



®

FreeBSD[®] **JOURNAL**

April/May/June 2025

NETWORKING

Interview with Kajetan Staszewicz

Installer Usability

Overlord is Deploy Jails as Fast
as You Code

Netgraph

Getting Started with WiFi Development

Embedded FreeBSD: Custom Hardware

Adventures in TCP/IP: Static Pacing



FreeBSD[®] JOURNAL

The FreeBSD Journal is Now Free!

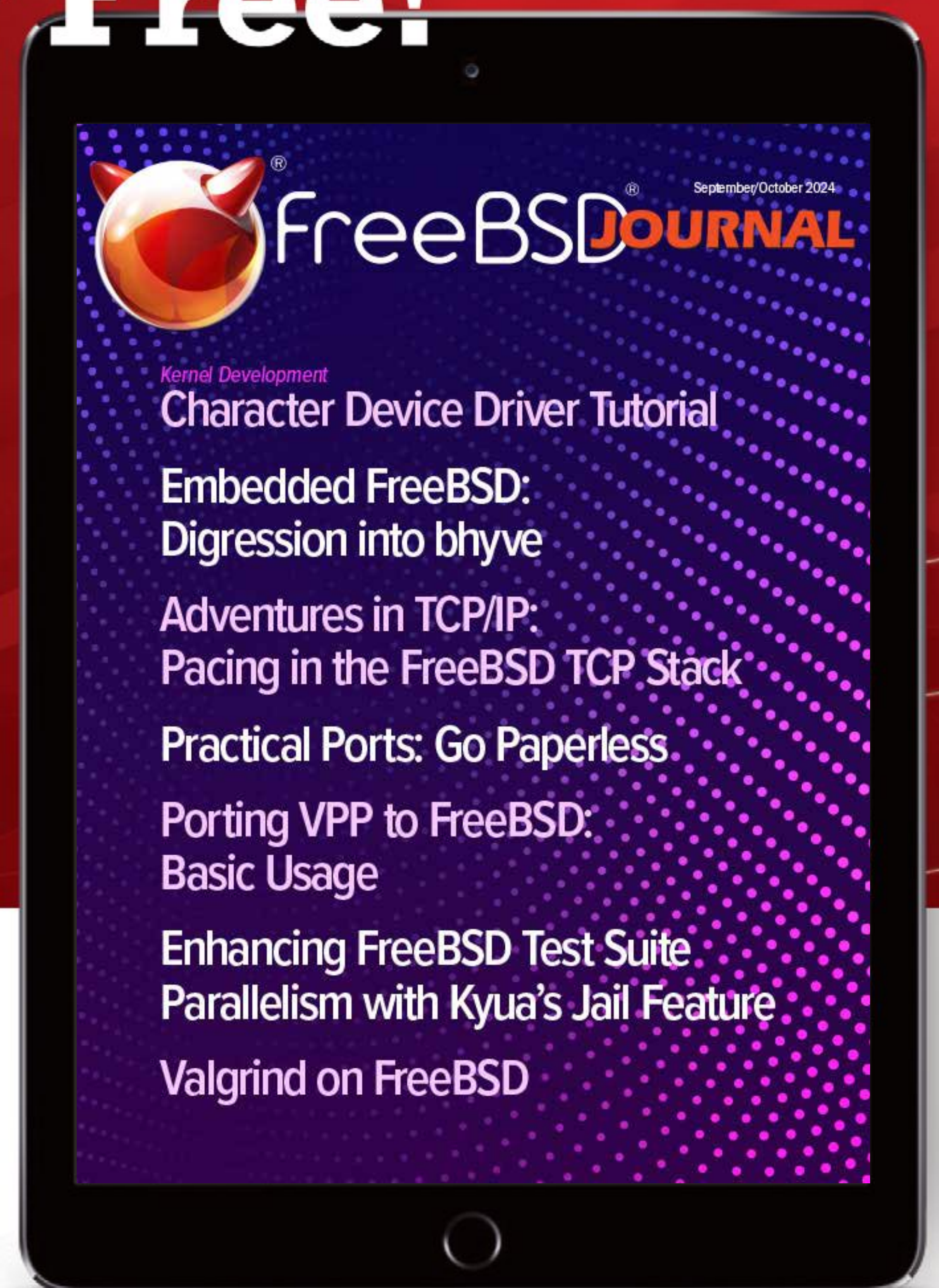
Yep, that's right Free.

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 Embedded
- Oct/Nov/Dec FreeBSD 15.0



Find out more at: freebsd.foundation/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 4 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@freebsd.foundation.org

Copyright © 2025 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

Welcome to the latest issue of the *FreeBSD Journal*! This time, we're focusing on networking, not just the technology, but also the tools and improvements that help connect people and systems more easily and securely.

In this issue, you'll learn about Overlord, a fast and flexible way to deploy jails as you code. We're also featuring a closer look at installer usability, including a comparison with other OS installers, lessons learned, and progress on improving FreeBSD's own installation experience. If you're curious about FreeBSD's wireless support, don't miss "Getting Started with WiFi Development". Plus, all the regular content and columns you've come to expect.

The *Journal* is made possible thanks to the generous support of donors like you. If you find value in these articles—or in the FreeBSD Project as a whole—please consider [donating](#) to the FreeBSD Foundation. Your support helps us fund development, improve tools and infrastructure, and grow the community.

Thanks for reading, and enjoy the issue!

Deb Goodkin

Executive Director, FreeBSD Foundation



TOPIC: NETWORKING

- 7** Starting FreeBSD Development:
Tom Jones Interviews Staszekiewicz
By Tom Jones
- 10** Installer Usability
By Pierre Pronchery
- 30** Overlord is Deploy Jails as Fast as You Code
By Jesús Daniel Colmenares Oviedo
- 77** Netgraph
By Daniel Bell
- 85** Getting Started with WiFi Development
By Tom Jones
- 3** Foundation Letter
By Deb Goodkin
- 5** We Get Letters
By Michael W. Lucas
- 96** Embedded FreeBSD: Custom Hardware
By Christopher R. Bowman
- 102** Adventures in TCP/IP: Static Pacing
By Randall Stewart and Michael Tüxen
- 109** Events Calendar
By Anne Dickison



**Dear Computer Touching Advice Columnist,
FreeBSD's jails have a few different ways to
network. How do I decide which to use?**

—Baffled and Distressed

Dear BAD,

I'm writing this at BSDCan 2025. Two stories above us, someone is using a hammer drill. Traversing the concrete building shifts the vibration so that it exactly matches the resonance frequency of human teeth. Turns out Mister Hammer Drill needed to demonstrate that he was doing work so that he'd make his fifty bucks for a day's labor. I gave him sixty bucks to knock it off and go home. Do not fear, this unexpected expense did not impact my gelato fund; I embezzled the cash from the con.

This is my problem only because I failed to sidestep quickly enough and they made me con chair, which shows that certain parts of the community need better methods of leader selection. Being drafted has left me baffled and distressed.

The point is, that we are all baffled and distressed. We're merely baffled and distressed about different things.

Jail networking is one of those topics that seems simple until you touch it. No matter what virtualization system you use, jails or bhyve or some lesser system, you face similar decisions. Jails can either share their host's networking stack or have their own stack. On a shared stack, a jail can have its own address or be bound to a private address on a loopback interface and access the outside world via NAT. You could give a jail a full network stack with VNET. All are disasters, but each disaster possesses unique qualities that might make it preferable.

Remember, a jail is a lie.

A jail thinks it is a full operating system and expects to have full control over whatever network it accesses. If you assign an IP (public or private) to a jail, the jail thinks it fully con-

**Jail networking is one
of those topics that seems
simple until you touch it.**

trols that address. The jail's `rc.conf` includes no information on configuring interfaces. With a shared stack, the host can stomp all over the jail's networking. Want to run `sshd(8)` on your jails? By default, the SSH daemon listens on every IP address attached to the system. You must reconfigure it to change that behavior. That's not hard, but it's easily overlooked. Similarly, if you want multiple jails running the same database server? They can't all use the same port and IP address. The amount of administrative tracking you must perform is directly proportional to the number of jails sharing that network stack.

Given all this, why not give each jail its own networking stack? Because it's more complicated. You must create a bridge on the host and then create a virtual interface for every jail. One end of the interface goes into the jail; the other gets plugged into the bridge. Every complication is a bright shiny chance for failure.

The good part of VNET is that each jail truly controls its IP addresses. You must set it in the jail's `rc.conf` and can set up each service to monopolize that address.

A VNET jail isn't exactly like a host. The most common interface is `epair(4)`. You cannot capture packets on an `epair`; if you want to debug the network or capture netflow, you must do so from the host. You might think of netflow as a network administrator thing, but that's not true. Netflow solves nothing but identifies everything. If I had written my netflow book using `nfdump` rather than flow tools, it would still be useful, but I have a talent for poor choices such as making a career in system administration. I'm not saying that a knowledge of promiscuous mode, `fdisk`, or window scaling leads to misery, but I will say that happy folks have no understanding of any of them. Anyway, you don't get promiscuous mode on an `epair` interface. Plan ahead.

Both shared stack jails and VNET jails can use private or public addresses. If you use a private address, the jail can't access the outside world without outbound NAT. Incoming connections must be mapped through the NAT. As per Rule of System Administration number 19, "the purpose of any private addressing scheme is to increase administrative overhead."

You might reasonably ask what I chose to do. My hosting environment has a limited number of public IPv4 addresses and an entire IPv6 /64. I could use a bastardization NAT/direct access system, but, instead, I declare that certain jails are IPv6-only and can only access the modern parts of the Internet. I have no problem declaring that certain people are unworthy of connecting to my private package server that nobody outside my /64 can download from.

Using only public addresses is much easier. Unfortunately, IPv4 addresses currently retail for about \$55 USD each with a minimum order of 256, and I can't embezzle enough because BSDCan doesn't have that kind of money.

Anyone know how EuroBSDCon's finances are? Do they need a con chair? My new assistant is great with a hammer drill.

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



Despite his best efforts, **MICHAEL W LUCAS** (<https://mwl.io>) has been unable to persuade the *FreeBSD Journal* board to fire him from this column. Yet. His most recent books are *Run Your Own Mail Server* and *Laserblasted*. His next book is a second edition of *Networking for System Administrators*, at <https://mwl.io/n4sa2e>

Starting FreeBSD Development

AN INTERVIEW WITH KAJETAN STASZKIEWICZ
BY TOM JONES

TJ: Can you tell me a bit about yourself and your background with FreeBSD?

KS: My name is Kajetan Staszkievicz. I was born in 1984 in the magical city of Kraków in Poland. I was introduced to computers when I was around 8 years old, starting with 8-bit machines like ZX Spectrum and Atari 65XE in a computer club. Later my parents got me a Commodore 64. I have a lot of fond memories of this machine and still, I have a working one on my desk.

In 1996 we got our first PC clone at home and very soon the Internet became available for home users in Poland through a dial-up connection. This was good enough only for a short time and around 1999 my neighbors and I decided to build a neighborhood network. We were around 15 people sharing a 115kb/s connection to an Internet cafe over 10Mb/s Ethernet cables hanging haphazardly from the roof to the balconies of our flats.

Internet sharing was done by a Linux box running IP chains provided by the sysadmin from the Cafe, which was soon replaced by another system managed by some older friends from our network. Later I took over managing it myself. I wrote some CGI scripts in Perl to manage the DHCP server and traffic shaping through a web interface.

With this knowledge, I managed to get a part-time job at a company running a network not unlike ours, albeit a commercial one. At that time, I was also studying computer science at the Jagiellonian University. One of my classmates had introduced me to FreeBSD, possibly version 5.0. At my new job, routing and traffic shaping had been performed by Linux systems, but FreeBSD's jails seemed like the perfect solution for our hosting platform. We needed to serve web pages, email, webmail, and so on. Virtualization was not yet a thing back then and with jails I was able to accomplish the separation of those services.

My next job was at a company using FreeBSD for most of its services, some employees were brave enough to even use it on their workstations.

Currently, I live in Hamburg, Germany. At my current job, FreeBSD is our choice for routers and load balancers.

“

My next job was at a company using FreeBSD for most of its services, some employees were brave enough to even use it on their workstations.

”

TJ: What were your steps to making your first FreeBSD contribution? How did you pick what to work on to begin with?

KS: The choice was simple: I had to fix what was broken. At \$WORK we depend on source tracking of pf(4), and we came across a rather nasty bug where deleting source nodes caused each source node to scan all states. With hundreds of source nodes and hundreds of thousands of states, this caused a lot of looping in kernel code, effectively locking the system.

With the help of glebius@, I've managed to fix it and land my first patch to FreeBSD.

TJ: How did you track down this problem? Were there existing tools in FreeBSD that helped, or did you have to develop your own?

KS: I already had some general idea when the issue happened, because it was caused by our custom tool we use for load balancing.

At the time I didn't have much experience with FreeBSD, especially with kernel development, so I went with the dumb brute force approach: debugging by adding printf's until I found where the code froze.

It was very helpful, though, that the system comes with all the tools needed to build a custom kernel.

The ability to produce memory dumps on demand which then can be analyzed with a debugger comes in very handy too.

TJ: You must have learned a lot in that process. Are there any shortcuts you would like to share for new developers that would make the debugging and development you had to do easier?

KS: It's not only knowledge but also additional tools and workflows that make my job easier.

The first one is having a proper development and testing environment. At \$WORK we only use FreeBSD on bare metal and my initial bug fixes had to be deployed on this hardware. Those are "real" servers, which means that they POST for a few minutes before they boot, and they all serve production traffic affecting our customers.

Testing bug fixes in such conditions was very stressful and slow. Nowadays I have 3 tiers of systems involved: virtual machines which can boot new kernel code in seconds, testing hardware which gets some traffic, but if they break, customers won't be affected, and staging hardware, which handles a bit more traffic, but still no paying customers.

Another thing is the testing suite of FreeBSD. That is a real lifesaver!

I can test my changes within a few minutes (yes, it can run in parallel!) and be sure that I'm not breaking any existing functionalities. Sadly, I occasionally find that there are things for which no tests have been written yet. But we're slowly getting there.

The last thing that comes to my mind is the option to enable various debugging options during kernel compilation. One can track lock order, various variable assertions, and even in-

“

My next job was at a company using FreeBSD for most of its services, some employees were brave enough to even use it on their workstations.

”

valid memory access in the kernel.
The *FreeBSD Developers' Handbook* documents those options quite well.

TJ: Is there any advice to new contributors or anything you would like to add?

KS: Open source projects, including FreeBSD, depend on contributors. FreeBSD is a mature project, and even though there is a lot of complex code, there's always room for contributions, including small and simple ones. The community is friendly, supportive, and always happy to help with code reviews and constructive feedback. Don't be intimidated!
I hope that contributions can bring in new, useful features, and give everyone, including the newcomers, a chance to not only learn new skills and grow professionally but also do something fun with computers.

TOM JONES is a FreeBSD committer interested in keeping the network stack fast.

KAJETAN STASZKIEWICZ is a system administrator specializing in networks and virtualization. He occasionally contributes to FreeBSD, primarily working on the Packet Filter.

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>

Installer Usability

BY PIERRE PRONCHERY

Dear reader,

Writing this article, I have no idea who you are. This might be the first article you've ever read about FreeBSD; or you might already be well-versed with the system because you wrote half of it after all. I do not have the slightest idea about the hardware you are reading this article on either; it might not even have been made yet, and I do not know if you have any disabilities or impairments.

Well, this is exactly the kind of challenge the installer of an Operating System has to deal with. It somehow needs to work in every situation, while being appealing or usable for everyone, possibly even anticipating future evolutions. It is a tough task.

An installer is often the first time a prospective user interacts with a given system, and it is therefore responsible for the first impression. At the same time, a power user may not have to use the installer very often on personal hardware — basically when setting up a new system — but may want to automate the installation of dozens of systems for various tasks, or even for customers when deployed in a professional context.

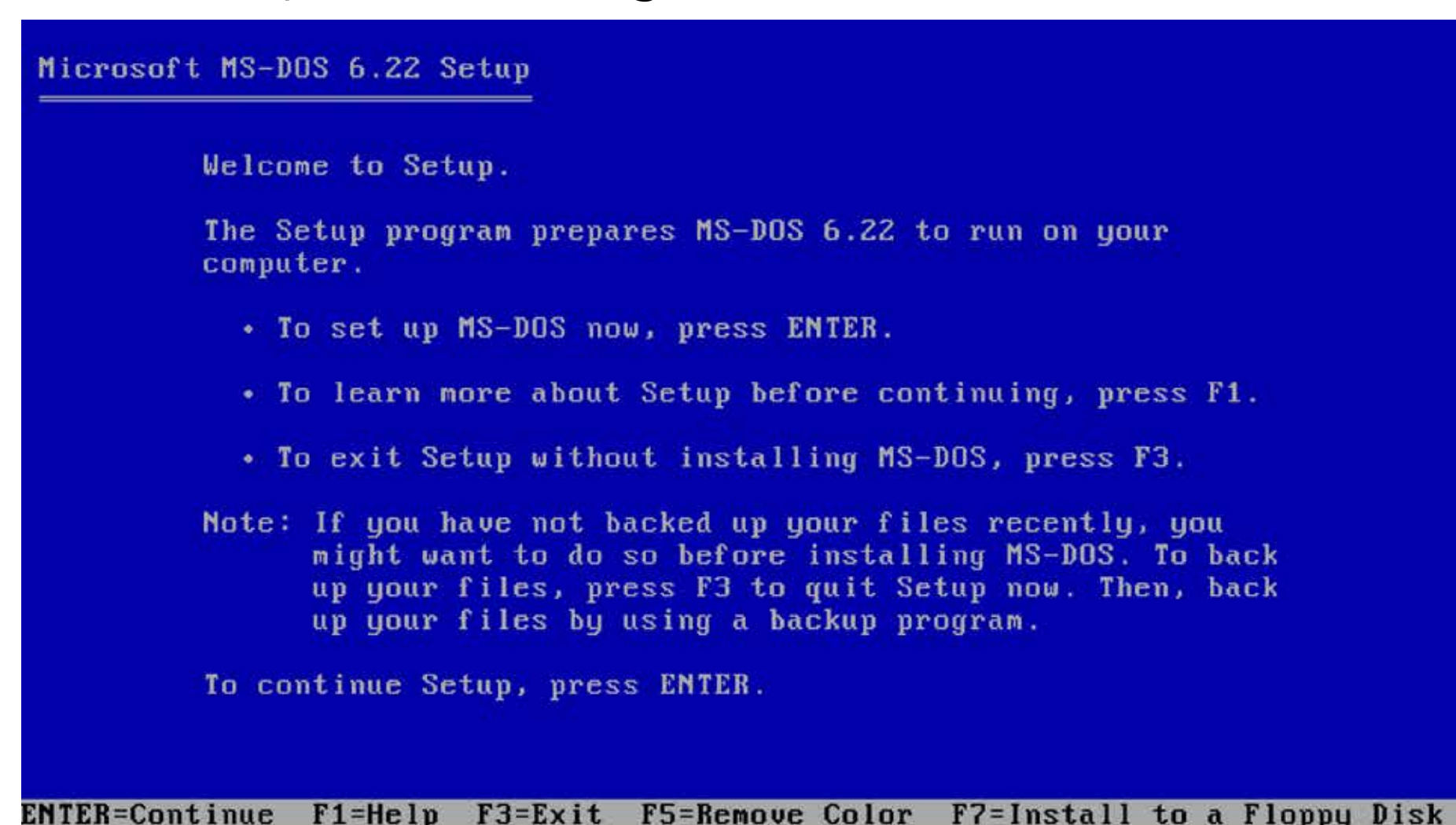
All things considered, these different aspects can be approached under a single perspective: usability. But what is usable? And more importantly, how usable is the FreeBSD installer in comparison to other systems?

