

Timely was designed for the data center environment where the RTT's and bandwidths at various points are known entities. For use in DGP this is not the case, so we substitute the new_rtt and prev_rtt in the above formulas with the curGpRTT and the prevGpRTT respectively. We only do a Timely calculation at the end of making a goodput estimate. The multiplier calculated then stays with the connection as is until the next goodput estimate is complete and the multiplier is again updated along with any update to the goodput. Note also that timely uses a minRTT i.e. the minimum expected RTT. Again, this is not something known on the Internet as compared to the data center where the RTT at any point is a known quantity, and so it is derived as the lowest RTT seen in the last 10 seconds, the same as BBR. Also, just like BBR, to reestablish the minimum RTT periodically DGP will go into a "probeRTT" mode where the cwnd is reduced to 4 segments for a short period of time so that a "new" low RTT can be found. Note that the addition of a BBR style probe-RTT phase also helps DGP to become more compatible with BBR flows it is competing with.

With these tweaks, the Timely algorithm is adapted into DGP. For the deeper details on either probeRTT or Timely I suggest reading the papers.

Loss Based Pacing or Filling the Congestion Window (FillCwBw)

To pace out packets for loss-based congestion control a simple method exists. Take the currentRTT (kept in any stack doing Recent Acknowledgement[6]) and divide that into the congestion window. This tells the pacing mechanism what rate to pace at that will spread the current congestion window over the current RTT. Any loss-based congestion control, New Reno or Cubic, can be used with this method to simply deduce a pacing rate that would be dictated by the congestion control algorithm. We call this rate the Fill Congestion Window rate (FillCWBw) since it is designed to fill the congestion window over an RTT.

It should be noted that by pacing packets out over the entire congestion window it is highly likely that the sender will have less loss. This is due to less pressure on the bottleneck

by allowing some time between each microburst of packets sent. This time allows the bottleneck to drain some before the next microburst of packets arrives. Having less loss will naturally mean that the congestion window will gain a higher value since loss is the only thing that causes the cwnd to be reduced.

The Fulcrum Point: Fill Congestion Window Cap (FCC)

So far, DGP has calculated a bandwidth based on the goodput estimate in combination with Timely to increase or decrease that rate based on the RTT gradient (a delay-based component). We have also calculated a bandwidth for pacing based on the value of the congestion window (via whatever congestion control is in play) and the current RTT (the loss-based component). This gives us two distinct bandwidths we could pace at.

So, this is where the Fill Congestion window Cap (FCC) comes into play. If one is set (you can set it to zero to always get the fastest bandwidth), it becomes the limit of how much we will allow the loss-based rate to apply. The current default in FreeBSD is set to 30Mbps. So, for example if the FillCwBw calculated out to 50Mbps and the Bw, factoring in the Timely value on top of the estimate bandwidth came out to 20Mbps, then we would pace at the limit of the FCC i.e. 30Mbps in a default setting. If the Timely calculated Bw was 80Mbps then we would pace at 80Mbps.

What happens here is that the FCC serves as a Fulcrum point and limit to how much the nominal loss-based congestion control algorithm will influence the pacing rate. The FCC declares that your connection will push against other loss-based flows to maintain a rate of at least FCC, if possible, based on the congestion control value. If neither value meets the FCC limit, then the larger of the two will be dominant.

General Performance While Testing on the Internet

In a past large-scale experimentation at my previous company, DGP running with an FCC limit of 30Mbps (now the default) reduced RTT by up to over 100ms with no real degradation in Quality of Experience (QoE) metrics. If the FCC point was raised to 50Mbps QoE metrics improved i.e. things like Play Delay and Rebuffers improved with a sacrifice of little to no reduction of the RTT. The 30Mbps setting was adopted as a default in response to this testing, valuing the reduction in RTT (indicating much better router buffer behavior) than the corresponding gain in QoE metrics.

Enabling DGP in FreeBSD

There are at least two ways of enabling DGP on a FreeBSD system that has the RACK stack loaded and set as the default stack. If the source code of the application is available, you can add to the source code the setting of the socket option `TCP_RACK_PROFILE` to a value of ‘1’ as follows:

```
socklen_t slen;
int profileno, err, sd;

….
profileno = 1;
slen = sizeof(profileno);
err = setsockopt(sd, IPPROTO_TCP, TCP_RACK_PROFILE, &profileno, slen);
```

The above code snippet will enable DGP on the socket associated with `sd`.

Another mechanism if you do not have access to the source code is to use **sysctl** to set the default profile for all TCP connections using the RACK stack to the value of '1'. You do this as follows:

```
sysctl net.inet.tcp.rack.misc.defprofile = 1
```

Note that once this value is set, all TCP connections using the RACK stack will use DGP with a default FCC value of 30Mbps. You can change that default (the FCC) as well to better match your network conditions with **sysctl** as well. The **sysctl**-variable **net.inet.tcp.rack.pacing.fillcw_cap** holds the FCC in bytes per second. For example, if I want to set the value to 50Mbps the following command can be used:

```
sysctl net.inet.tcp.rack.pacing.fillcw_cap = 6250000
```

The default value is 3750000 i.e. 30Mbps, you take the value you would like set in bits per second and divide by 8. So, 50,000,000 / 8 = 6,250,000.

You can also use the **TCP_FILLCW_RATE_CAP** socket option if you have access to the source code as follows:

```
socklen_t slen;
int err, sd;
uint64_t fcc;
….
fcc = (50000000 / 8);
slen = sizeof(fcc);
err = setsockopt(sd, IPPROTO_TCP, TCP_RACK_PROFILE, &fcc, slen);
```

Note that this will change the FCC value for just the specified connection and not the entire system.

You can also turn the FCC feature off and pace at always the maximum allowed by either Timely or the congestion control by setting the FCC value to 0. This will likely give you the best performance but will not reduce router buffer usage and thus buffer bloat.

How to Set Parameters?

So what settings are right for your network? In most cases the bottleneck is in your home gateway so knowing the bandwidth of your Internet connection can give you a good idea on what the FCC value should be set to for your connection. For example, I have two sites I administer, one is a symmetric 1Gbps connection, my FCC value for that machine I leave at the default of 30Mbps. This of course only affects outbound TCP connections using the RACK stack where the server is sending data. Leaving the default implies that for the most part delay-based performance will be coming out of my server and each connection will only push to maintain 3% of the network uplink capacity with loss-based mechanisms.

In my second system it has an asymmetric cable modem and only has 40Mbps up. In such a situation I have my FCC point set to 5Mbps. If I get more than 7 connections, they will start to push against each other using the loss-based mechanisms all attempting to get at least 5Mbps.

Future Work

Currently the FCC point is set in a static fashion on the entire system. This means that often the value is suboptimal, and a better value could possibly be selected (possibly gain-

ing both performance and reductions in RTT). The author is currently working on a more dynamic mechanism for setting the FCC point. The basic idea is that the connection would measure, over some time, the actual path capacity. Then once a value is available for the "Path Capacity Measurement" (PCM) a set percentage of that would be dedicated as the FCC point. This would then in theory make DGP more dynamic in tuning to the network path being used while reserving and pushing for some portion of the available bandwidth specific to each network type. Hopefully the work will be completed in 2025. Once completed, the RACK stack will change its default to enable DGP.

References

1. S. Floyd, T. Henderson: "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 6582, April 1999.
2. S. Ha, I. Rhee, L. Xu: "Cubic: A New TCP-Friendly High-Speed TCP Variant", in: ACM SIGOPS Operating Systems Review, Volume 42, Issue 5, July 2008.
3. L. Brakmo, L. Peterson: "TCP Vegas: End to End Congestion Avoidance on a Global Internet", in: IEEE Journal on Selected Areas in Communications, Volume 13, No. 8, October 1995.
4. R. Mittal, V. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Gohabdi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats: "TIMELY: RTT-based Congestion Control for the Datacenter", in: ACM SIGCOMM Computer Communication Review, Volume 45, Issue 4, August 2015.
5. N. Cardwell, Y. Cheng, C. Gunn, S. Yeganeh, V. Jacobson: "BBR: Congestion-Based Congestion Control", in: Queue, Volume 14, Issue 5, December 2016.
6. Y. Cheng, N. Cardwell, N. Dukkipati, P. Jah: "The RACK-TLP Loss Detection Algorithm for TCP", RFC 8985, February 2021.

RANDALL STEWART (rrs@freebsd.org) has been an operating system developer for over 40 years and a FreeBSD developer since 2006. He specializes in Transports including TCP and SCTP but has also been known to poke into other areas of the operating system. He is currently an independent consultant.