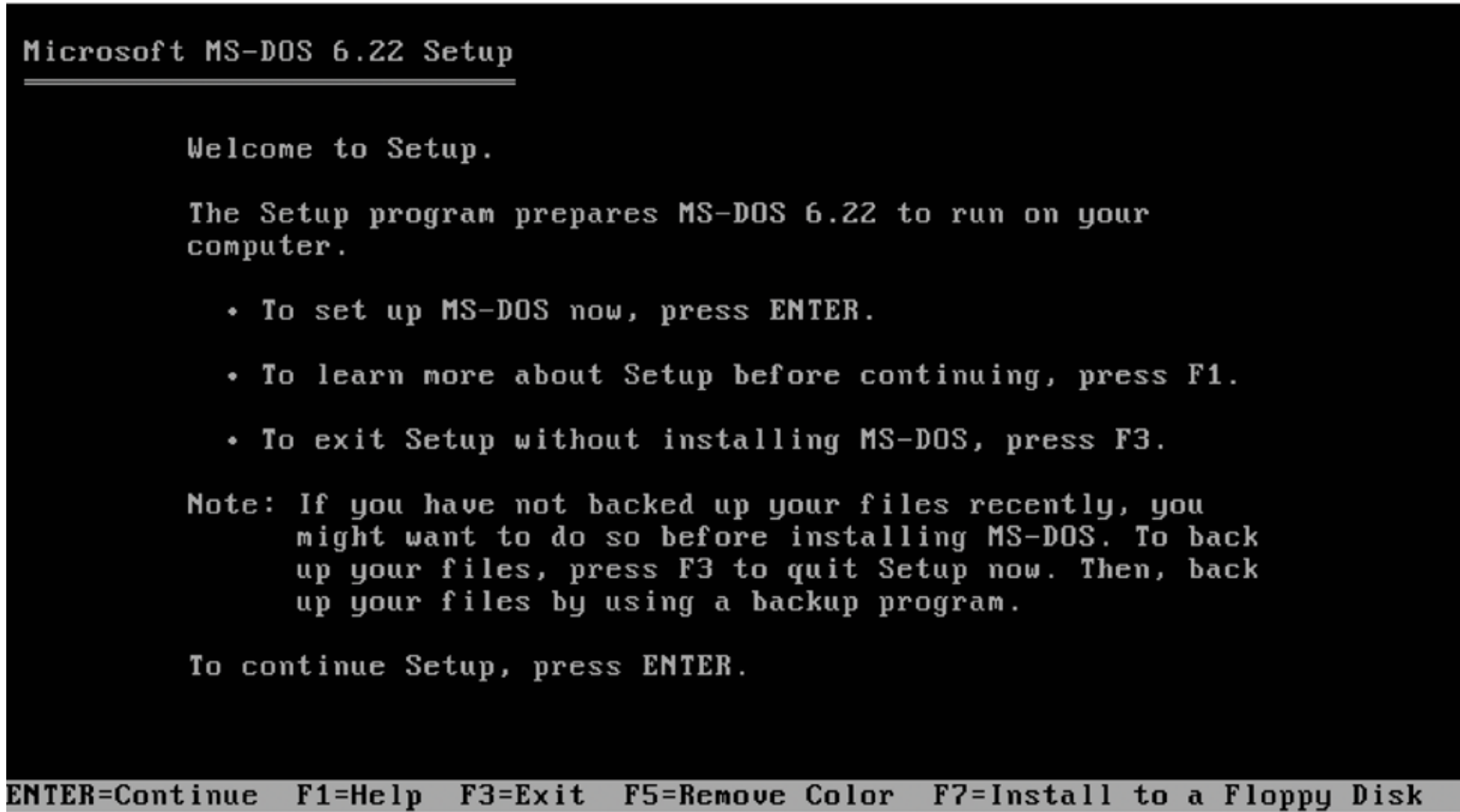
An installer is often
the first time
a prospective user
interacts with
a given system.

State of the Art

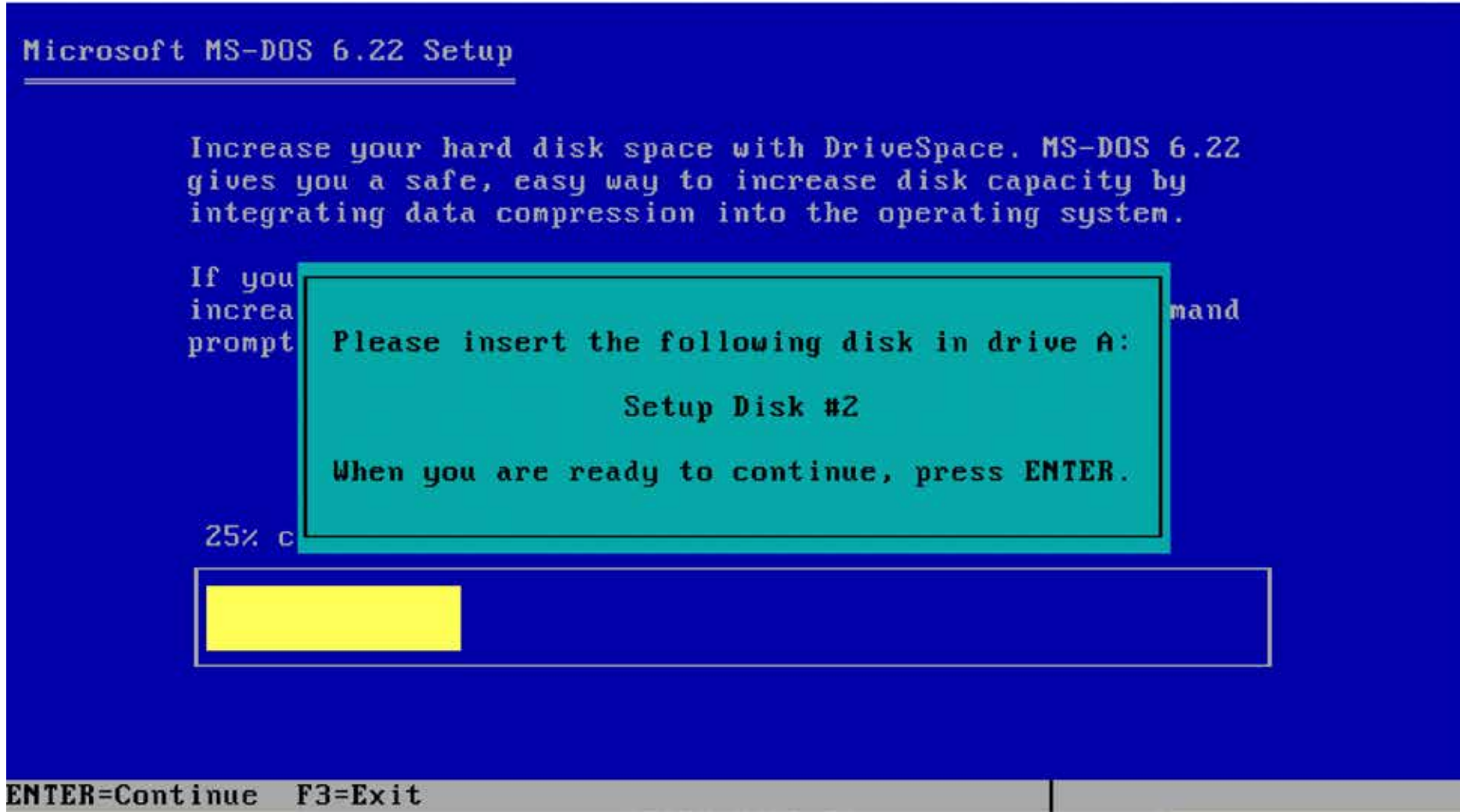
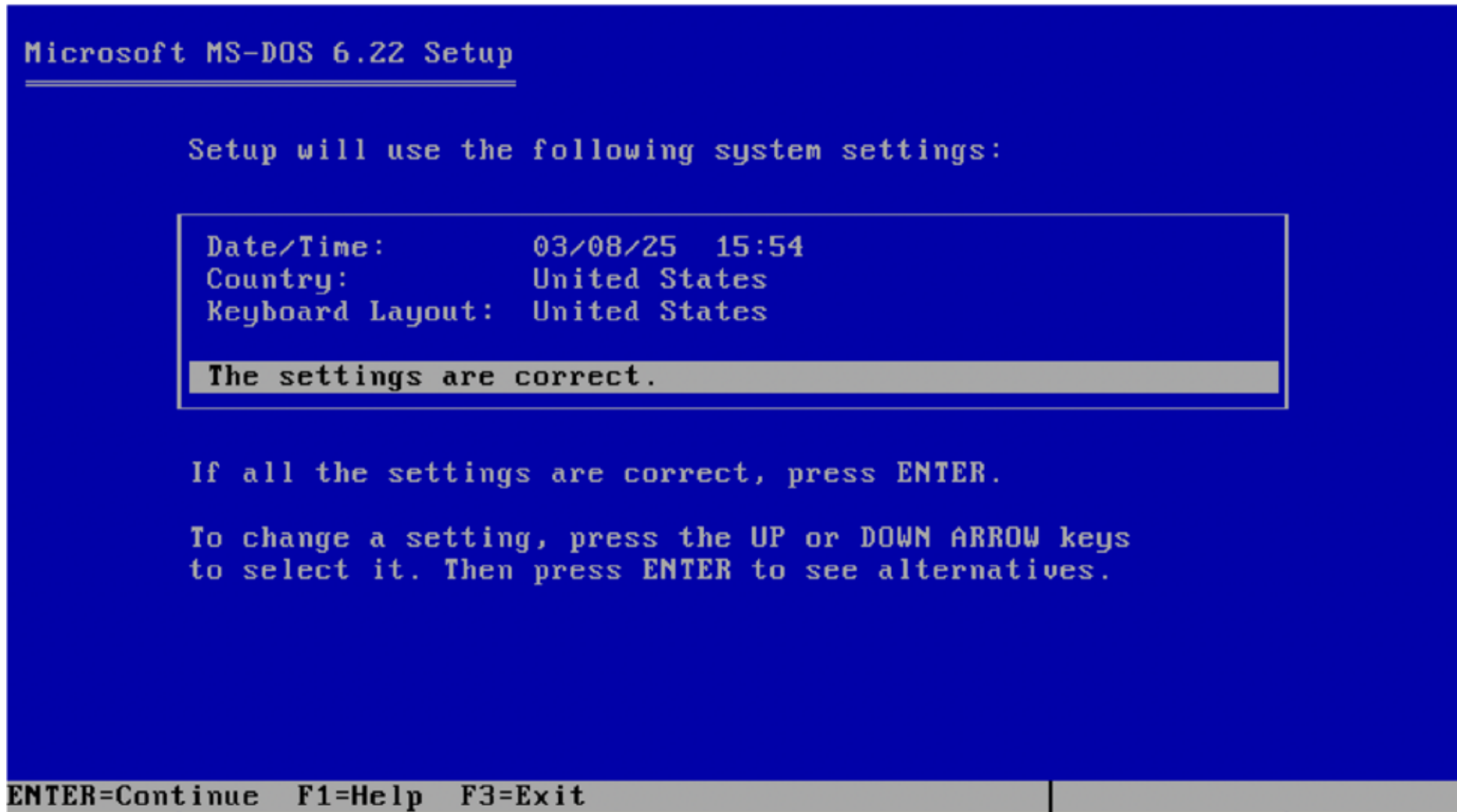
Evolution of Microsoft Windows

Installers go back a long way. My first contact with installers takes me back to the introduction of a computer into my household around 1989. At the time, this usually meant installing MS-DOS, and then installing Windows 3.1 on top. Obviously, installing MS-DOS was performed in text mode (80-columns VGA) but it was possible to disable colors — which doubled as a crude accessibility feature: high contrast.

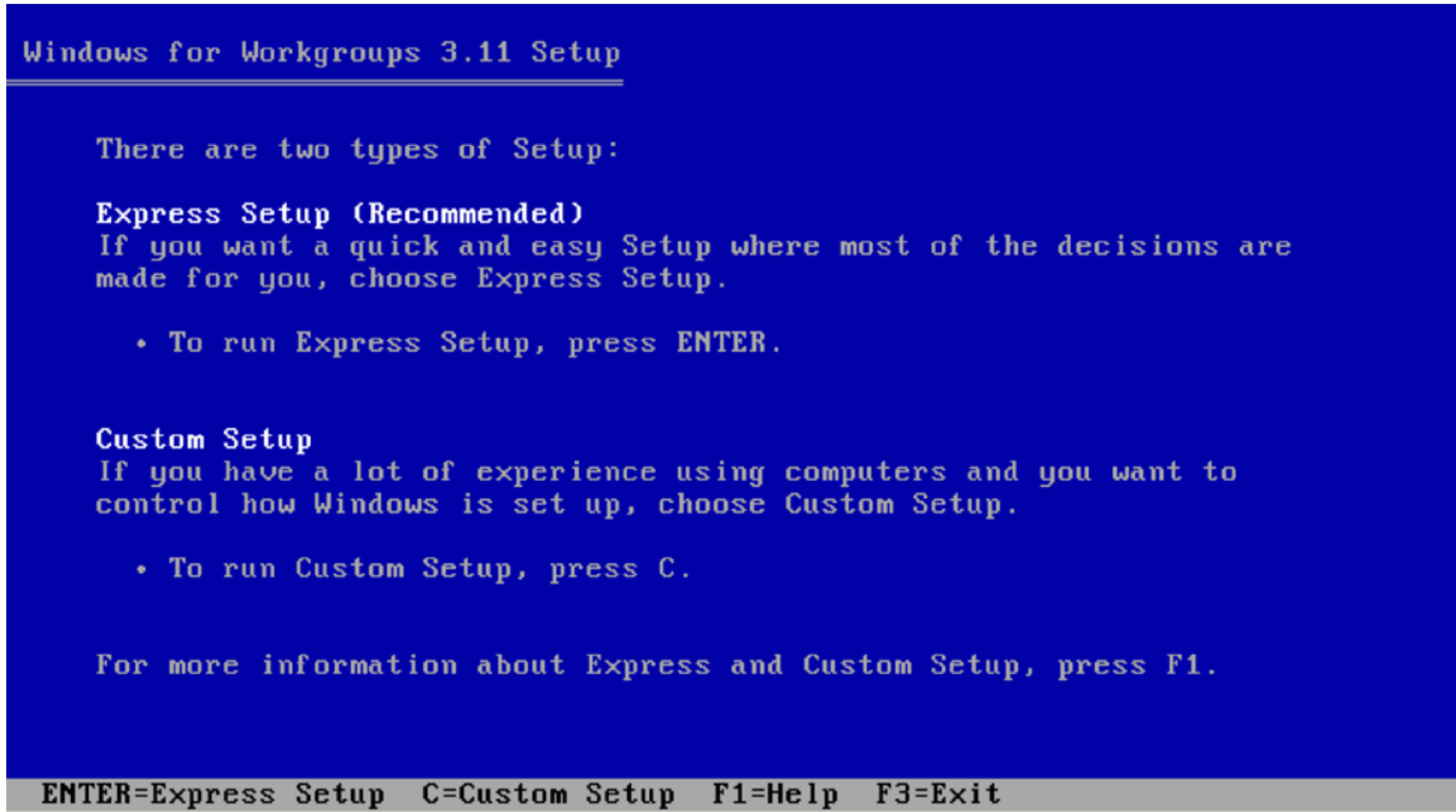




While performed in text mode, the installation process already featured a progress bar, and specific highlighting for user input.



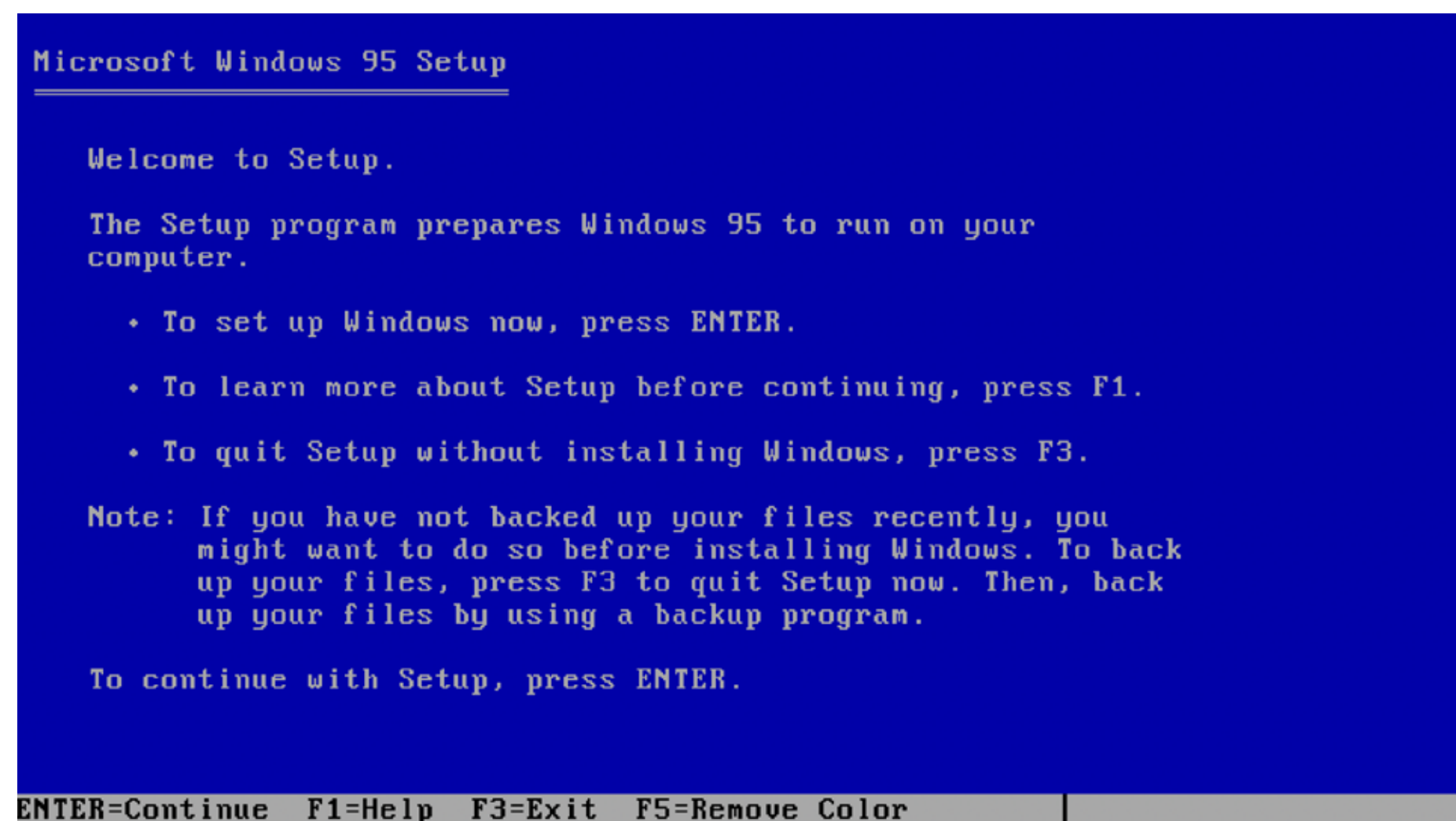
On the other hand, the installer for Windows (here in version 3.11) was quickly able to use its own graphical interface for the installation, as illustrated here.

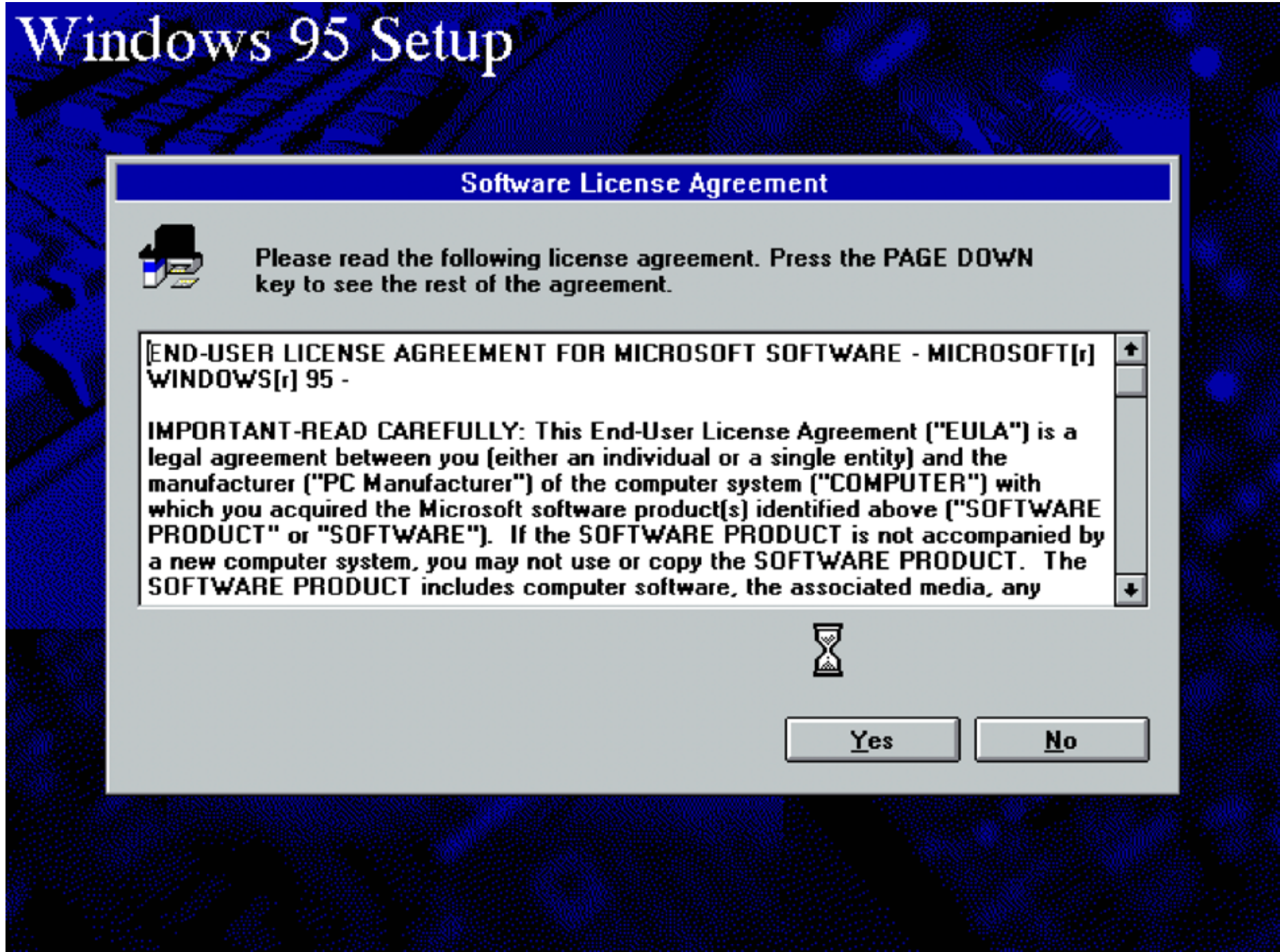




It is worth noting that I have created these screenshots myself, in VirtualBox, from material downloaded safely from archive.org! It seems that Microsoft has published the necessary files as public domain over there.^{1,2,3}

Going forward brings us to Windows 95. Just like Windows 3.11, it leveraged its own user interface in the installer, albeit with two intermediate steps. Here again, the screenshots tell a story, and it was relatively painless to install; the number of steps requiring user interaction was noticeably higher.⁴



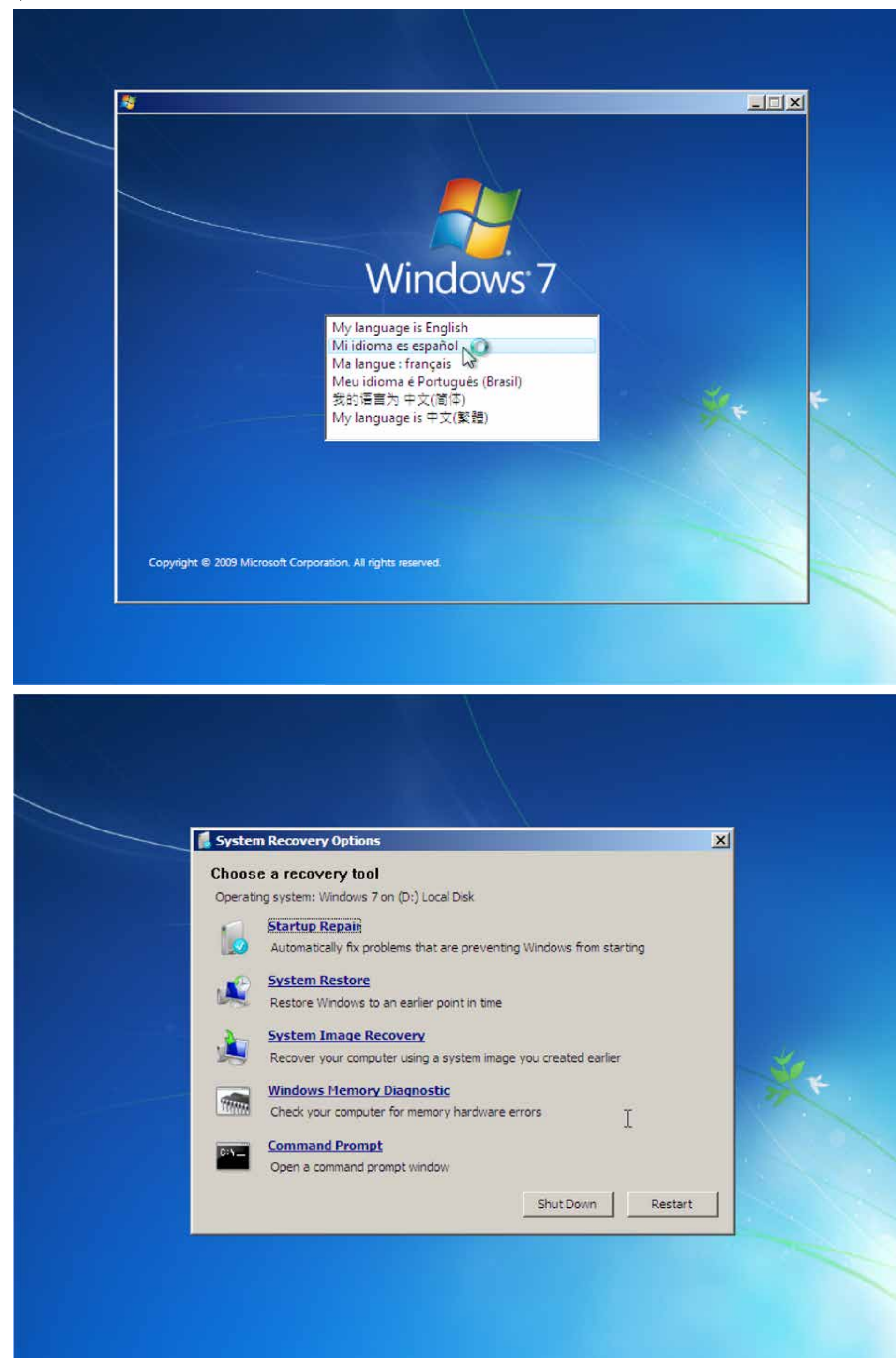


Including Windows 98 in this list is worth a mention, since it brought some improvements: a list of the overall steps, and a time estimate for the remaining process.⁵





I will spare you the pain of going through the installation of every version of Windows. With the simple addition of Windows 7, we can already extract some patterns regarding the usability of its installer:⁶



We could endlessly argue whether FreeBSD actually competes with Microsoft Windows. From its own web homepage, the FreeBSD project does have the ambition to target desktop systems. This is now evidenced by the Laptop Desktop Working Group (LDWG). But without entering this debate, I think we can only benefit from looking at the consumer mar-

ket, if only as a reference. The following patterns and criteria clearly emerged from this experience here:

Criteria	MS-DOS 6.22	Windows 3.11	Windows 95	Windows 98	Windows 7
Language support	MIX (different releases for each language)				YES
Accessibility	YES (high contrast)	NO	NO	NO	YES (magnifier, on-screen keyboard, high contrast)
License agreement	NO	NO	YES	YES	YES
Number of steps	9 (with 3 disks)	15 (first 5 disks)	21	17	10
Navigation	NO	NO	YES	YES	NO
Advanced mode	NO	MIX	YES	YES	YES
List of steps	NO	NO	NO	YES	MIX
Questions first	NO	NO	NO	NO	NO
Time estimate	NO	NO	NO	YES	YES
Live system	NO	NO	NO	NO	NO
Repair system	MIX	MIX	MIX	MIX	YES
Text mode	YES	NO	NO	NO	NO
Graphical mode	NO	MIX	MIX	MIX	YES
Graphical outcome	NO	YES	YES	YES	YES
Automation	NO	NO	NO	NO	YES

- If I had to summarize the strength of this series of installers, I would point out:
- Indicators for overall progress were often provided.
 - The time estimate, however inaccurate as it can be, can feel useful to the end user.
 - Only the most essential decisions were left up to the user (target disk and installation directory), with the technical decisions left out by default:
 - Choice of filesystem technology,
 - List of drivers to install,
 - Desktop environment, (evidently)
 - Pre-installed list of applications. Alternatively, there was a clear way into an advanced mode.
 - It was often possible to revert a decision and come back to the previous menu; curiously, Windows 7 dropped this possibility, but it was the most trivial version to install, with only 10 steps.
 - The final system was directly usable for desktop use.

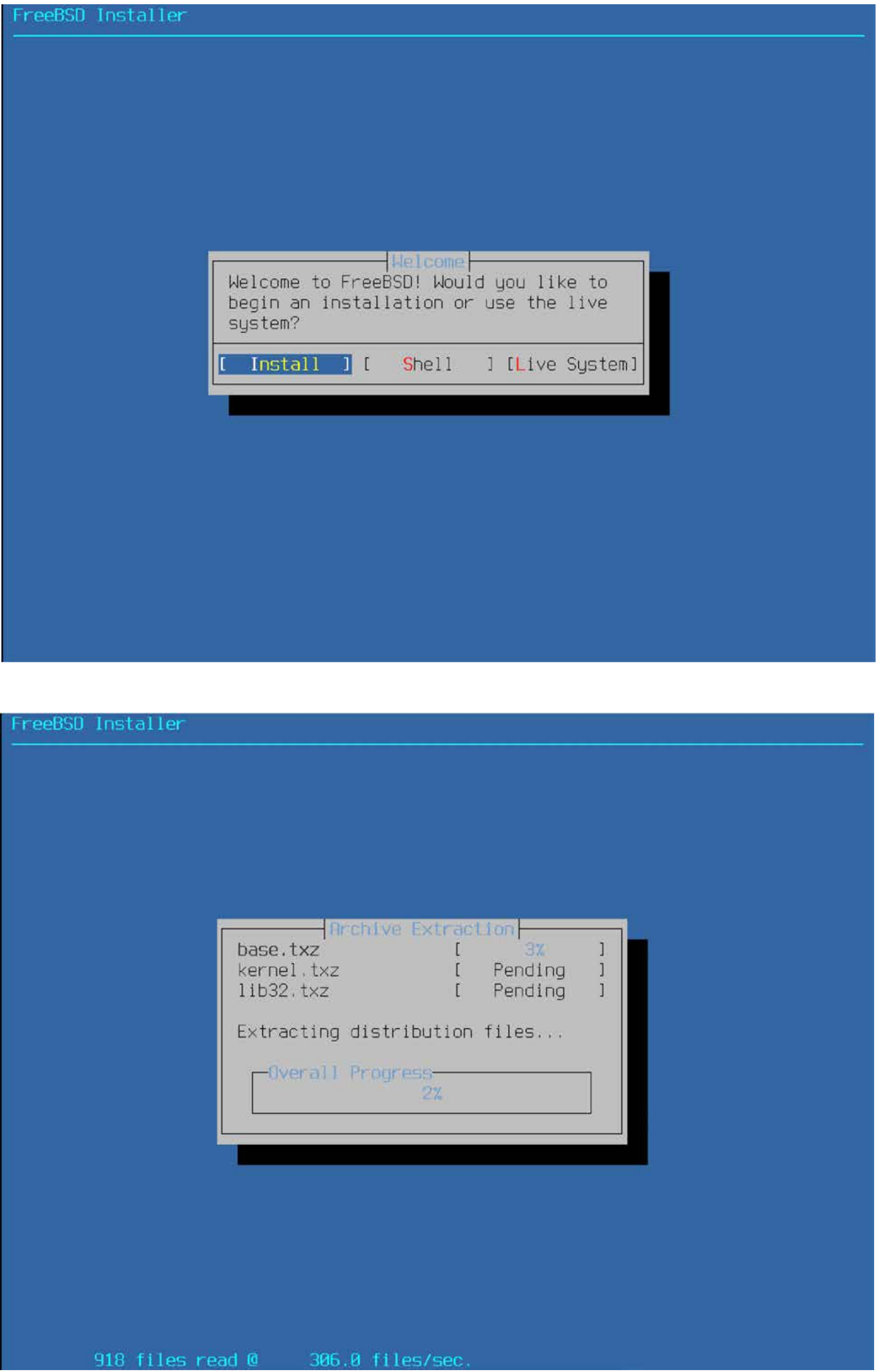
Although three major versions of Windows have been released since Windows 7, I am not aware of significant changes to the outcome in terms of usability.

FreeBSD as of today

The latest release while writing these lines is version 14.2.^{7,8}

After a fresh installation of Windows 7, if only looking at the usability aspect, the FreeBSD installer is certainly dated and left behind.

So how does
FreeBSD fare
when facing its
closer competition?



Criteria	FreeBSD 14.2
Language support	NO
Accessibility	NO
License agreement	NO
Number of steps	23
Navigation	NO
Questions first	NO
Advanced mode	YES
List of steps	NO
Time estimate	NO
Live system	MIX (CLI)
Repair system	MIX (CLI)
Text mode	YES
Graphical mode	NO
Graphical outcome	NO
Automation	YES

However, the comparison is not necessarily fair: Microsoft Windows clearly targets non-technical users, who are expecting a graphical interface, in a one-size-fits-all ap-

proach, and offers customer support.
So how does FreeBSD fare when facing its closer competition?

Linux Ubuntu (server)

Ubuntu offers different installation images, depending on the purpose of the installation. I have used Ubuntu Server in its version 24.04.02 LTS for the initial comparison.⁹



Looking at this installer, at first glance it is much closer to what FreeBSD offers! The system is kept in text mode throughout the process, including the user login prompt concluding the first boot. This is certainly more on par with the FreeBSD installer. But other than that, the process felt simpler and more straightforward.

Criteria	Ubuntu (server)
Language support	YES
Accessibility	NO
License agreement	NO
Number of steps	15
Navigation	YES
Questions first	YES
Advanced mode	YES
List of steps	NO
Time estimate	NO
Live system	MIX (CLI)
Repair system	MIX (CLI)
Text mode	YES