My EuroBSDCon Experience in Dublin

BY STEFANO MARINELLI

Before the Conference

The idea of taking part in EuroBSDCon first came to mind in 2023. Coimbra would have been a great location for returning to a conference after many years, but unfortunately I couldn't make it. When the call for registration in Dublin approached, I decided that would be the right occasion. In the meantime, the BSD Cafe project had been launched, and the welcoming enthusiasm it received provided a strong push. I thought I might propose a talk. My work involves BSD systems daily with a variety of clients, and sharing how I've managed (and still manage) the transition from Linux to the BSDs might interest the community.

Sharing how I've managed the transition from Linux to the BSDs might interest the community.

I still remember that moment in June when I clicked "Submit". I told my wife, "Whether as a speaker or as a participant, in September we're going to Dublin." She works with me, so she would also attend the conference. It was one of the best ideas in recent years.

I still recall when, on my way to the office, I glanced at my smartwatch notification: my talk had been accepted. In an instant, I felt two distinct, intense emotions: enthusiasm and terror. I have no issues with public speaking, but doing it again in English at such an event… In reality, that fear was unfounded, as I would only discover two months later.

As the event drew closer, the organizers provided all the necessary information. They were extremely patient, even when I asked seemingly trivial questions. It was my first EuroBSDCon, and the "BSD" at the center of the name should have taught me something every BSD user knows: everything is documented in the finest detail. The FAQs were frequently updated. It's a known *feature*: BSD-related documentation is always impeccable.

Photos by Carla Suffritti

We left on Thursday, as we wouldn't be attending the tutorials. It was my first time in Ireland, my first BSD Conference, and my first time as a speaker. Upon arrival, everything was perfectly organized: directions for transportation, accommodation at the speakers' hotel with a splendid view of the Irish Sea. The organizers know how to pamper their speakers.

That evening we went down to dinner and I was nervous, being terrible at recognizing

faces. I was sure I would run into someone I had known online for a long time but wouldn't recognize.

The Day Before the Talks

Friday morning was dedicated to reviewing my notes and slides. Each presentation had a 45-minute slot, including questions, and I was pretty sure I'd go slightly over. In my test runs, I managed to stay under 50 minutes, though not under 45. The advantage of having the last talk before the Social Event is that a small time overrun wouldn't affect other speakers, and I could "postpone" questions to the Social Event. In any case, I was satisfied with the result. I didn't yet know how the audience would receive my talk, but the community's warmth and flawless organization already made me feel part of something special.

In the morning, I received an email from Philipp Buehler suggesting a quick trip to the venue in the afternoon to test the projectors. The goal was to avoid any last-minute technical issues before my talk that would waste precious time. So, after lunch, we took a pleasant stroll through Dublin, heading to the conference venue.

The Venue and Early Impressions

EuroBSDCon was held at University College Dublin (UCD), in the O'Reilly Hall — a location I found extremely fitting and comfortable. At the entrance there were two areas: on the right, check-in and registration (with badges in three colors: orange for organizers and staff, white for participants, and green for speakers); on the left, a zone with t-shirts and more staff and friends. On arrival, as soon as I introduced myself, the welcome was warm and friendly. Henning Brauer handed us our badges — seeing that green badge filled me with pride. With him was Katie McMillan, whom I had never interacted with even online, but I had seen her presentation video from BSDCan, so it was a real pleasure to meet her. At the other table were several people, including Mischa Peters (who gave me an OpenBSD. Amsterdam t-shirt), my "colleague" Jeroen Janssen (aka h3artbl33d, one of the admins of the Mastodon instance exquisite.social), Peter Hansteen, Paul de Weerd, René Ladan, Janne Johansson, Guido Van Rooij, Michael Reim, Benny Siegert, and others. People I've respected for years, all gathered there. We talked for over an hour, with others joining and leaving as time passed. I already knew the BSD community was positive, but I was surprised at how cohesive the various groups were. I immediately felt at ease, that sensation you get when you're in a familiar place among dear friends.

On arrival, as soon as I introduced myself, the welcome was warm and friendly.

At a certain point, Franco Fichtner arrived to do the same technical check. OPNSense is one of my first choices for router/firewall solutions for clients, and chatting with him directly was a real pleasure.