Graphical mode	NO
Graphical outcome	NO
Automation	YES

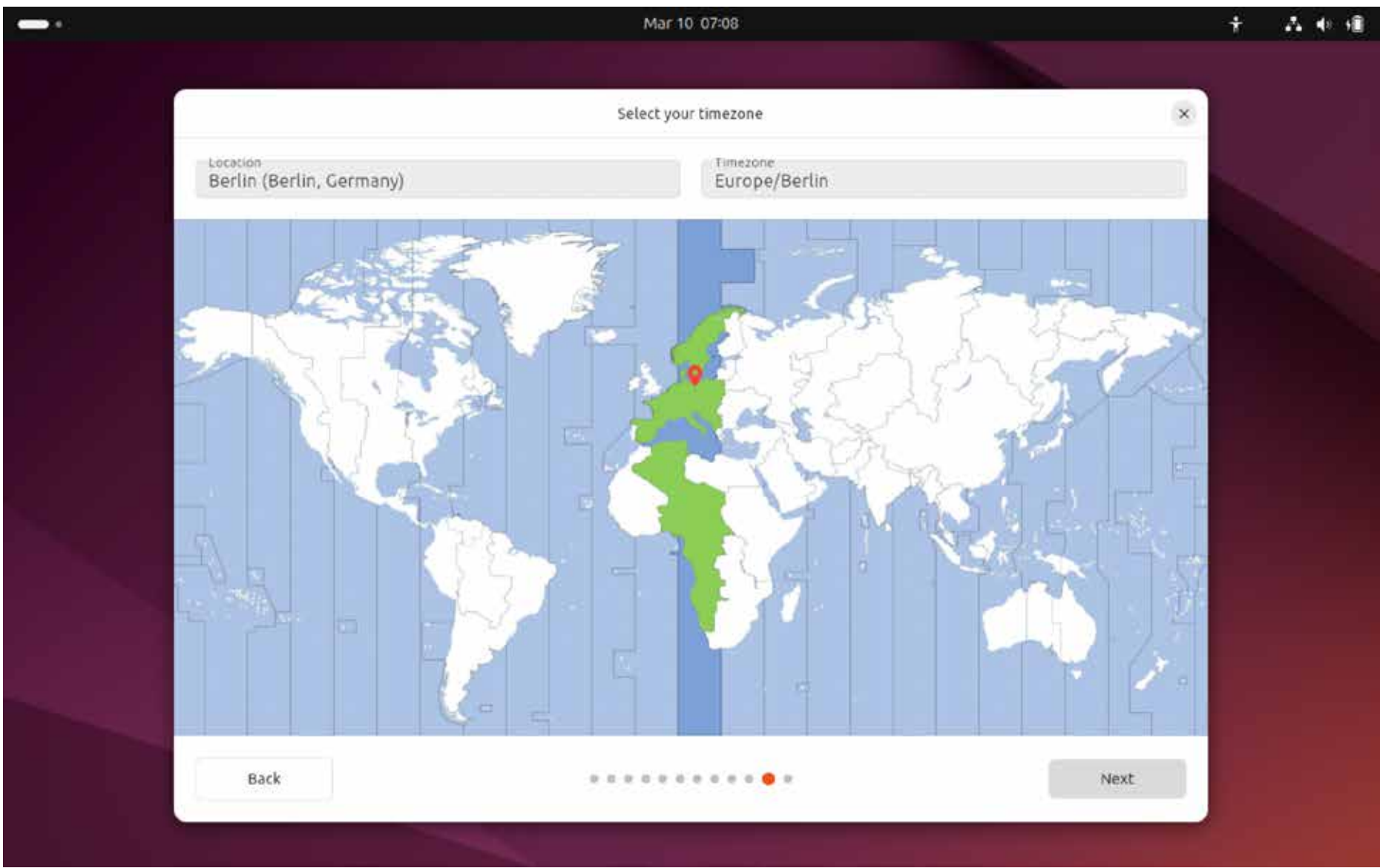
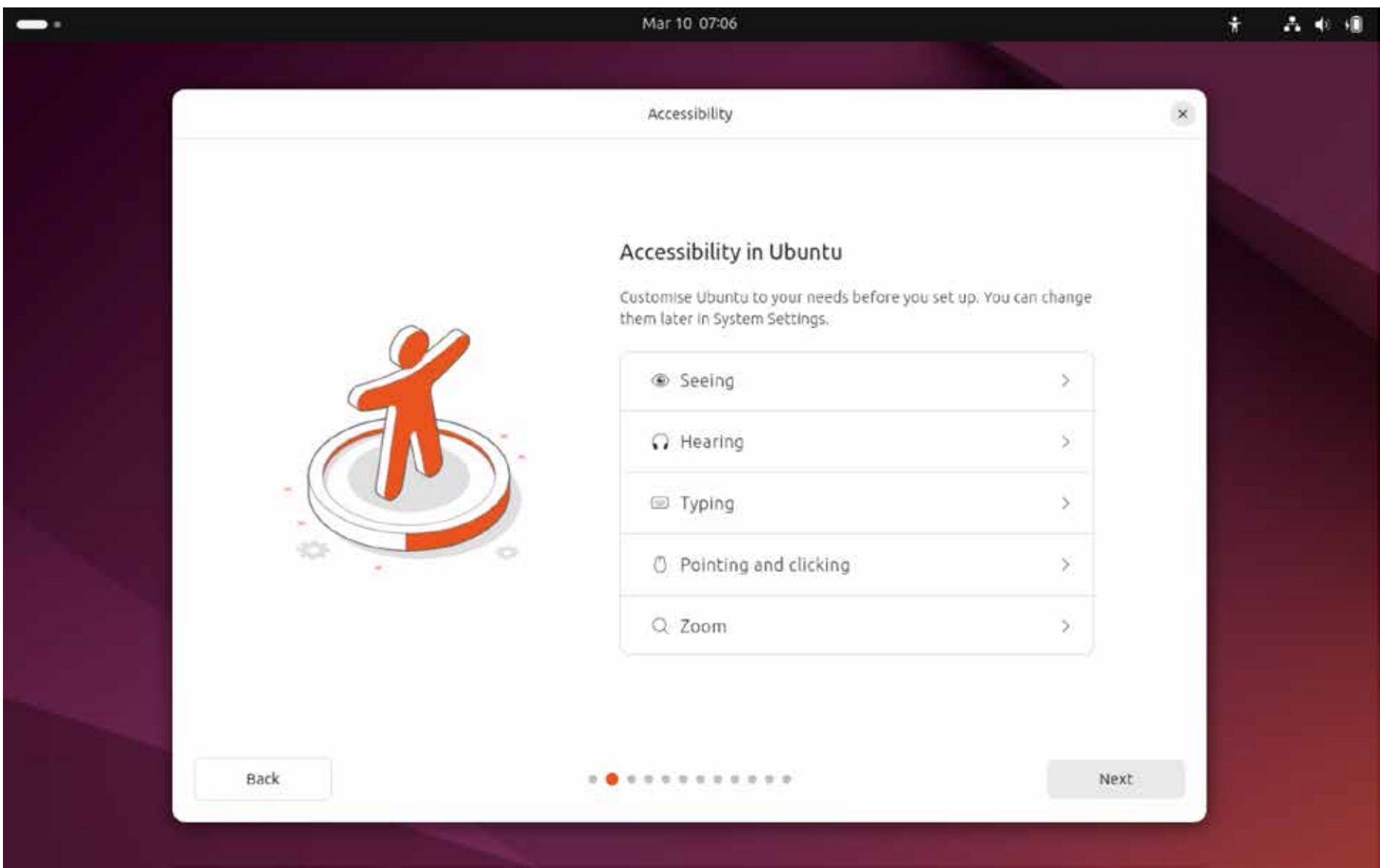
Definitely a closer match to FreeBSD's own offering. I would identify the following aspects as key improvements over FreeBSD however:

- Support for multiple languages in the installer.
- Questions asked in a batch, the actual installation performed in one go.
- Installation purposes available during the installation (snaps).

Linux Ubuntu (desktop)

Next, I have tried Ubuntu Desktop, also in version 24.04.02 LTS.¹⁰

As expected, this version provides a graphical user experience. But more than that, the overall level of quality and polishing was — in my own perception anyway — on a different level.



Criteria	Ubuntu (desktop)
Language support	YES
Accessibility	YES
License agreement	NO
Number of steps	15
Navigation	YES
Questions first	YES

Advanced mode	YES
List of steps	NO
Time estimate	NO
Live system	YES
Repair system	MIX
Text mode	NO
Graphical mode	YES
Graphical outcome	YES
Automation	YES

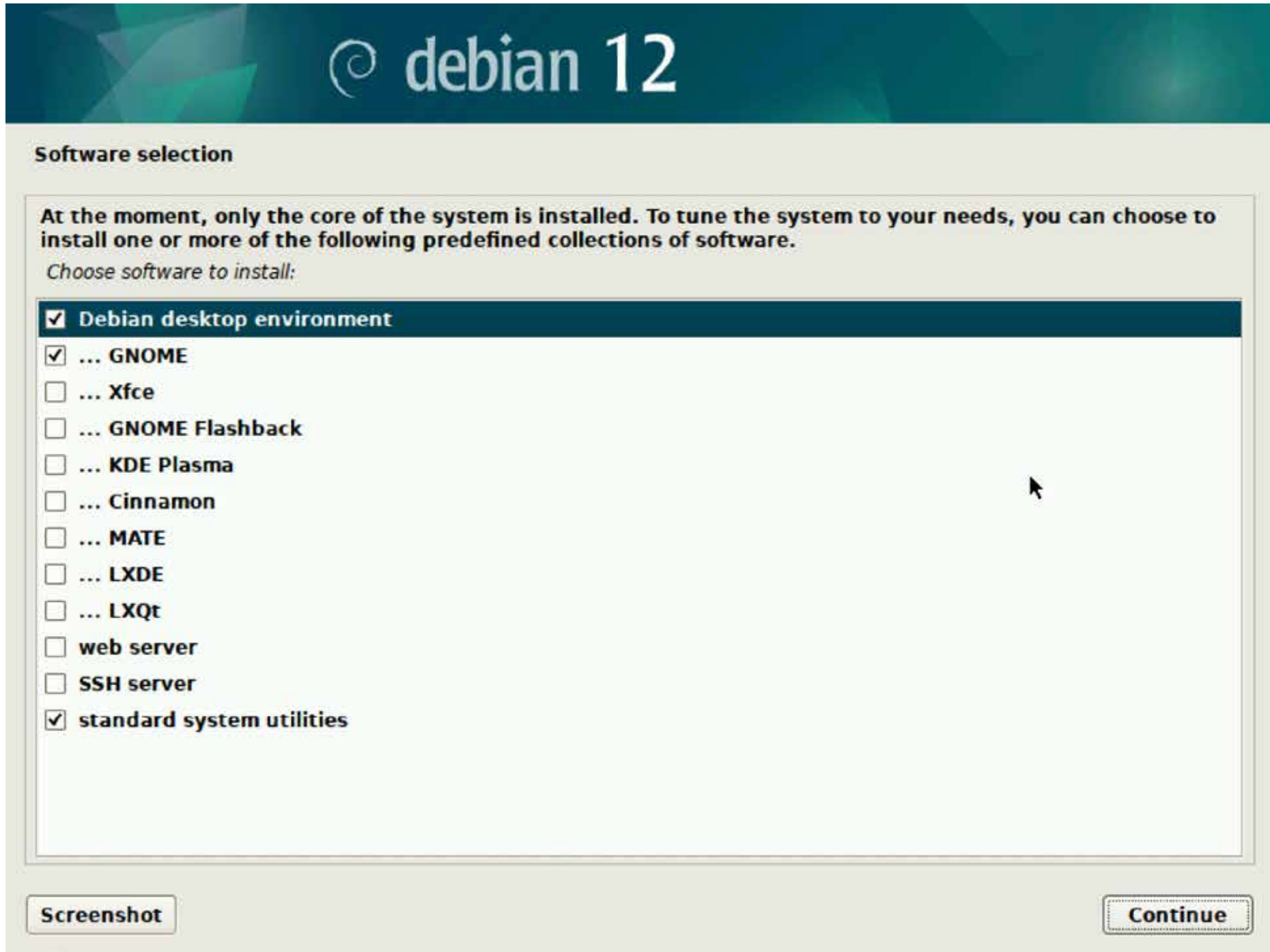
In terms of usability, this is definitely a reference. As for Microsoft Windows, this can be expected from a solution with commercial support. But it would be more fair to compare FreeBSD to another community project. For this reason, before diving deeper into FreeBSD’s own installer, let’s see if we can learn from Ubuntu’s parent, non-commercial distribution: Debian.

Debian GNU/Linux

Unlike Ubuntu, this last comparison will be a two-in-one: it turns out that the netinstall version of Debian’s installer, at only 632 MB, features both a text-based installer and its graphical equivalent.¹¹

On another personal note: before switching mainly to BSD systems, my longtime UNIX-like distribution of choice was Debian. After the pleasant surprise of the hybrid approach, I ran into a frustrating repeat of this past experience: the installation process asks a seemingly endless list of questions, and a lot of time-consuming activity often takes place between each set.





Criteria	Debian GNU/Linux
Language support	YES
Accessibility	NO
License agreement	NO
Number of steps	26
Navigation	MIX
Questions first	NO
Advanced mode	NO
List of steps	NO
Time estimate	MIX
Live system	YES (other image)
Repair system	MIX (CLI)
Text mode	YES
Graphical mode	YES
Graphical outcome	YES
Automation	YES

In spite of offering a graphical mode, the visual appearance was rather crude. I was also surprised to not find an accessibility mode beyond the high contrast mode in the boot menu; I suspect I may have simply missed it.

On a very positive note however, it was very easy to obtain an installation suitable for either server or desktop use. This matches perfectly with the hybrid approach for the installer.

Can FreeBSD achieve a similar outcome?

Special mention: macOS

Last but not least, I would like to mention macOS. While also originally based on a BSD system, macOS can rely on a database of well-known, in-house hardware targets. The system firmware is shipped with a number of fallback mechanisms in order to (re-)install the system in different ways, including without installation media: the whole system can be pulled from the Internet by the firmware.

This is impressive, and on par with the commercial offerings above; but again, this is unfair to our beloved FreeBSD system. However, I would like to insist on the availability of enough basic tools to allow for analysis and repair operations; this can be a life-saver in unexpected situations.

Diving into FreeBSD

From my own experience looking at its source code, the current situation with FreeBSD's installer is a direct consequence of some design decisions. I would like to emphasise that this is not pure criticism of its implementation, or of the decisions made at the time; the point is to understand where we are, if the installer is ripe for a rewrite, or if some battles can be fought (and won!) with easy steps.

Architecture

Most of the FreeBSD installer is written as a combination of shell scripts, with a few steps implemented as dedicated C programs. The entire source code can be found in the `src.git` repository, but under the hood, it really consists of three main components:

- **bsdconfig**, under **usr.sbin/bsdconfig**, which is used as a library of shell routines by **bsdinstall** and its set of scripts.
- **bsddialog**, under **usr.bin/bsddialog**, is the tool interacting with the user, both for obtaining input and for monitoring the activity.
- Finally, **bsdinstall** itself is under **usr.sbin/bsdinstall**.

When starting the FreeBSD installation image, a dedicated `rc.local` startup script (from `release/rc.local`) concludes the boot process by running **startbsdinstall**. This script offers the welcome screen, as well as the possibility to use the installation media as a live system. By default, the live mode simply spawns a shell, which is known territory for regular UNIX users, but does not really qualify as a live or repair system for the average user.

If we focus on the installation process, the procedure is performed as follows:

- **startbsdinstall** (see above).
- **bsdinstall** can be called directly with a specific operation, as found in `/usr/libexec/bsdinstall`, and defaults to `auto` (our choice here).
- **auto** performs a series of operations:
 - Optionally, **local.pre-everything** if present.
 - **keymap** lists different possible keyboard layout.
 - **hostname** sets the hostname for the target system.
 - Selection of optional system components, as per the **MANIFEST** file (which should be at least **base.txz** and **kernel.txz**).

If any system component is missing from the installation media:

- **netconfig**
 - (Optionally, **local.pre-partition** if present).
 - Application of known quirks depending on the platform detected (e.g., via SMBIOS or boot architecture).
 - Choice of partitioning scheme (UFS automated, manual setup, or ZFS if supported).
 - Application of the partitioning scheme and mounting of the target filesystem.
 - (Optionally, **local.pre-fetch** if present).
 - If any system component was found to be missing from the installation media:
 - **fetchmissingdists** (saved within the target volume in `/usr/freebsd-dist`).
 - checksum and `distextract` verify and extract the system components.
 - **bootconfig** configures the bootloader.
 - (Optionally, **local.pre-configure** if present).
 - `rootpass` prompts for the root password.
 - If the network hasn't been setup yet, **netconfig** configures the network.

- A series of questions:
 - **time** sets the date, time, and timezone.
 - **services** offers a selection of services to be started at boot.
 - **hardening** offers some hardening measures to be applied.
 - **firmware** allows the installation of firmware packages.
 - **adduser** allows adding user accounts, but calls **adduser(8)** with **chroot(8)** to do so, which surprisingly alters the look & feel of this operation.
- A not-quite final menu, **finalconfig**, allows further changes before restarting.
- A non-interactive step, **config**, actually applies the configuration values chosen above.
- Cleanup of any system component downloaded.
- (Optionally, **local.post-configure** if present).
- One last final menu, lets the user start a shell within the target system (with **chroot(8)**)
- A non-interactive security step saves entropy.
- Finally, **umount** allows for a safe reboot.
- Back to **startbsdinstall**, if **bsdinstall(8)** was successful the user is greeted with a comforting message, and offered to reboot or shutdown the system, or to spawn a shell.

As a first, positive note, we can see that the installation process behaves well even when missing the installation files for the system components. This allows installation media to be generated in different sizes, without having to modify the installation logic.

On another hand, this is an awful lot of questions for the user. They may feel essential and useful in isolation, but most of the systems described above deliver functional systems with a third of the interactive steps. Unfortunately, this is not the only problem identified with the user interface.

Limitations and use cases

Heavy reliance on **bsddialog(1)** (or **libbsddialog** for the C programs) introduces limitations similar to those of e.g., the Debian installer: the user interface for each step is rather basic, and does not support dynamic interaction according to the user input. As an example, the network configuration step will not be able to suggest a suitable network mask, DNS server or gateway, according to the most common conventions. This could not only save some keystrokes and unnecessary efforts for the user, but would also feel more modern and polished overall.

Possible solution: Lua This specific aspect could be solved with the introduction of Lua scripting capability in **libbsddialog** or otherwise in **bsddialog(1)**. This has been suggested by the author of **bsddialog**, Alfonso Siciliano (asiciliano@), who is currently considering implementing this extension into the project.

Beginner users An easy win for beginners, or generally for desktop users, would be the possibility to conveniently install a desktop system, ready and functional upon the first reboot. While introducing an additional step, it could be inspired by the Debian installer (as illustrated above) and fit both server and desktop use.

There is ongoing experimentation in this direction (notably by Alfonso) but this is easier written in an article, than implemented reliably. Some of the challenges involve the installation of the corresponding firmware files, drivers, including a particular pain point with the DRM kernel modules. There is an ongoing effort (hi manu@ and thanks!) to move the lat-

ter from the ports back into the base system, which should help with recurring version mismatch issues in the current binary ports.

Enterprise use On a positive note this time, the FreeBSD installer can be tweaked and automated for deployment typical of enterprise use.

First, it is capable by default to set up jails instead of performing regular installations. This behaviour can be selected with the jail target, instead of the default auto described above.

The operation then looks as follows:

```
.....
# bsdinstall jail /path/to/the/new/jail
.....
```

Should the FreeBSD installer be re-implemented or modified extensively, it might be important to keep this functionality in mind. On the other hand, numerous jail management solutions already exist outside of **bsdinstall(8)**.

More importantly, the installer also supports a script target, which then requires an installation script to be executed in the context of **bsdinstall**. This has to be provided by the system administrator but allows for fully automated installations with the FreeBSD installer.

I have to wrap up this section with a missing feature, observed in all of the competition above, and documented in Bugzilla's #214390: the installer assumes no proxy is necessary to reach the Internet. In my own experience penetration-testing deployment environments, on premise at dozens of different customers, this limitation was usually disqualifying.

Development environment

To be fair, it can be rather difficult to work on the implementation of **bsdinstall(8)**. Testing some changes often requires building a full bootable image, running it into a virtual environment, and repeating installation steps until reaching the corresponding situation. In order to be more efficient when working on the installer, I have experimented with a development environment suitable for this purpose.

It consists of a PXE-based setup, which can be tweaked for traditional PXE environments as well as more modern, UEFI-assisted network boots. I had to combine this boot sequence with a TFTP server (managed by inetd), a DHCP service (with the **isc-dhcp44-server** package), and a read-only NFS mount. Describing the entire setup goes beyond the scope of this article, but the usability of the FreeBSD installer depends on that of its development environment after all. Feel free to adapt the following snippets to your own taste.

For **/etc/exports**:

```
.....
/jail/bsdinstall -ro -network 192.168.x.0 -mask 255.255.255.0 -maproot 0:0
.....
```

For **/etc/inetd.conf**:

```
.....
tftp dgram udp wait root /usr/libexec/tftpd tftpd -l -s /tftpboot
tftp dgram udp6 wait root /usr/libexec/tftpd tftpd -l -s /tftpboot
.....
```

For **/etc/rc.conf**:

```
.....
mountd_enable="YES"
nfs_server_enable="YES"
nfs_server_flags="-h 192.168.x.y"
rpcbind_enable="YES"
rpcbind_flags="-h 192.168.x.y"
.....
```

For `/usr/local/etc/dhcpd.conf`:

```
.....
option subnet-mask 255.255.255.0;
default-lease-time 600;
max-lease-time 7200;

subnet 192.168.x.0 netmask 255.255.255.0 {
    range 192.168.x.128 192.168.x.254;
    option routers 192.168.x.y;
    option subnet-mask 255.255.255.0;
}

option arch code 93 = unsigned integer 16;

host example {
    hardware ethernet aa:bb:cc:dd:ee:ff;
    fixed-address 192.168.x.127;
    next-server 192.168.x.y;
    if option arch = 0:07 {
        #UEFI
        filename "FreeBSD/boot/loader.efi";
    } else {
        #BIOS
        filename "FreeBSD/boot/pxeboot";
    }
    option root-path "192.168.x.y:/jail/bsdinstall/";
}
.....
```

Alternatives suggested
included GhostBSD,
MidnightBSD,
and NomadBSD.

The files `loader.efi` and `pxeboot` should be placed in the `/tftpboot/FreeBSD/boot` folder.

You may want to setup this `/jail/bsdinstall` folder as a typical jail, tweaking it to reflect the regular boot environment of the installer, or to expose the `disc1` folder from a release build instead. Either way, this should save you from generating, transferring, or rebooting image files when working on `bsdinstall`.

Note that this setup works equally well for physical hardware and as Virtual Machines, since hypervisors like VirtualBox support booting from PXE.

Graphical mode

Finally, it is time to answer the question: can we improve the FreeBSD installer, and bring it to feature parity with that of Debian for instance?

Background

In early 2024, I was tasked by the FreeBSD Foundation to look at the state of the art of graphical installers in the Open Source landscape. The underlying goal was to identify a possible approach for the FreeBSD project to add this feature. Calamares¹², a new installer framework for Linux, was an obvious candidate — if it could be confirmed to be usable with

FreeBSD as the target system. Unfortunately, its use of the GPL license was found to be incompatible with FreeBSD's target demographics of the base system.

Another alternative specific to FreeBSD, and with a lot of visibility at the time, was PC-BSD. After going through a first rebranding as TrueOS, it was discontinued in 2020. The alternatives suggested included GhostBSD, MidnightBSD, and NomadBSD.

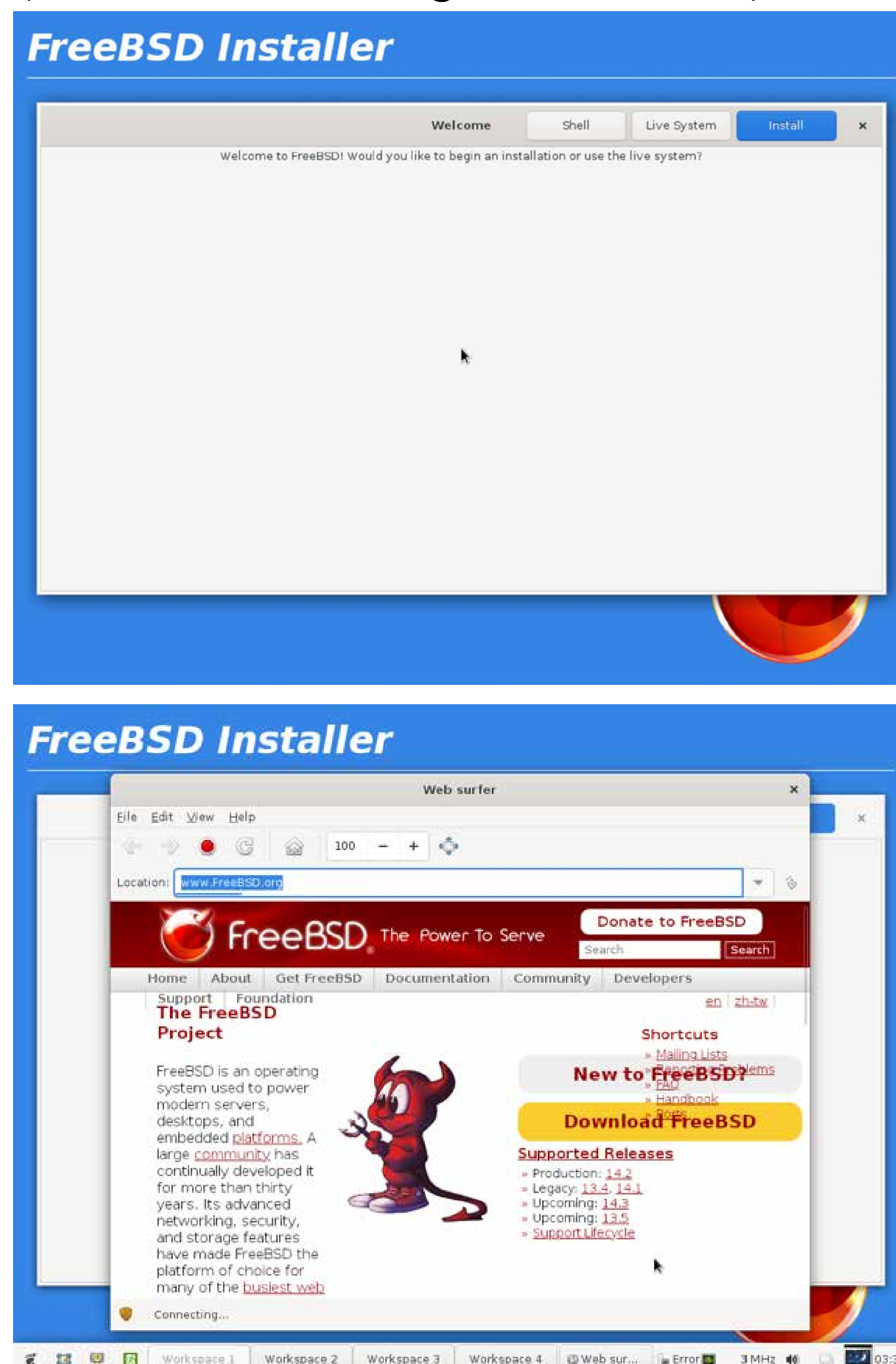
GhostBSD features a graphical installer, gbi. While a good candidate, it would significantly differ from the current codebase of bsdinstall, while also requiring a complete Python and Gtk+ environment. It seemed to me that a more efficient approach was possible.

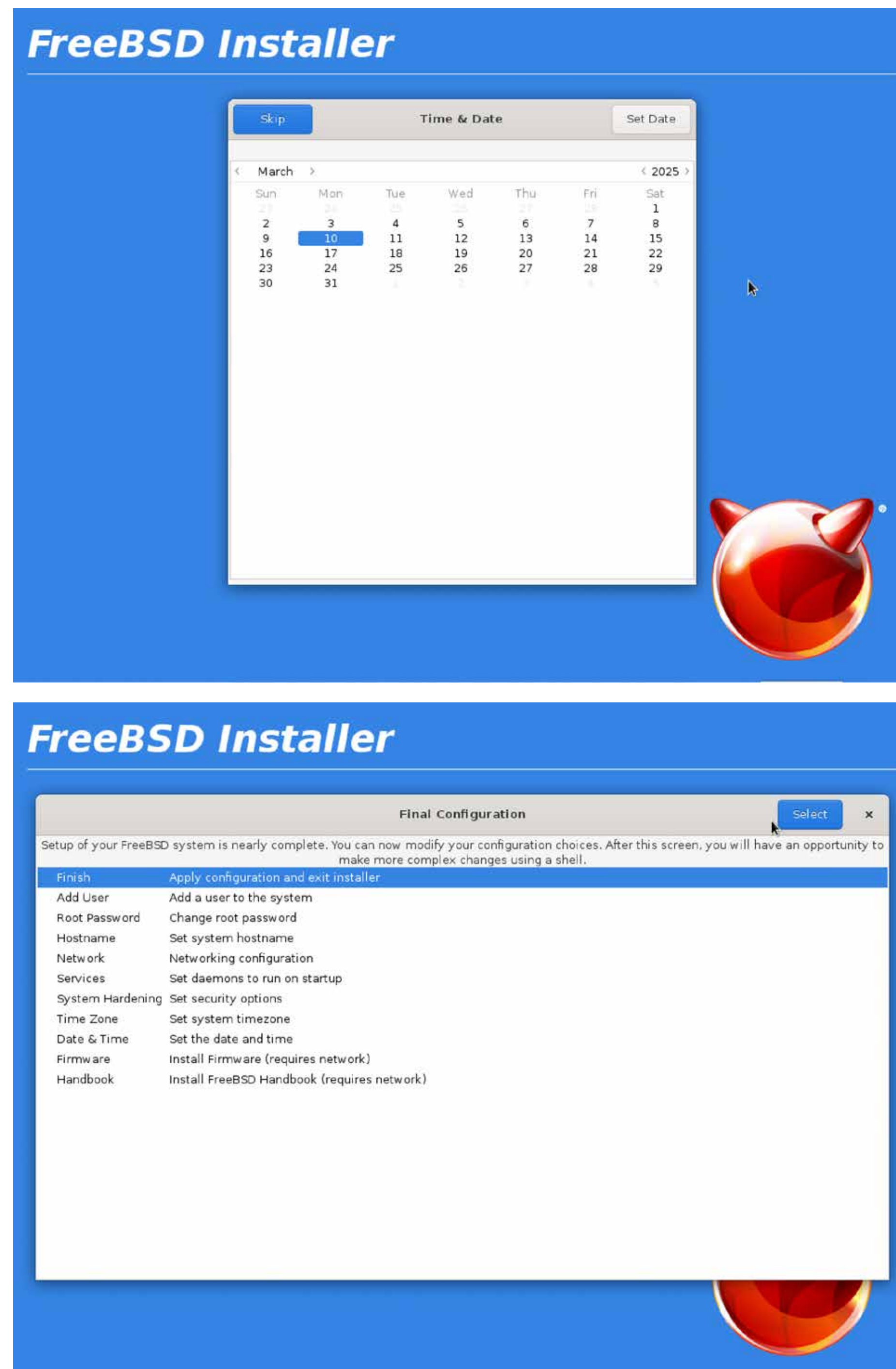
Neither MidnightBSD nor NomadBSD offer a graphical installer from what I could tell.

Least intrusive approach

Instead, being already familiar with **bsdinstall** and **bsddialog**, I could picture a way to re-use the existing code base and architecture, while performing the installation process with a graphical interface. By replacing **bsddialog(1)** with an equivalent tool, also BSD-licensed, it was possible to deliver a fully functional installation image within the few weeks assigned for the project.

Being already familiar with Gtk+, I considered it a reasonable trade-off if only to come up with a working demonstrator. Gtk+ is available under the LGPL license, which is still not enough a match for the target demographics. But I managed to implement **gbsddialog**, a desktop application equivalent to **bsddialog**, within a few days.





My initial vision for this tool was slightly different. I was hoping to apply additional work-arounds or improvements:

- Prevent the dialog window from disappearing and reappearing each time between calls, for instance by leveraging the GtkPlug and GtkSocket widget.
- Using the GtkAssistant widget for a better look & feel, once the previous item addressed.
- A specific widget from **bsddialog** is implemented in a way that does not suit a graphical implementation of the tool: **mixedgauge**. In **bsdinstall**, the persistence of the console output of **bsddialog** is abused in order to give the illusion of progress, while in reality a new instance of **bsddialog** is executed for each update. In the graphical installer, this monitoring had to be replaced with a regular **gauge** widget, at the expense of usability.

On the other hand, I am quite happy with the implementation of the desktop window, reproducing the look & feel of the original **bsddialog** tool quite accurately in its graphical version. This part of the code is not very elegant as of now, but it does the job.

Repair and upgrade capability

I had the pleasure to co-mentor a Google Summer of Code project last year (hi Li-Wen!) with a very proactive student: Leaf Yen. The project was about bringing new capabilities to the FreeBSD installer: upgrading or repairing existing systems, with removable media.

The project concluded successfully, and was submitted as three pull-up requests on GitHub:

- #1395, GSoC 2024 — Improving Installer with Repair and Upgrade Ability.
- #1424, bsdinstall: Add pkg install support in live env.
- #1427, bsdinstall: Add repair scripts to installer menu.

I have not managed to integrate this work into the graphical version of the installer, but I consider it an equally important part of a modern installer. Consider this article a call for attention on this project too, which should be able to work equally well with the current installer and the graphical version once polished and merged.

Graphical installer as web application

This article would not be complete without this possible added bonus. Since Gtk+ 3, it is possible to render Gtk+ applications inside a web browser instead of a computer screen. This is implemented by the GDK Broadway backend.

While I haven't experimented with this capability yet, it seems possible to turn the graphical version of the installer into a web server. This would allow the installation of FreeBSD systems over the network, without any additional change! This is clearly another potential easy win.

Accessibility

I would like to mention an additional criterion for the adoption of Gtk+ for the graphical installer demonstrator: accessibility.

While I am subject to my own set of impairments myself, they are not severe enough for me to have experimented much with Gtk+' set of accessibility features. However, I was aware of its capabilities in this regard, and they are also relevant to the usability of an Operating System installer for the part of the population affected. In fact, I have heard of requests for accessibility support in the installer, either privately or at conferences.

There is even a funded project aiming at bringing accessibility features to the FreeBSD installer, under investigation by Alfonso Siciliano. (Hello again!)

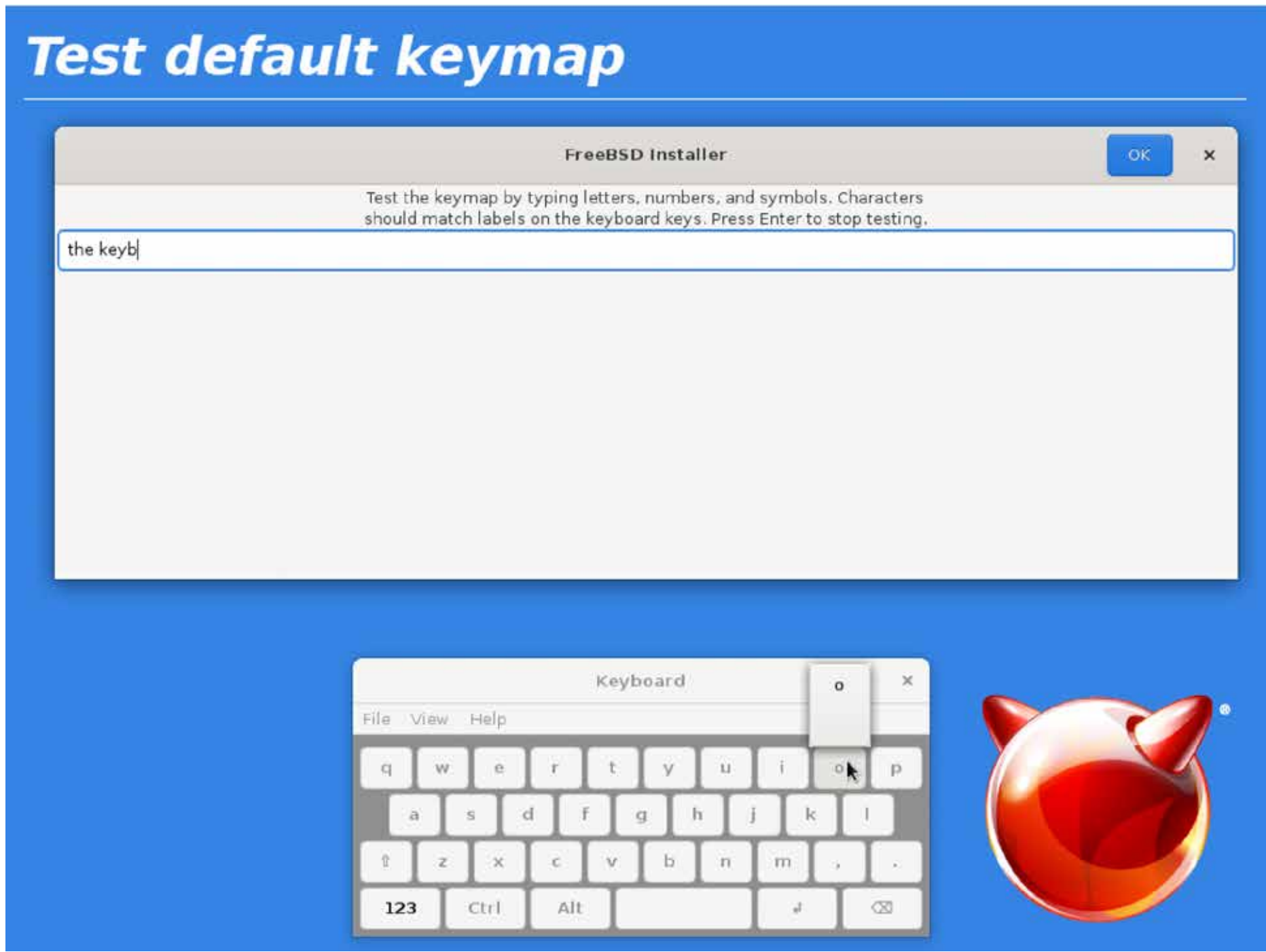
Back to Gtk+, it supports the following requirements for accessibility:

- Screen readers, like Orca for conversion to speech or braille.
- Full keyboard navigation.
- Accessible behaviour in the standard widgets.
- Theming for high contrast or further visibility improvements.
- Text and interface scaling.
- Plug-in system for further improvements.

I have not had the opportunity to try out these features with someone affected, but I have kept this aspect in mind throughout the development of **gbsddialog**, sticking to the core Gtk+ API everywhere. As a result, the tool is compatible with both Gtk+ 2 and Gtk+ 3.

Before concluding this article, I am going to mention the DeforaOS desktop environment. It is another lightweight desktop environment, written by a single individual for the most part (hi!) and to be honest, it shows. It is nowhere near the level of completion and quality that I am aiming for. It is partly packaged in FreeBSD's ports (thanks Olivier!) and I have used it as part of the demonstrator for the graphical installer here — if only to illustrate the capabilities.

Still, I am glad to mention that it features a virtual keyboard program, and that I could successfully deploy it in combination with the graphical version of the installer.



This allows me to update FreeBSD's feature table as follows, considering the progress documented in this article:

Criteria	FreeBSD 14.2
Language support	NO
Accessibility	YES (magnifier, on-screen keyboard, high contrast)
License agreement	NO
Number of steps	22
Navigation	NO
Questions first	NO
Advanced mode	YES
List of steps	NO
Time estimate	NO
Live system	YES
Repair system	YES
Text mode	YES
Graphical mode	YES
Graphical outcome	MIX (ongoing)
Automation	YES

Conclusion

This can be a very divisive and subjective subject to tackle. I hope that this article will be helpful, so that the installation process can be improved for the next releases of FreeBSD. Unfortunately, working on the current code base is not always easy, and can require a lot of patience when it involves generating and booting installation media for each and every test. Simple changes can really harm hardware support or the overall quality of the user experience in unintended ways.

Evidently, there is ample room for improvement, and during my work on the installer, I have heard calls for a complete rewrite. It is not an easy decision to make, nor an easy project to implement — especially while addressing every possible use case, trying to please everyone, and keeping it within an adequate budget.

Quite frankly I do not feel like I am in a position to judge or impose a specific decision over another. I had hoped to be able to offer and maintain a simple evolution of the cur-

rent installer myself, extending its current capabilities without bringing unnecessary changes. By reusing its code as described here, bugs fixed and improvements made to the console-based installer are reflected in the graphical version, and vice versa.