When the tutorials resumed, participants went back to their sessions. We returned to the hotel for a bit of relaxation and another review of my talk, as well as to write the "I Solve Problems" blog post I would publish once back home.

Saturday: The Conference in Full Swing

Saturday morning, 9:30: registration and participants arriving. I was positively anxious, looking at everyone's badges to read names and try to identify and remember! Right in the entrance hall, I started meeting people: Vanja, Toni, and Natalino approached and we introduced ourselves, talking a bit about our roles. Among the buzz, I appreciated being able to speak some Italian. Alfonso Siciliano also arrived, and we talked quite a lot. Alfonso is pleasant, friendly, and extremely competent and serious, with the same qualities I recognized in Leonardo Taccari (from the NetBSD team).

I took a walk around to see how everything was arranged. There were three conference rooms (Foyer A, Foyer B, and Stage End) and a wonderful, bright, spacious hall overlooking the UCD lake. This hall offered tea, coffee, and water. There was a table where people had left stickers and other swag — I had brought many BSD Cafe stickers and coasters, which I distributed personally and left on that table, and they seemed appreciated. There were also sponsor tables and, at the center, a table reserved for the FreeBSD Foundation. I was delighted to meet Deb Goodkin and Kim McMahon in person — talking with them was a real pleasure. Hearing about the Foundation's ideas and projects directly from them was fantastic.

Meanwhile, the technical team was fine-tuning the last details — including the whims of a Decimator device — promptly handled by Michael Dexter, who swiftly hooked up his laptop and replaced its firmware. The wonders of being among highly competent professionals!

At 10:30, everyone moved to the Stage End room for the Opening Session, where everyone greeted each other, the program was presented, and some useful information was provided. At 11:00, Tom Smyth presented the Keynote: *Evidence based Policy formation in the EU what Evidence are we Presenting to the EU?*

Tom, positively emotional, presented with passion, competence, and pride. His message and information were comprehensive and valuable, and I believe everyone appreciated his talk. At the end, after well-deserved applause, the gift for the speakers was handed out: a marvelous green merino wool scarf with an inscription in Ogham. Choosing which talks to attend was the hardest part. They were all interesting, but the three tracks ran in parallel, forcing tough decisions. Fortunately, I knew the presentation videos would be available later.

I attended Franco Fichtner's talk: *Tooling Around With FreeBSD — A tale of scripting a custom firewall distribution*. This was very interesting for someone like me who often uses

OPNSense. Understanding some of their design decisions was enlightening, as it helped me grasp the project's direction.

Meeting People and Sharing Experiences

One of the people I had the opportunity to meet — whom I already knew online through social interactions and his role in the BSD Now podcast — is Jason Tubnor. Jason is friendly and upbeat, and he asked if he could do a brief interview about the event, me, BSD Cafe, and my talk. I gladly accepted, and we did it during the lunch break. There was a fully equipped room upstairs, perfectly set up for the purpose. BSD-style organization — always efficient.

During the lunch break, I also got to meet and chat with Benedict Reuschling.

During the lunch break, I also got to meet and chat with Benedict Reuschling. I'm very grateful to Benedict (and Jason and the other presenters) for the BSD Now podcast and for introducing me to Jim Maurer about a year ago, enabling me to write my first article for the *FreeBSD Journal*. Still in the Foundation's sphere, I had a pleasant conversation with Ed Maste. Truthfully, I'm a somewhat shy person, and I felt like I might be "bothering" people by just approaching them. For instance, I didn't get to speak with Colin Percival, Allan Jude, Dan Langille, and others — including the great Jon "Maddog" Hall. I'll catch up next time!

Lunch was served in the hall. There were small portions of various dishes being handed out. There were also tables with different kinds of sweets, all very good. During the break, it was possible to meet and chat with many people — too many to list. I fear I'd forget some, and that would be a pity, because everyone attending the event is, in some way, linked to the IT, Open Source, and BSD world — people I'd love to talk with for hours, not just minutes.

After lunch, I chose to attend Nicola Mingotti's talk: *An introduction to GPIO in RPi3B+ and NetBSD, building a wind-speed logger as an application*. I'd already spoken with Nicola before the event, and the NetBSD + Raspberry Pi combo is something I also frequently use. Nicola presented one of his setups, the issues he faced, and how this solution effectively solves his problems.