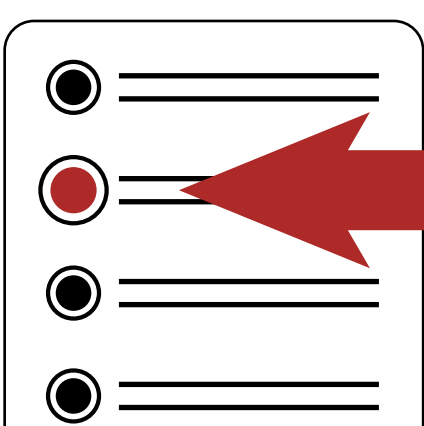
References

1. https://archive.org/details/MS_DOS_6.22_MICROSOFT
2. <https://archive.org/details/windows-for-workgroups>
3. <https://forums.virtualbox.org/viewtopic.php?t=102121>
4. <https://archive.org/details/win-95-osr-2>
5. <https://archive.org/details/windows-98-se-isofile>
6. https://archive.org/details/en_windows_7_professional_with_sp1_x64_dvd_u_676939_202302
7. <https://download.freebsd.org/releases/amd64/amd64/ISO-IMAGES/14.2/> (amd64)
8. <https://download.freebsd.org/releases/arm64/aarch64/ISO-IMAGES/14.2/> (aarch64)
9. <https://ubuntu.com/download/server>
10. <https://ubuntu.com/download/desktop>
11. <https://cdimage.debian.org/debian-cd/current/amd64/iso-cd/> (amd64)
12. <https://calamares.io>

PIERRE PRONCHERY is a Security Engineer for the FreeBSD Foundation and has been a member of their team as a Userland Software Developer since May 2023. He installed his first FreeBSD system in 2001 and has a passion for operating system design and implementation.

Write For Us!

Contact Jim Maurer
with your article ideas.
(maurer.jim@gmail.com)



Overlord: Deploy Jails as Fast as You Code

BY JESÚS DANIEL COLMENARES OVIEDO

When [AppJail](#), a BSD-3 licensed open source framework entirely written in `sh(1)` and C to create isolated, portable and easy to deploy environments using FreeBSD jails that behave like an application, was created, my intention was to test ports so as not to mess up my main environment. Today AppJail is more than just a script for testing ports, it is highly flexible and has some very useful automation features.

After AppJail reached stability and had been used on a variety of systems, I realized that just deploying a jail for every service I wanted was not feasible, especially when more and more services need to be deployed. Thus Director was born.

[AppJail Director](#) is a tool for running multi-jail environments on AppJail using a simple YAML specification. A Director file is used to define how one or more jails that make up your application are configured. Once you have a Director file, you can create and start your application with a single command: **appjail-director up**.

AppJail is more than just
a script for testing ports.

Director is the first attempt to follow the "everything is code" philosophy related to AppJail. Director organizes jails into projects, so you create a project with one or more jails declaratively, and Director takes into account any changes you have made to that file or a related file (such as the Makejail used). If Director has seen a change, it doesn't hesitate to destroy your jail to recreate it. This sounds a bit crazy, but is best explained in *The Ephemeral Concept*:

Director treats each jail as ephemeral. This does not mean that your jails will not persist after you stop them or restart your system, what it means is that Director assumes that it is safe to destroy the jails since you have clearly separated the data that should be persisted from the data considered ephemeral.

There are more details in the **appjail-ephemeral(7)** man page, but the principle is the same as above.

Director by itself does not deploy jails, it needs instructions that perform configuration, package installation, etc., so it heavily exploits a feature of AppJail called Makejails, a simple text file that automates the steps of creating a jail. There are many created in the [Centralized Repository](#), but nothing prevents you from using your own repository to host your Makejails.

Both AppJail and Director simplify my life a lot, however you have to deal with a problem that neither AppJail nor Director solve, which is orchestrating jails on many servers. AppJail and Director combined with SSH may be workable for a few servers, but when you have more and more, this is just painful. Thus Overlord was born.

[Overlord](#) is a fast, distributed orchestrator for FreeBSD jails oriented to GitOps. You define a file with the service intended to run on your cluster and deployment takes seconds to minutes.

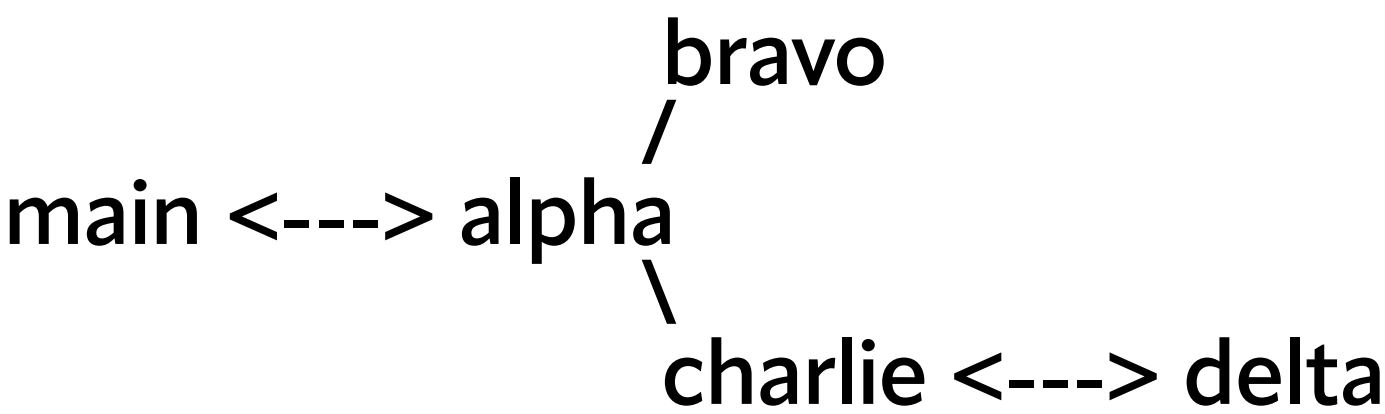
Fortunately for Overlord (and for me), AppJail and Director are mature, so it's a smart move to reuse those extensively tested tools and combine them with Overlord. This is what I have done and borrowing the same philosophy from Director that "everything is code" is why the orchestrator is easy to use. Another decision I have made is that everything in Overlord is asynchronous. Deployments can take a long time when the service is huge, but even when the deployment only takes a little while, it's much better to send instructions declaratively and let Overlord handle our work. In this article, in more detail, we'll look at many of the things I've said here.

Architecture

The Overlord architecture is described as a tree chain architecture. Each Overlord instance running the API server can be configured to group other chains. Each member can be further configured to group more chains together. However, although this step can be performed almost infinitely, doing this without caution can introduce latency, so it is important to know how you plan to organize your servers.

The reason for choosing this architecture is because it is very simple and scales very well, so sharing resources among many servers is just sticking each chain together forming a cluster to deploy projects.

This architecture also abstracts the way projects are deployed. A user who wants to deploy a project does not need to know the endpoint of each chain, only the first one (also known as the root chain) is enough. This is because each chain is tagged with an arbitrary string, so a user only needs to specify in his deployment file the endpoint of the root chain, the access token and the labels. Although labels are subjective, this can represent a desire. For example, we can label servers with the string `vm-only` for those servers that have the ability to deploy virtual machines or `db-only` for databases, it is really very arbitrary.



Assume that only charlie and delta have the `db-only` label. To deploy projects to the API servers with the specified labels, the client must make an HTTP request to main, specifying the chain `alpha.charlie` and `alpha.charlie.delta`. This is done transparently and does not require user intervention.

Fortunately for Overlord,
AppJail and Director
are mature

```
main . alpha . charlie
      &
main . alpha . charlie . delta
```

Smart Timeouts

What happens if a chain is down? If the root chain is down, we can do nothing, although we can specify more than one (although in the rest of this document we will only use one). However, if a chain after the root chain is down something interesting happens.

When a chain positioned after the root chain is down, the chain that detects the error can blacklist the failed chain for a while. The blacklist is used to not display failed chains, although it is not forbidden to attempt to connect to a failed chain through another chain, this is because successfully connecting to a blacklisted chain automatically re-enables it.

Each blacklisted chain has an assigned time to remain on the blacklist. After a period of time, the chain is removed from the blacklist, however, if the chain continues to fail, it is added back to the blacklist but for a longer time. This time increment has a limit so as not to disable a chain forever.

Doing the above has the good effect that an HTTP request can decrease the time to complete since there is no need to connect to a failed chain.

Each blacklisted chain
has an assigned time to
remain on the blacklist.

Deploying Projects

The best way to demonstrate Overlord is to deploy a small project.

Note that a project can have more than one jail, however, in the following example only one jail is needed.

filebrowser.yml:

```
kind: directorProject
datacenters:
  main:
    entrypoint: !ENV '${ENTRYPOINT}'
    access_token: !ENV '${TOKEN}'
deployIn:
  labels:
    - desktop
projectName: filebrowser
projectFile: |
  options:
    - virtualnet: ':<random> default'
    - nat:
services:
  filebrowser:
    makejail: 'gh+AppJail-makejails/filebrowser'
```

```

volumes:
  - db: filebrowser-db
  - log: filebrowser-log
  - www: filebrowser-www
start-environment:
  - FB_NOAUTH: 1
arguments:
  - filebrowser_tag: 14.2
options:
  - expose: '8432:8080 ext_if:tailscale0 on_if:tailscale0'
default_volume_type: '<volumeefs>'
volumes:
  db:
    device: /var/appjail-volumes/filebrowser/db
  log:
    device: /var/appjail-volumes/filebrowser/log
  www:
    device: /var/appjail-volumes/filebrowser/www

```

You have noticed that I'm not specifying the access token and entry point explicitly, but through environment variables, which are loaded through the `.env` file:

`.env`:

```

ENTRYPOINT=http://127.0.0.1:8888
TOKEN=<access token>

```

And now another question: how is the access token generated? This is easy, the token is generated from the machine running the Overlord instance you want to contact, however only who has the privileges to access the secret key (which is generated pseudo-randomly by default) can generate tokens.

```
# OVERLORD_CONFIG=/usr/local/etc/overlord.yml overlord gen-token
```

The next step is simply to apply the deployment file.

```
$ overlord apply -f filebrowser.yml
```

If there is no output, everything is fine, however this does not mean that the project is deployed. When a deployment file is applied and contains the specification to deploy a project (in the above case a **directorProject**), it is queued waiting for its turn. However, since there are no other projects currently running, our project will be deployed as fast as possible.

```

$ overlord get-info -f filebrowser.yml -t projects --filter-per-project
datacenter: http://127.0.0.1:8888
entrypoint: main
chain: None
labels:
  - all
  - desktop
  - services
  - vm-only

```

```

projects:
  filebrowser:
    state: DONE
    last_log: 2025-04-22_17h57m45s
    locked: False
    services:
      - {'name': 'filebrowser', 'status': 0, 'jail': 'e969b06736'}
    up:
      operation: COMPLETED
      output:
        rc: 0
        stdout: {'errlevel': 0, 'message': None, 'failed': []}
      last_update: 7 minutes and 42.41 seconds
      job_id: 14
      restarted: False
      labels:
        error: False
        message: None

```

Metadata

Metadata is used to create small files (such as configuration files) that can be used when deploying projects or VMs. While a git hosting such as GitLab, GitHub, Gitea, etc., is very useful in combination with Makejails, you can use Metadata instead of relying on a git hosting to further configure the service or VM you are deploying.

The other advantage of metadata is that it can be shared between different deployments. For example, by deploying virtual machines that share the same `sshd_config(5)` and `authorized_keys` files.

tor.yml:

```

kind: directorProject
datacenters:
  main:
    entrypoint: !ENV '${ENTRYPOINT}'
    access_token: !ENV '${TOKEN}'
deployIn:
  labels:
    - desktop
projectName: tor
projectFile: |
  options:
    - virtualnet: ':<random> address:10.0.0.50 default'
    - nat:
services:
  tor:
    makejail: !ENV '${OVERLORD_METADATA}/tor.makejail'
    volumes:
      - data: '/var/db/tor'

```

```
volumes:
  data:
    device: '/var/appjail-volumes/tor/data'
```

metadata.yml:

```
kind: metadata
datacenters:
  main:
    entrypoint: !ENV '${ENTRYPOINT}'
    access_token: !ENV '${TOKEN}'
deployIn:
  labels:
    - desktop
metadata:
  tor.makejail: |
    OPTION start
    OPTION overwrite=force

    INCLUDE gh+DtxdF/efficient-makejail

    PKG tor

    CMD echo "SocksPort 0.0.0.0:9050" > /usr/local/etc/tor/torrc
    CMD echo "HTTPTunnelPort 0.0.0.0:9080" >> /usr/local/etc/tor/torrc

    SERVICE tor oneenable
    SERVICE tor start
```

.env:

```
ENTRYPOINT=http://127.0.0.1:8888
TOKEN=<access token>
```

There is no difference between deploying a project and the metadata from the user’s point of view. However, metadata is not queued, it is simply written (asynchronously) to disk.

```
$ overlord apply -f metadata.yml
$ overlord apply -f tor.yml
$ overlord get-info -f metadata.yml -t metadata
datacenter: http://127.0.0.1:8888
  entrypoint: main
  chain: None
  labels:
    - all
    - desktop
    - services
    - vm-only
  metadata:
    tor.makejail: |
```

```

OPTION start
OPTION overwrite=force

INCLUDE gh+DtxdF/efficient-makejail

PKG tor

CMD echo "SocksPort 0.0.0.0:9050" > /usr/local/etc/tor/torrc
CMD echo "HTTPTunnelPort 0.0.0.0:9080" >> /usr/local/etc/tor/torrc

SERVICE tor onenable
SERVICE tor start
$ overlord get-info -f tor.yml -t projects --filter-per-project
datacenter: http://127.0.0.1:8888
entrypoint: main
chain: None
labels:
- all
- desktop
- services
- vm-only
projects:
tor:
state: UNFINISHED
last_log: 2025-04-22_18h40m30s
locked: True
services:
- {'name': 'tor', 'status': 0, 'jail': '7ce0dfdcef'}
up:
operation: RUNNING
last_update: 38.01 seconds
job_id: 16

```

Deploying FreeBSD VMs

Overlord can deploy virtual machines thanks to the great [vm-bhyve](#) project. A virtual machine isolates many parts that a jail cannot, with the overhead that such a thing implies, however that overhead is not a problem depending on what you are doing.

This deployment works as follows: A director file is created (overlord does this internally), which is used to further create a jail that represents the environment that must have [vm-bhyve](#) installed, must be configured to use the firewall (one supported by FreeBSD) and must be configured with the bridge used by the VMs. This sounds really complicated, but [there is a Makejail](#) that does it, so take a look at it for details. The Makejail mentioned above creates the environment with [vm-bhyve-devel](#) installed, configures pf(4) and creates a bridge with an assigned IPv4 (192.168.8.1/24), so we must assign our VM an IPv4 between that range. pf(4) is not configured to isolate further connections, so an application inside the VM can “escape” to other services, which may or may not be desirable depending on what it is doing.

vm.yml:

```
kind: vmJail
datacenters:
  main:
    entrypoint: !ENV '${ENTRYPOINT}'
    access_token: !ENV '${TOKEN}'
deployIn:
  labels:
    - !ENV '${DST}'
vmName: !ENV '${VM}'
makejail: 'gh+DtxdF/vm-makejail'
template:
  loader: 'bhyveload'
  cpu: !ENV '${CPU}'
  memory: !ENV '${MEM}'
  network0_type: 'virtio-net'
  network0_switch: 'public'
  wired_memory: 'YES'
diskLayout:
  driver: 'nvme'
  size: !ENV '${DISK}'
  from:
    type: 'components'
    components:
      - base.txz
      - kernel.txz
    osArch: amd64
    osVersion: !ENV '${VERSION}-RELEASE'
disk:
  scheme: 'gpt'
  partitions:
    - type: 'freebsd-boot'
      size: '512k'
      alignment: '1m'
    - type: 'freebsd-swap'
      size: !ENV '${SWAP}'
      alignment: '1m'
    - type: 'freebsd-ufs'
      alignment: '1m'
      format:
        flags: '-Uj'
  bootcode:
    bootcode: '/boot/pmbr'
    partcode: '/boot/gptboot'
    index: 1
fstab:
```

```

- device: '/dev/nda0p3'
  mountpoint: '/'
  type: 'ufs'
  options: 'rw,sync'
  dump: 1
  pass: 1
- device: '/dev/nda0p2'
  mountpoint: 'none'
  type: 'swap'
  options: 'sw'
  dump: 0
  pass: 0
script-environment:
  - HOSTNAME: !ENV '${HOSTNAME}'
script: |
  set -xe
  set -o pipefail

  . "/metadata/environment"

sysrc -f /mnt/etc/rc.conf ifconfig_vtnet0="inet 192.168.8.2/24"
sysrc -f /mnt/etc/rc.conf defaultrouter="192.168.8.1"
sysrc -f /mnt/etc/rc.conf fsck_y_enable="YES"
sysrc -f /mnt/etc/rc.conf clear_tmp_enable="YES"
sysrc -f /mnt/etc/rc.conf dumpdev="NO"
sysrc -f /mnt/etc/rc.conf moused_nondefault_enable="NO"
sysrc -f /mnt/etc/rc.conf hostname="${HOSTNAME}"

if [ -f "/metadata/resolv.conf" ]; then
  cp -a /metadata/resolv.conf /mnt/etc/resolv.conf
fi

if [ -f "/metadata/loader.conf" ]; then
  cp /metadata/loader.conf /mnt/boot/loader.conf
fi

if [ -f "/metadata/zerotier_network" ]; then
  pkg -c /mnt install -y zerotier

  zerotier_network='head -1 -- "/metadata/zerotier_network"'

  cat << EOF > /mnt/etc/rc.local
  while ;; do
    if ! /usr/local/bin/zerotier-cli join ${zerotier_network}; then
      sleep 1
      continue
    fi

```

```

        break
    done

    rm -f /etc/rc.local
EOF

    sysrc -f /mnt/etc/rc.conf zerotier_enable="YES"
elif [ -f "/metadata/ts_auth_key" ]; then
    pkg -c /mnt install -y tailscale

    ts_auth_key='head -1 -- "/metadata/ts_auth_key"'

    echo "/usr/local/bin/tailscale up --accept-dns=false --auth-key=\"${ts_auth_key}\" &&
rm -f /etc/rc.local" > /mnt/etc/rc.local

    sysrc -f /mnt/etc/rc.conf tailscaled_enable="YES"
fi

if [ -f "/metadata/timezone" ]; then
    timezone='head -1 -- "/metadata/timezone"'

    ln -fs "/usr/share/zoneinfo/${timezone}" /mnt/etc/localtime
fi

if [ -f "/metadata/sshd_config" ]; then
    sysrc -f /mnt/etc/rc.conf sshd_enable="YES"
    cp /metadata/sshd_config /mnt/etc/ssh/sshd_config
fi

if [ -f "/metadata/ssh_key" ]; then
    cp /metadata/ssh_key /mnt/etc/ssh/authorized_keys
fi

if [ -f "/metadata/sysctl.conf" ]; then
    cp /metadata/sysctl.conf /mnt/etc/sysctl.conf
fi

if [ -f "/metadata/pkg.conf" ]; then
    mkdir -p /mnt/usr/local/etc/pkg/repos
    cp /metadata/pkg.conf /mnt/usr/local/etc/pkg/repos/Latest.conf
fi
metadata:
- resolv.conf
- loader.conf
- timezone
- sshd_config