The next three talks all covered topics of immense interest to me. Unable to decide, I used that slot to review my own talk, settling into a very comfortable lounge area in the UCD building. Soft couches allowed me to focus. My greatest fear was skipping a part or forgetting something important. While I was there, I met Dave Cottlehuber, and we talked a bit about various things, including email system management. Meeting Dave confirmed the impression I had formed of him online: a pleasant and friendly person, as well as extremely knowledgeable.

The following talk I attended was by Kim McMahon: *How You Can Advocate for FreeBSD — And How We Can Help*. I was very interested in this, and Kim delivered it brilliantly at the Stage End room. I try to advocate only for solutions I use and trust, without any barriers. Today, FreeBSD can tackle the vast majority of challenges I and my clients face, effectively and efficiently. But I'm not a trained communicator, so getting advice from a professional like Kim was helpful.

The next talk choice was easy: Foyer A, Walter Belgers: *Hacking — 30 years ago*. I chose it both for my interest in the topic (I love real-life experiences) and because my talk would be next in that same room. Also, I knew this talk wouldn't be recorded, so this was a unique chance to see it live. Walter shared fascinating anecdotes and stories in an ironic and engaging way. A different era, a different type of computing, a completely different notion of security compared to today. Yet some things never change, providing a sense of continuity over time.

My Talk

The moment had come to get up and move towards the speaker's station. Suddenly, adrenaline surged — and then receded. Some people left the room, others entered. I was busy connecting my laptop and barely noticed what was happening around me, except that — much to my pleasure and honor — Professor Marshall Kirk McKusick stayed to listen to my presentation. He's another person I didn't have the "courage" to approach this time, but I will next time.

Patrick McEvoy, efficient and professional as always, helped me put on the microphone, took his position, and nodded. Everything worked perfectly.

"I'm Stefano Marinelli. I Solve Problems."

People sat down, Henning introduced me. The stage was mine. It was time to tell my story: how nearly 30 years ago, a CD-ROM set of Linux distributions, and how, over 22 years ago, meeting a teacher (Özalp Babaoğlu — one of the fathers of original BSD), buying a laser printer (convincing my parents it was for "university purposes"), and printing out the FreeBSD Handbook all led me to where I am now. Thanks to a teacher, a printer, and a passion, here I am among friends, and these friends are here to listen to my story. I instantly became calm. I started timidly, but my shyness lasted only a few minutes.

"I'm Stefano Marinelli. I Solve Problems." I saw smiles. The reference was understood. No doubt about that.

As I continued speaking, I saw the audience's attention grow. Over the years, interest in BSDs had somewhat waned. Many big companies, after ignoring open source solutions for years, began embracing Linux and its ecosystem. While this gave a significant boost to open source in general, it indirectly reduced the adoption of other operating systems like FreeBSD. Sometimes the reason is corporate politics, know-how ("it's easier to find people experienced in Linux"), or purely ideological or commercial motives ("Everyone knows what Linux is, so it sells better"). But I am doing the exact opposite. I don't disdain Linux, but I prefer the BSDs. And people wanted to know how that was going.

At one point, as I mentioned my blog post about a NetBSD server running unattended

for over 10 years, I saw a guy in the front row open his eyes wide: "I can't believe it! You're that guy!" A wonderful moment: he had been following my blog for a long time and had attended my talk without realizing I was the same person. After the talk, we spoke for a while, and he said some very kind things. I greatly appreciated it. Thank you, Raymundo Soto!

At the end of my presentation (I exceeded the time limit by just a bit), Henning gave me the speaker's scarf and suggested postponing questions to the Social Event. Still, a few people approached immediately, and I was happy to chat and answer questions. If these people dedicated an hour of their time to listen to me, the least I could do was listen to their ideas, experiences, and opinions.

As I was leaving, I met Max Stucchi and Salvatore Cuzzilla. After a handshake, Max confirmed that the GUFI (Gruppo Utenti FreeBSD Italia) is still alive and invited me to join, which I did with great pleasure.

The Social Event

We then headed to the bus stop that would take us to the Social Event, held at the BrewDog, a very distinctive building in the Docklands area, overlooking the River Liffey. The bus was significantly late, so we waited outside for a while. I was still relieved and happy about how my talk had been received, so that wait was pleasant. We chatted with others, and finally the bus arrived. We hopped on the upper deck and traveled about 25 minutes to our destination.