```

- ssh_key
- sysctl.conf
- pkg.conf
- ts_auth_key

metadata.yml:

```
kind: metadata
datacenters:
  main:
    entrypoint: !ENV '${ENTRYPOINT}'
    access_token: !ENV '${TOKEN}'
deployIn:
  labels:
    - vm-only
metadata:
  ts_auth_key: '<tailscale auth key>'
  resolv.conf: |
    nameserver 192.168.1.107
  timezone: 'America/Caracas'
  loader.conf: |
    nvme_load="YES"
    if_bridge_load="YES"
    bridgestp_load="YES"
    if_wg_load="YES"
    kern.racct.enable=1
  ssh_key: '<SSH public key>'
  sshd_config: |
    # Ports
    Port 22

    # Authentication
    PubkeyAuthentication yes
    AuthenticationMethods publickey
    PermitRootLogin prohibit-password
    PrintMotd no

    # Forwarding
    X11Forwarding no
    AllowAgentForwarding yes

    # Connection checks
    ClientAliveCountMax 3
    ClientAliveInterval 15

    # Compression
    Compression no
```

```

# Limits
LoginGraceTime 40

# Public keys
AuthorizedKeysFile      /etc/ssh/authorized_keys

# SFTP
Subsystem sftp internal-sftp
sysctl.conf: |
# A bit of hardening
security.bsd.see_other_uids=0
security.bsd.see_other_gids=0
security.bsd.see_jail_proc=0
kern.randompid=1

# Allow packet filtering in if_bridge(4)
net.link.bridge.pfil_member=1
net.link.bridge.pfil_bridge=1
pkg.conf: |
FreeBSD: {
  url: "pkg+http://pkg.FreeBSD.org/${ABI}/latest",
  mirror_type: "srv",
  signature_type: "fingerprints",
  fingerprints: "/usr/share/keys/pkg",
  enabled: yes
}

```

.profile-vmtest.env:

```

ENTRYPOINT=http://127.0.0.1:8888
TOKEN=<access token>
VM=vmtest
CPU=1
MEM=256M
DISK=10G
VERSION=14.2
SWAP=1G
HOSTNAME=vmtest
DST=provider

```

Instead of copy and paste the deployment file each time you want to deploy a virtual machine, it is preferable to create several environment (or profile-like) files.

```

$ overlord -e .profile-vmtest.env apply -f metadata.yml
$ overlord -e .profile-vmtest.env apply -f vm.yml
$ overlord -e .profile-vmtest.env get-info -f vm.yml -t projects --filter-per-project
datacenter: http://127.0.0.1:8888
entrypoint: main
chain: None

```

```

labels:
  - all
  - provider
  - vm-only
projects:
  vmtest:
    state: UNFINISHED
    last_log: 2025-04-22_20h19m34s
    locked: True
    services:
      - {'name': 'vm', 'status': 0, 'jail': 'vmtest'}
    up:
      operation: RUNNING
      last_update: 58.85 seconds
      job_id: 17

```

Depending on the type of installation, this may take some time. In the above case we chose to install FreeBSD from its components, so if the server does not have them yet or if it has them but they change remotely (for example: modification time), Overlord will proceed to download them.

```

$ overlord -e .profile-vmtest.env get-info -f vm.yml -t projects --filter-per-project
datacenter: http://127.0.0.1:8888
entrypoint: main
chain: None
labels:
  - all
  - provider
  - vm-only
projects:
  vmtest:
    state: DONE
    last_log: 2025-04-22_20h19m34s
    locked: False
    services:
      - {'name': 'vm', 'status': 0, 'jail': 'vmtest'}
    up:
      operation: COMPLETED
      output:
        rc: 0
        stdout: {'errlevel': 0, 'message': None, 'failed': []}
      last_update: 6 minutes and 10.02 seconds
      job_id: 17
      restarted: False
$ overlord -e .profile-vmtest.env get-info -f vm.yml -t vm --filter-per-project
datacenter: http://127.0.0.1:8888
entrypoint: main
chain: None

```

```

labels:
  - all
  - provider
  - vm-only
projects:
  vmtest:
    virtual-machines:
      operation: COMPLETED
      output: |
        md0 created
        md0p1 added
        md0p2 added
        md0p3 added
        /dev/md0p3: 9214.0MB (18870272 sectors) block size 32768, fragment size 4096
          using 15 cylinder groups of 625.22MB, 20007 blks, 80128 inodes.
          with soft updates
        super-block backups (for fsck_ffs -b #) at:
          192, 1280640, 2561088, 3841536, 5121984, 6402432, 7682880, 8963328, 10243776,
          11524224, 12804672, 14085120, 15365568, 16646016, 17926464
        Using inode 4 in cg 0 for 75497472 byte journal
        bootcode written to md0
        partcode written to md0p1
        ifconfig_vtnet0: -> inet 192.168.8.2/24
        defaultrouter: NO -> 192.168.8.1
        fsck_y_enable: NO -> YES
        clear_tmp_enable: NO -> YES
        dumpdev: NO -> NO
        moused_nondefault_enable: YES -> NO
        hostname: -> vmtest
        [vmtest.appjail] Installing pkg-2.1.0...
        [vmtest.appjail] Extracting pkg-2.1.0: ..... done
        Updating FreeBSD repository catalogue...
        [vmtest.appjail] Fetching meta.conf: . done
        [vmtest.appjail] Fetching data.pkg: ..... done
        Processing entries: ..... done
        FreeBSD repository update completed. 35950 packages processed.
        All repositories are up to date.
        The following 2 package(s) will be affected (of 0 checked):

        New packages to be INSTALLED:
          ca_root_nss: 3.108
          tailscale: 1.82.5

        Number of packages to be installed: 2

        The process will require 35 MiB more space.
        11 MiB to be downloaded.

```

```

[vmtest.appjail] [1/2] Fetching tailscale-1.82.5.pkg: ..... done
[vmtest.appjail] [2/2] Fetching ca_root_nss-3.108.pkg: ..... done
Checking integrity... done (0 conflicting)
[vmtest.appjail] [1/2] Installing ca_root_nss-3.108...
[vmtest.appjail] [1/2] Extracting ca_root_nss-3.108: ..... done
Scanning /usr/share/certs/untrusted for certificates...
Scanning /usr/share/certs/trusted for certificates...
Scanning /usr/local/share/certs for certificates...
[vmtest.appjail] [2/2] Installing tailscale-1.82.5...
[vmtest.appjail] [2/2] Extracting tailscale-1.82.5: ..... done
=====
Message from ca_root_nss-3.108:

--

FreeBSD does not, and can not warrant that the certification authorities
whose certificates are included in this package have in any way been
audited for trustworthiness or RFC 3647 compliance.

Assessment and verification of trust is the complete responsibility of
the system administrator.

This package installs symlinks to support root certificate discovery
for software that either uses other cryptographic libraries than
OpenSSL, or use OpenSSL but do not follow recommended practice.

If you prefer to do this manually, replace the following symlinks with
either an empty file or your site-local certificate bundle.

    * /etc/ssl/cert.pem
    * /usr/local/etc/ssl/cert.pem
    * /usr/local/openssl/cert.pem
tailscaled_enable: -> YES
sshd_enable: NO -> YES
vm_list: -> vmtest
Starting vmtest
    * found guest in /vm/vmtest
    * booting...
newfs: soft updates journaling set
+ set -o pipefail
+ . /metadata/environment
+ export 'HOSTNAME=vmtest'
+ sysrc -f /mnt/etc/rc.conf 'ifconfig_vtnet0=inet 192.168.8.2/24'
+ sysrc -f /mnt/etc/rc.conf 'defaultrouter=192.168.8.1'
+ sysrc -f /mnt/etc/rc.conf 'fsck_y_enable=YES'
+ sysrc -f /mnt/etc/rc.conf 'clear_tmp_enable=YES'
+ sysrc -f /mnt/etc/rc.conf 'dumpdev=NO'
+ sysrc -f /mnt/etc/rc.conf 'moused_nondefault_enable=NO'

```

```

+ sysrc -f /mnt/etc/rc.conf 'hostname=vmtest'
+ [ -f /metadata/resolv.conf ]
+ cp -a /metadata/resolv.conf /mnt/etc/resolv.conf
+ [ -f /metadata/loader.conf ]
+ cp /metadata/loader.conf /mnt/boot/loader.conf
+ [ -f /metadata/zerotier_network ]
+ [ -f /metadata/ts_auth_key ]
+ pkg -c /mnt install -y tailscale
+ head -1 -- /metadata/ts_auth_key
+ ts_auth_key=[REDACTED]
+ echo '/usr/local/bin/tailscale up --accept-dns=false --auth-key="[REDACTED]"
&& rm -f /etc/rc.local'
+ sysrc -f /mnt/etc/rc.conf 'tailscaled_enable=YES'
+ [ -f /metadata/timezone ]
+ head -1 -- /metadata/timezone
+ timezone=America/Caracas
+ ln -fs /usr/share/zoneinfo/America/Caracas /mnt/etc/localtime
+ [ -f /metadata/sshd_config ]
+ sysrc -f /mnt/etc/rc.conf 'sshd_enable=YES'
+ cp /metadata/sshd_config /mnt/etc/ssh/sshd_config
+ [ -f /metadata/ssh_key ]
+ cp /metadata/ssh_key /mnt/etc/ssh/authorized_keys
+ [ -f /metadata/sysctl.conf ]
+ cp /metadata/sysctl.conf /mnt/etc/sysctl.conf
+ [ -f /metadata/pkg.conf ]
+ mkdir -p /mnt/usr/local/etc/pkg/repos
+ cp /metadata/pkg.conf /mnt/usr/local/etc/pkg/repos/Latest.conf
last_update: 5 minutes and 12.6 seconds
job_id: 17

```

As I'm using tailscale and the VM above is about to be configured to join my tailnet, after a while it should appear in the node list:

```

$ tailscale status
...
100.124.236.28  vmtest          REDACTED@    freebsd -
$ ssh root@100.124.236.28
The authenticity of host '100.124.236.28 (100.124.236.28)' can't be established.
ED25519 key fingerprint is SHA256:0c61mU8erpgS2evkwL9Wh00l4Ze94sSNfhImLy3b4UQ.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '100.124.236.28' (ED25519) to the list of known hosts.
root@vmtest:~ #

```

Service Discovery

A service is deployed, however depending on how many servers you have in your cluster the service endpoint can be a difficult part to build. You know the port and external inter-

face used and certainly the IP address is easy to get, but it is much easier to use DNS, which is its primary purpose. The service can be deployed on different servers but its endpoint is always the same.

SkyDNS is an older but powerful protocol that, in combination with Etcd, can provide easy service discovery. Overlord can be configured to use both Etcd and SkyDNS. Let's deploy our Etcd cluster.

etcd.yml:

```
kind: directorProject
datacenters:
  main:
    entrypoint: !ENV '${ENTRYPOINT}'
    access_token: !ENV '${TOKEN}'
deployIn:
  labels:
    - desktop
    - r2
    - centralita
projectName: etcd-cluster
projectFile: |
  options:
    - alias:
    - ip4_inherit:
  services:
    etcd:
      makejail: 'gh+AppJail-makejails/etcd'
      arguments:
        - etcd_tag: '14.2-34'
      volumes:
        - data: etcd-data
      start-environment:
        - ETCD_NAME: !ENV '${NAME}'
        - ETCD_ADVERTISE_CLIENT_URLS: !ENV 'http://${HOSTIP}:2379'
        - ETCD_LISTEN_CLIENT_URLS: !ENV 'http://${HOSTIP}:2379'
        - ETCD_LISTEN_PEER_URLS: !ENV 'http://${HOSTIP}:2380'
        - ETCD_INITIAL_ADVERTISE_PEER_URLS: !ENV 'http://${HOSTIP}:2380'
        - ETCD_INITIAL_CLUSTER_TOKEN: 'etcd-demo-cluster'
        - ETCD_INITIAL_CLUSTER: !ENV '${CLUSTER}'
        - ETCD_INITIAL_CLUSTER_STATE: 'new'
        - ETCD_HEARTBEAT_INTERVAL: '5000'
        - ETCD_ELECTION_TIMEOUT: '50000'
        - ETCD_LOG_LEVEL: 'error'
      default_volume_type: '<volumeefs>'
      volumes:
        data:
          device: /var/appjail-volumes/etcd-cluster/data
environment:
```

```

CLUSTER: 'etcd0=http://100.65.139.52:2380,etcd1=http://100.109.0.125:2380,etc-
d2=http://100.96.18.2:2380'
labelsEnvironment:
  desktop:
    NAME: 'etcd0'
    HOSTIP: '100.65.139.52'
  r2:
    NAME: 'etcd1'
    HOSTIP: '100.109.0.125'
  centralita:
    NAME: 'etcd2'
    HOSTIP: '100.96.18.2'

```

Profit!

```

$ overlord apply -f etcd.yml
$ overlord get-info -f etcd.yml -t projects --filter-per-project
datacenter: http://127.0.0.1:8888
  entrypoint: main
  chain: None
  labels:
    - all
    - desktop
    - services
    - vm-only
  projects:
    etcd-cluster:
      state: DONE
      last_log: 2025-04-23_02h28m36s
      locked: False
      services:
        - {'name': 'etcd', 'status': 0, 'jail': 'f094a31c46'}
      up:
        operation: COMPLETED
        output:
          rc: 0
          stdout: {'errlevel': 0, 'message': None, 'failed': []}
        last_update: 8 minutes and 11.51 seconds
        job_id: 20
        restarted: False
        labels:
          error: False
          message: None
datacenter: http://127.0.0.1:8888
  entrypoint: main
  chain: centralita
  labels:
    - all

```

```

- centralita
- services
projects:
  etcd-cluster:
    state: DONE
    last_log: 2025-04-23_02h28m37s
    locked: False
    services:
      - {'name': 'etcd', 'status': 0, 'jail': '1ff836df47'}
    up:
      operation: COMPLETED
      output:
        rc: 0
        stdout: {'errlevel': 0, 'message': None, 'failed': []}
      last_update: 5 minutes and 37.82 seconds
      job_id: 2
      restarted: False
      labels:
        error: False
        message: None
datacenter: http://127.0.0.1:8888
entrypoint: main
chain: r2
labels:
  - all
  - r2
  - services
projects:
  etcd-cluster:
    state: DONE
    last_log: 2025-04-23_02h28m38s
    locked: False
    services:
      - {'name': 'etcd', 'status': 0, 'jail': '756ae9d5ca'}
    up:
      operation: COMPLETED
      output:
        rc: 0
        stdout: {'errlevel': 0, 'message': None, 'failed': []}
      last_update: 5 minutes and 5.04 seconds
      job_id: 1
      restarted: False
      labels:
        error: False
        message: None

```

Once the deployment is done, it is time to configure each Overlord instance with the fol-

lowing parameters.

/usr/local/etc/overlord.yml:

```
etcd:
  100.65.139.52: {}
  100.109.0.125: {}
  100.96.18.2: {}
```

Remember to restart the Overlord processes for the changes to take effect.

```
supervisorctl restart overlord:
```

The next service we have to deploy is CoreDNS. Thanks to it we can use SkyDNS through the [Etcd plugin](#).

coredns.yml:

```
kind: directorProject
datacenters:
  main:
    entrypoint: !ENV '${ENTRYPOINT}'
    access_token: !ENV '${TOKEN}'
deployIn:
  labels:
    - desktop
    - r2
projectName: dns-server
projectFile: |
  options:
    - alias:
    - ip4_inherit:
  services:
    coredns:
      makejail: !ENV '${OVERLORD_METADATA}/coredns.makejail'
```

metadata.yml:

```
kind: metadata
datacenters:
  main:
    entrypoint: !ENV '${ENTRYPOINT}'
    access_token: !ENV '${TOKEN}'
deployIn:
  labels:
    - desktop
    - r2
metadata:
  Corefile: |
    .:53 {
      bind tailscale0
      log
```

```

errors
forward . 208.67.222.222 208.67.220.220
etcd overlord.lan. {
    endpoint http://100.65.139.52:2379 http://100.109.0.125:2379 http://100.96.18.2:2379
}
hosts /etc/hosts namespace.lan.
cache 30
}
coredns.hosts: |
100.65.139.52    controller.namespace.lan
100.96.18.2     centralita.namespace.lan
100.127.18.7    fbsd4dev.namespace.lan
100.123.177.93  provider.namespace.lan
100.109.0.125   r2.namespace.lan
172.16.0.3      cicd.namespace.lan
coredns.makejail: |
OPTION start
OPTION overwrite=force
OPTION healthcheck="health_cmd:jail:service coredns status" "recover_cmd:jail:service
coredns restart"

INCLUDE gh+DtxdF/efficient-makejail

CMD mkdir -p /usr/local/etc/pkg/repos
COPY ${OVERLORD_METADATA}/coredns.pkg.conf /usr/local/etc/pkg/repos/Latest.conf

PKG coredns

CMD mkdir -p /usr/local/etc/coredns
COPY ${OVERLORD_METADATA}/Corefile /usr/local/etc/coredns/Corefile

COPY ${OVERLORD_METADATA}/coredns.hosts /etc/hosts

SYSRC coredns_enable=YES
SERVICE coredns start
coredns.pkg.conf: |
FreeBSD: {
    url: "pkg+https://pkg.FreeBSD.org/${ABI}/latest",
    mirror_type: "srv",
    signature_type: "fingerprints",
    fingerprints: "/usr/share/keys/pkg",
    enabled: yes
}

```

As you can see in the CoreDNS configuration file, the **overlord.lan.** zone is assumed but by default Overlord only uses **.** which does not make sense for this context, so proceed to configure Overlord with this in mind and deploy CoreDNS.

/usr/local/etc/overlord.yml:

```
skydns:
  zone: 'overlord.lan.'
```

Note: Remember to restart the Overlord processes for the changes to take effect.

```
$ overlord apply -f metadata.yml
$ overlord apply -f coredns.yml
$ overlord get-info -f coredns.yml -t projects --filter-per-project
datacenter: http://127.0.0.1:8888
  entrypoint: main
  chain: None
  labels:
    - all
    - desktop
    - services
    - vm-only
  projects:
    dns-server:
      state: DONE
      last_log: 2025-04-23_13h32m49s
      locked: False
      services:
        - {'name': 'coredns', 'status': 0, 'jail': '8106aaca6d'}
    up:
      operation: COMPLETED
      output:
        rc: 0
        stdout: {'errlevel': 0, 'message': None, 'failed': []}
      last_update: 2 minutes and 30.14 seconds
      job_id: 25
      restarted: False
      labels:
        error: False
        message: None
datacenter: http://127.0.0.1:8888
  entrypoint: main
  chain: r2
  labels:
    - all
    - r2
    - services
  projects:
    dns-server:
      state: DONE
      last_log: 2025-04-23_13h32m54s
      locked: False
      services:
        - {'name': 'coredns', 'status': 0, 'jail': '9516eb48aa'}
```

```

up:
  operation: COMPLETED
  output:
    rc: 0
    stdout: {'errlevel': 0, 'message': None, 'failed': []}
  last_update: 3 minutes and 26.9 seconds
  job_id: 4
  restarted: False
  labels:
    error: False
    message: None

```

Our Etcd cluster is up and running and our DNS servers are up and running. Clients should be configured to resolve DNS hostnames through those DNS servers, so configure them in their **resolv.conf(5)** or similar.