We made our way through the characteristic Docks to reach the pub. After scanning our badges, we were given three drink tickets. After the positive tension of the talk, it felt great to relax, chatting with colleagues, enjoying good food and good beer, experiencing a real Saturday night out in Dublin. Masanobu Saitoh, who had come all the way from Japan to attend the conference, came by my table to express his appreciation. It meant a lot to me, and one of his photos is among the best taken of me at the event.

Around 22:30, we decided to head back to the hotel and called a taxi. On the way down, we met Robert Clausecker, who was going the same way, and decided to return together.

My mood was sky-high. I saw how my wife looked at me, happy to see me so calm and positive, already thinking about the next day.

Sunday: Wrapping Up

On Sunday morning, the event started half an hour later. Surprise upon waking: I was almost voiceless. Talking so much over the last few days, plus the Dublin climate — so different from Italy's — probably played equal parts. This further prevented me from interacting with many people I would have liked to meet.

Sunday's keynote was presented by Kent Inge Fagerland Simonsen: *Is our software sustainable?* I was very, very interested in this topic since I'm quite sensitive to the concept of sustainability in IT. I am convinced that optimization (both hardware and software) is crucial, especially in the medium/long term. There's no point in making hardware more energy-efficient and powerful if we bloat the software so much that it negates or even worsens the overall situation.

Unfortunately, I couldn't attend, because system administrators don't have fixed working hours. That night, two physical servers decided to fail simultaneously. Since these were important servers and Sunday was another conference day, I preferred to fix them before leaving the hotel, arriving at the conference a bit later. Nothing that a zfs-send, zfs-receive, and a DNS update couldn't handle. I waited for the data to transfer and ensured everything was fine. Users never noticed a thing, which made the effort worthwhile. FreeBSD, bhyve, and jails once again helped minimize problems and downtime.

More Talks and the Family Photo

For the next talk, it was tough to choose — they were all very interesting. I ended up at Alexander Bluhm's presentation: *A Packet's Journey Through the OpenBSD Network Stack*. It was very interesting. Alexander showed, step by step, the path of packets and explained the decisions made along the way. There were many questions and answers, making it even more engaging. At the exit, I met Sven Ruediger, who had just presented his work. We exchanged a few words. I'm sorry I missed his talk, as I heard very positive comments about it.

People often don't know what FreeBSD can do for them.

After lunch, everyone gathered for the traditional family photo. In just a few minutes, we assembled outside the UCD hall — with the lake behind the photographer — and took the shot. Over 200 people were quickly lined up in an organized manner. Even here, the BSD community's efficiency was evident. It was a joyful, positive moment. The term "family" photo, rather than "group" photo, perfectly conveys the atmosphere.

Then it was time for another challenging choice: Kirk McKusick, David Brooks, or Jason Tubnor. I chose Jason's talk: *Building a SD-WAN appliance suitable for an Australian Health Sector NFP/NGO*. Jason has a similar approach to mine, detailing interesting reasons and issues he faced while building his infrastructure. I also like to solve problems using technologies, not just "boxes," so I really enjoyed his presentation.

Next, I chose Michael Dexter's talk: *FreeBSD and Windows Environments*. Michael is constantly involved with OpenZFS, jails and bhyve — three essential tools in my work — and his presentation was, as usual, brilliant, informative, and extremely inspiring. He gave me another reason to migrate some Windows servers from Linux/KVM to bhyve, and the results have been excellent. At the end of Michael's presentation, Patrick McEvoy came up to tell me he greatly enjoyed my talk. That meant a lot to me. Patrick is someone I highly respect, and his opinion matters to me.

Unfortunately, luck is blind, but misfortune sees all too well. While I was getting ready for Dan Langille's talk, *Doing stupid things with FreeBSD jails*, I received a flurry of server alerts and had to rush out to solve them. Fortunately, the wifi connection was excellent, and I managed to intervene, but I missed the talk. I was free again midway through the next session, and it didn't feel right to enter halfway. So, I used the time to talk to Deb and Kim about my advocacy ideas. People often don't know what FreeBSD can do for them. That's why I try to show, by sharing my story and blog articles, that BSDs are not "untamable beasts" but our friends.

The Closing Session

At the end, everyone gathered at Stage End for the closing session. I learned about some traditions (like the FreeBSD Foundation raffle — how did Vanja get that Lego guitar home, given its size?) and the auction of "lost" items. The money went to https://www.womensaid.ie/.

The closing session was fun and informative. There were thanks to all sponsors and friends of the event, a link to download the Family Photo was provided, and we heard about the next events like AsiaBSDCon and BSDCan. Then came the much-anticipated moment: the announcement of the next EuroBSDCon location. I hoped it would be reasonably convenient, since it was now clear that EuroBSDCon would be a must-attend event for me. And… Zagreb!. Just a few hours' drive from home, so I must attend. Mischa made me promise that, since I can go by car, I'll bring more BSD Cafe gadgets. I will.

Finally, there were goodbyes. It was a heartfelt farewell, with promises to meet again soon. I thanked and congratulated everyone I could, praising the technical, organizational, and content quality of the event. I received the same warmth and affection from all. But…

…I still had one last sticker in my pocket. Just one, because a BSD Cafe user (Kaveman) had told me he would attend the conference, but we hadn't met yet. Right at the end, just before leaving, we found each other. It was a pleasure to give him the sticker I had saved for two days, just for him.

Final Thoughts

Even Liam Proven (whom I didn't speak to during the event) attended, and a few weeks later, he wrote an article about it and my talk on The Register — an article that I really appreciated.

After meeting all these people and attending the talks, I realized that the BSDs are more alive than ever, that development is ongoing, and that the FreeBSD Foundation and the developers have very clear plans on how to move forward, what is needed, and how to proceed to make it happen. It was a conference that greatly enriched both me and the other attendees, because it was led by people who work daily WITH the systems being discussed. Little commercial hype, a lot of real technical content. So much substance. Tremendous human and technological value.

EuroBSDCon was a memorable event for me. I doubt I can fully express in words all the emotions and positivity it conveyed. The BSD community is inclusive, open, and collaborative, and the event showcased this spirit in every participant. Collaboration, not competition. The BSD community sometimes stays quiet because it's focused on creating rather than "selling." In my view, this is a huge advantage and a strong point.

Janne Johansson summarized my feelings perfectly on IRC:

"If you saw a short guy smiling ALL THE TIME it was Stefano Marinelli. He seemed super happy to be there every time I saw him (which is the correct way to feel on EuroBSDCons ;))"

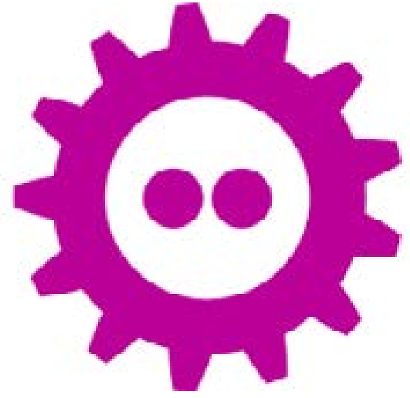
STEFANO MARINELLI is an IT Consultant with over two decades of experience in the realms of IT consulting, training, research, and publishing. His expertise spans across operating systems, with a special emphasis on *BSD systems — FreeBSD, NetBSD, OpenBSD, DragonFlyBSD — and Linux. Stefano is also the 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.

Events Calendar

BSD Events taking place through March 2025

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.



FOSDEM 2025

February 1-2, 2025
Brussels, Belgium
https://fosdem.org/2025/

FOSDEM is a two-day event organized by volunteers to promote the widespread use of free and open source software. Taking place February 1-2, 2025, FOSDEM offers open source and free software developers a place to meet, share ideas and collaborate.

SCALE 22X

March 6-9, 2025
Pasadena, CA
https://www.socallinuxexpo.org/scale/22x

SCaLE 22X – the 22nd annual Southern California Linux Expo – will take place March 6-9, 2025, in Pasadena, CA.

SCaLE is the largest community-run open-source and free software conference in North America.

AsiaBSDCon 2025

March 20-23, 2025
Tokyo, Japan
https://2025.asiabsdcon.org/

AsiaBSDCon is a conference for users and developers on BSD based systems. The next conference will be held in Tokyo, Japan, March 20-23, 2025. The conference is for anyone developing, deploying, and using systems based on FreeBSD, NetBSD, OpenBSD, DragonFlyBSD, Darwin, and MacOS X. AsiaBSDCon is a technical conference that 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.