/etc/resolv.conf:

```

nameserver 100.65.139.52
nameserver 100.109.0.125

```

Our Frankenstein is alive! So the next step is to deploy a service and test if all parts are working as expected.

homebox.yml:

```

kind: directorProject
datacenters:
  main:
    entrypoint: !ENV '${ENTRYPOINT}'
    access_token: !ENV '${TOKEN}'
deployIn:
  labels:
    - centralita
projectName: homebox
projectFile: |
  options:
    - virtualnet: ':<random> default'
    - nat:
services:
  homebox:
    makejail: gh+AppJail-makejails/homebox
    options:
      - expose: '8666:7745 ext_if:tailscale0 on_if:tailscale0'
      - label: 'overlord.skydns:1'
      - label: 'overlord.skydns.group:homebox'
      - label: 'overlord.skydns.interface:tailscale0'
  volumes:
    - data: homebox-data

```

```

arguments:
  - homebox_tag: 14.2
default_volume_type: '<volumeefs>'
volumes:
  data:
    device: /var/appjail-volumes/homebox/data

```

So simple. Overlord intercepts the labels that we define in our [Director](#) file and based on that it creates the DNS record.

```

$ overlord apply -f homebox.yml
$ overlord get-info -f homebox.yml -t projects --filter-per-project
datacenter: http://127.0.0.1:8888
entrypoint: main
chain: centralita
labels:
  - all
  - centralita
  - services
projects:
  homebox:
    state: DONE
    last_log: 2025-04-23_15h44m38s
    locked: False
    services:
      - {'name': 'homebox', 'status': 0, 'jail': '1f97e32f36'}
    up:
      operation: COMPLETED
      output:
        rc: 0
        stdout: {'errlevel': 0, 'message': None, 'failed': []}
      last_update: 4 minutes and 4.1 seconds
      job_id: 6
      restarted: False
      labels:
        error: False
        message: None
        load-balancer:
          services:
            homebox:
              error: False
              message: None
      skydns:
        services:
          homebox:
            error: False
            message: (project:homebox, service:homebox, records:[address:True,ptr:None,s-
rv:None] records has been updated.

```

Finally, our endpoint is <http://homebox.overlord.lan:8666/>

Load Balancing

An interesting fact about SkyDNS is that multiple domains are grouped, so if we deploy a service on multiple servers and they use the same group, a DNS request will return three A records (in the case of IPv4), or what amounts to three IPv4 addresses.

hello-http.yml:

```
kind: directorProject
datacenters:
  main:
    entrypoint: !ENV '${ENTRYPOINT}'
    access_token: !ENV '${TOKEN}'
deployIn:
  labels:
    - services
projectName: hello-http
projectFile: |
  options:
    - virtualnet: ':<random> default'
    - nat:
services:
  darkhttpd:
    makejail: 'gh+DtxdF/hello-http-makejail'
    options:
      - expose: '9128:80 ext_if:tailscale0 on_if:tailscale0'
      - label: 'appjail.dns.alt-name:hello-http'
      - label: 'overlord.skydns:1'
      - label: 'overlord.skydns.group:hello-http'
      - label: 'overlord.skydns.interface:tailscale0'
    arguments:
      - darkhttpd_tag: 14.2
```

A nice side effect of this is that services are load-balanced in a round-robin fashion, although this is entirely client-dependent, but most modern ones do it.

```
$ overlord apply -f hello-http.yml
$ overlord get-info -f hello-http.yml -t projects --filter-per-project
datacenter: http://127.0.0.1:8888
entrypoint: main
chain: None
labels:
  - all
  - desktop
  - services
  - vm-only
projects:
  hello-http:
    state: DONE
```

```

last_log: 2025-04-23_16h26m08s
locked: False
services:
  - {'name': 'darkhttpd', 'status': 0, 'jail': '7c2225c5fe'}
up:
  operation: COMPLETED
  output:
    rc: 0
    stdout: {'errlevel': 0, 'message': None, 'failed': []}
  last_update: 2 minutes and 43.3 seconds
  job_id: 28
  restarted: False
  labels:
    error: False
    message: None
    load-balancer:
      services:
        darkhttpd:
          error: False
          message: None
    skydns:
      services:
        darkhttpd:
          error: False
          message: (project:hello-http, service:darkhttpd, records:[ad-
dress:True,ptr:None,srv:None] records has been updated.
datacenter: http://127.0.0.1:8888
  entriypoint: main
  chain: centralita
  labels:
    - all
    - centralita
    - services
projects:
  hello-http:
    state: DONE
    last_log: 2025-04-23_16h26m09s
    locked: False
    services:
      - {'name': 'darkhttpd', 'status': 0, 'jail': '3822f65e97'}
    up:
      operation: COMPLETED
      output:
        rc: 0
        stdout: {'errlevel': 0, 'message': None, 'failed': []}
      last_update: 2 minutes and 18.56 seconds
      job_id: 13

```

```

restarted: False
labels:
  error: False
  message: None
  load-balancer:
    services:
      darkhttpd:
        error: False
        message: None
  skydns:
    services:
      darkhttpd:
        error: False
        message: (project:hello-http, service:darkhttpd, records:[ad-
datacenter: http://127.0.0.1:8888
entrypoint: main
chain: r2
labels:
  - all
  - r2
  - services
projects:
  hello-http:
    state: DONE
    last_log: 2025-04-23_16h26m10s
    locked: False
    services:
      - {'name': 'darkhttpd', 'status': 0, 'jail': '0e0e64eb3c'}
up:
  operation: COMPLETED
  output:
    rc: 0
    stdout: {'errlevel': 0, 'message': None, 'failed': []}
  last_update: 51.17 seconds
  job_id: 8
  restarted: False
  labels:
    error: False
    message: None
    load-balancer:
      services:
        darkhttpd:
          error: False
          message: None
  skydns:
    services:

```

```

darkhttpd:
  error: False
  message: (project:hello-http, service:darkhttpd, records:[ad-
dress:True,ptr:None,srv:None] records has been updated.
$ host -t A hello-http.overlord.lan
hello-http.overlord.lan has address 100.65.139.52
hello-http.overlord.lan has address 100.109.0.125
hello-http.overlord.lan has address 100.96.18.2
$ curl http://hello-http.overlord.lan:9128/
curl http://hello-http.overlord.lan:9128
Hello, world!
UUID: 472ffbdb-9472-4aa2-95ff-39f4bde214df
$ curl http://hello-http.overlord.lan:9128/
Hello, world!
UUID: 7db3b268-87bb-4e81-8be3-e888378fa13b

```

However, I know that in most cases a more complex configuration is needed. Worse, as noted above this is client-dependent, so it may or may not fit your intent.

Fortunately, Overlord comes with an integration with HAProxy, or more specifically with Data Plane API, so your configuration can be as complex as you need.

haproxy.yml:

```

kind: directorProject
datacenters:
  main:
    entrypoint: !ENV '${ENTRYPOINT}'
    access_token: !ENV '${TOKEN}'
deployIn:
  labels:
    - desktop
    - r2
projectName: load-balancer
projectFile: |
  options:
    - alias:
    - ip4_inherit:
services:
  haproxy:
    makejail: !ENV '${OVERLORD_METADATA}/haproxy.makejail'
    arguments:
      - haproxy_tag: 14.2-dataplaneapi
    options:
      - label: 'overlord.skydns:1'
      - label: 'overlord.skydns.group:revproxy'
      - label: 'overlord.skydns.interface:tailscale0'

```

metadata.yml:

```
kind: metadata
datacenters:
  main:
    entrypoint: !ENV '${ENTRYPOINT}'
    access_token: !ENV '${TOKEN}'
deployIn:
  labels:
    - desktop
    - r2
metadata:
  haproxy.makejail: |
    ARG haproxy_tag=13.5
    ARG haproxy_ajspect=gh+AppJail-makejails/haproxy

    OPTION start
    OPTION overwrite=force
    OPTION copydir=${OVERLORD_METADATA}
    OPTION file=/haproxy.conf

    FROM --entrypoint "${haproxy_ajspect}" haproxy:${haproxy_tag}

    INCLUDE gh+DtxdF/efficient-makejail

    SYSRC haproxy_enable=YES
    SYSRC haproxy_config=/haproxy.conf

    SERVICE haproxy start

    STOP

    STAGE start

    WORKDIR /dataplaneapi

    RUN daemon \
      -r \
      -t "Data Plane API" \
      -P .master \
      -p .pid \
      -o .log \
      ./dataplaneapi \
        -f /usr/local/etc/dataplaneapi.yml \
        --host=0.0.0.0 \
        --port=5555 \
        --spoe-dir=/usr/local/etc/haproxy/spoe \
```

```

--haproxy-bin=/usr/local/sbin/haproxy \
--reload-cmd="service haproxy reload" \
--restart-cmd="service haproxy restart" \
--status-cmd="service haproxy status" \
--maps-dir=/usr/local/etc/haproxy/maps \
--config-file=/haproxy.conf \
--ssl-certs-dir=/usr/local/etc/haproxy/ssl \
--general-storage-dir=/usr/local/etc/haproxy/general \
--dataplane-storage-dir=/usr/local/etc/haproxy/dataplane \
--log-to=file \
--userlist=dataplaneapi

haproxy.conf: |
  userlist dataplaneapi
    user admin insecure-password cuwBvS5XMphtCNuC

  global
    daemon
    log 127.0.0.1:514 local0
    log-tag HAProxy

  defaults
    mode http
    log global
    option httplog
    timeout client 30s
    timeout server 50s
    timeout connect 10s
    timeout http-request 10s

  frontend web
    bind :80
    default_backend web

  backend web
    option httpchk HEAD /
    balance roundrobin

```

We are about to deploy HAProxy / Data Plane API on two servers. The reason for doing this is to avoid SPOF (single point of failure), at least if one instance of HAProxy / Data Plane API goes down at any time, the other will rescue us. However, Overlord can only point to one instance of Data Plane API, so if we use two (as below) servers, we need to specify one instance on one and a different instance on the other. As you can see in the [Director](#) file, we have used SkyDNS, so that clients can use the **revproxy.overlord.lan** domain instead of each individual IP address, with the advantage that even if one instance of HAProxy / Data Plane API is down, we have the other and the client can make requests to the other.

```

$ overlord apply -f metadata.yml
$ overlord apply -f haproxy.yml
$ overlord get-info -f haproxy.yml -t projects --filter-per-project

```

```

datacenter: http://127.0.0.1:8888
  entrypoint: main
  chain: None
  labels:
    - all
    - desktop
    - services
    - vm-only
  projects:
    load-balancer:
      state: DONE
      last_log: 2025-04-23_17h04m01s
      locked: False
      services:
        - {'name': 'haproxy', 'status': 0, 'jail': '8d92fc6d2d'}
      up:
        operation: COMPLETED
        output:
          rc: 0
          stdout: {'errlevel': 0, 'message': None, 'failed': []}
        last_update: 2 minutes and 20.12 seconds
        job_id: 30
        restarted: False
        labels:
          error: False
          message: None
          load-balancer:
            services:
              haproxy:
                error: False
                message: None
          skydns:
            services:
              haproxy:
                error: False
                message: (project:load-balancer, service:haproxy, records:[ad-
dress:True,ptr:None,srv:None] records has been updated.
datacenter: http://127.0.0.1:8888
  entrypoint: main
  chain: r2
  labels:
    - all
    - r2
    - services
  projects:
    load-balancer:
      state: DONE

```

```
last_log: 2025-04-23_17h04m02s
locked: False
services:
  - {'name': 'haproxy', 'status': 0, 'jail': '05c589c8a1'}
up:
  operation: COMPLETED
  output:
    rc: 0
    stdout: {'errlevel': 0, 'message': None, 'failed': []}
  last_update: 2 minutes and 53.27 seconds
  job_id: 10
  restarted: False
  labels:
    error: False
    message: None
  load-balancer:
    services:
      haproxy:
        error: False
        message: None
  skydns:
    services:
      haproxy:
        error: False
        message: (project:load-balancer, service:haproxy, records:[ad-
dress:True,ptr:None,srv:None] records has been updated.
```

/usr/local/etc/overlord.yml (centralita)

```
dataplaneapi:
  auth:
    username: 'admin'
    password: 'cuwBvS5XMphtCNuC'
  entrypoint: 'http://100.65.139.52:5555'
```

/usr/local/etc/overlord.yml (provider):

```
dataplaneapi:
  auth:
    username: 'admin'
    password: 'cuwBvS5XMphtCNuC'
  entrypoint: 'http://100.109.0.125:5555'
```

hello-http.yml:

```
kind: directorProject
datacenters:
  main:
    entrypoint: !ENV '${ENTRYPOINT}'
```

```

    access_token: !ENV '${TOKEN}'
deployIn:
  labels:
    - centralita
    - provider
projectName: hello-http
projectFile: |
  options:
    - virtualnet: ':<random> default'
    - nat:
services:
  darkhttpd:
    makejail: 'gh+DtxdF/hello-http-makejail'
    options:
      - expose: '9128:80 ext_if:tailscale0 on_if:tailscale0'
      - label: 'overlord.load-balancer:1'
      - label: 'overlord.load-balancer.backend:web'
      - label: 'overlord.load-balancer.interface:tailscale0'
      - label: 'overlord.load-balancer.interface.port:9128'
      - label: 'overlord.load-balancer.set.check:"enabled"'
    arguments:
      - darkhttpd_tag: 14.2

```

Profit!

```

$ overlord apply -f hello-http.yml
$ overlord get-info -f hello-http.yml -t projects --filter-per-project
datacenter: http://127.0.0.1:8888
entrypoint: main
chain: provider
labels:
  - all
  - provider
  - vm-only
projects:
  hello-http:
    state: DONE
    last_log: 2025-04-23_17h57m16s
    locked: False
    services:
      - {'name': 'darkhttpd', 'status': 0, 'jail': '79f16243de'}
    up:
      operation: COMPLETED
      output:
        rc: 0
        stdout: {'errlevel': 0, 'message': None, 'failed': []}
      last_update: 1 minute and 22.1 seconds
      job_id: 1

```

```

restarted: False
labels:
  error: False
  message: None
  load-balancer:
    services:
      darkhttpd:
        error: False
        message: (project:hello-http, service:darkhttpd, backend:web, serverid:-
fa8f94b1-6b2b-4cb4-a808-e9da46014c86, code:202, transaction_id:8fcf5d67-12df-4fcd-a2e3-
1f5f18fe1844, commit:1) server has been successfully added.
      skydns:
        services:
          darkhttpd:
            error: False
            message: None
datacenter: http://127.0.0.1:8888
entrypoint: main
chain: centralita
labels:
  - all
  - centralita
  - services
projects:
  hello-http:
    state: DONE
    last_log: 2025-04-23_17h57m16s
    locked: False
    services:
      - {'name': 'darkhttpd', 'status': 0, 'jail': '52dfa071cb'}
    up:
      operation: COMPLETED
      output:
        rc: 0
        stdout: {'errlevel': 0, 'message': None, 'failed': []}
      last_update: 1 minute and 19.53 seconds
      job_id: 15
      restarted: False
      labels:
        error: False
        message: None
        load-balancer:
          services:
            darkhttpd:
              error: False
              message: (project:hello-http, service:darkhttpd, backend:web, serverid:f-
4b9e170-67bb-403e-88da-112c55b45fce, code:202, transaction_id:0fa886c8-68a6-4716-aa6b-

```

```
824aa3e776ad, commit:1) server has been successfully added.
```

```
skydns:
```

```
services:
```

```
darkhttpd:
```

```
error: False
```

```
message: None
```

```
$ curl http://revproxy.overlord.lan
```

```
Hello, world!
```

```
UUID: 8579af73-7d11-40b3-8444-6dac62e34b8e
```

```
$ curl http://revproxy.overlord.lan
```

```
Hello, world!
```

```
UUID: e463b1d5-13eb-4f04-9b0a-caf4339a8058
```

Horizontal Autoscaling

Even when there are hundreds of servers, deploying projects is an easy task, however the problem with this approach is that we are wasting resources and the clients probably only use less than 5% of the resources of our cluster, or on the contrary, you can deploy your project on a few servers which you think is enough until you realize at some point that this is not enough, even worse, some servers can be down at any time for any reason. This is what Overlord autoscaling can solve.

hello-http.yml:

```
kind: directorProject
datacenters:
  main:
    entrypoint: !ENV '${ENTRYPOINT}'
    access_token: !ENV '${TOKEN}'
deployIn:
  labels:
    - desktop
projectName: hello-http
projectFile: |
  options:
    - virtualnet: ':<random> default'
    - nat:
services:
  darkhttpd:
    makejail: 'gh+DtxdF/hello-http-makejail'
    options:
      - expose: '9128:80 ext_if:tailscale0 on_if:tailscale0'
      - label: 'overlord.load-balancer:1'
      - label: 'overlord.load-balancer.backend:web'
      - label: 'overlord.load-balancer.interface:tailscale0'
      - label: 'overlord.load-balancer.interface.port:9128'
      - label: 'overlord.load-balancer.set.check:"enabled"'
    arguments:
      - darkhttpd_tag: 14.2
```

```

autoScale:
  replicas:
    min: 3
  labels:
    - services
    - provider

```

As you have probably noticed, we have specified two types of labels. This is the subtle difference between autoscaling and non-autoscaling deployments. Unlike non-autoscaling deployments, the labels in **deployIn.labels** are for using matching servers for autoscaling and monitoring, or in other words, the servers that match the specified labels (in this case **desktop**) are responsible for deployment, monitoring and, if necessary, redeployment. On the other side, the servers matching the labels in **autoScale.labels** (in this case **services** and **provider**) are for deploying the project as non-autoscaling deployments. We have specified that the project will have at least three replicas. There are other things we can specify like **rctl(8)** rules, but for simplicity this is sufficient.

```

$ overlord apply -f hello-http.yml
$ overlord get-info -f hello-http.yml -t projects --filter-per-project --use-autoscale-labels
datacenter: http://127.0.0.1:8888
  entrypoint: main
  chain: None
  labels:
    - all
    - desktop
    - services
    - vm-only
  projects:
    hello-http:
      state: DONE
      last_log: 2025-04-23_19h36m17s
      locked: False
      services:
        - {'name': 'darkhttpd', 'status': 0, 'jail': '0524bcf91b'}
      up:
        operation: COMPLETED
        output:
          rc: 0
          stdout: {'errlevel': 0, 'message': None, 'failed': []}
        last_update: 58.24 seconds
        job_id: 31
        restarted: False
        labels:
          error: False
          message: None
          load-balancer:
            services:

```

```

    darkhttpd:
      error: False
      message: (project:hello-http, service:darkhttpd, backend:web, serverid:0d67d160-61af-4810-b277-5fb9e20da8eb, code:202, transaction_id:baa5b939-f724-4bd3-9d65-2ef769def3f5, commit:1) server has been successfully added.
    skydns:
      services:
        darkhttpd:
          error: False
          message: None
datacenter: http://127.0.0.1:8888
entrypoint: main
chain: provider
labels:
  - all
  - provider
  - vm-only
projects:
  hello-http:
    state: DONE
    last_log: 2025-04-23_20h00m11s
    locked: False
    services:
      - {'name': 'darkhttpd', 'status': 0, 'jail': '2c2d22d2a5'}
    up:
      operation: COMPLETED
      output:
        rc: 0
        stdout: {'errlevel': 0, 'message': None, 'failed': []}
      last_update: 4 minutes and 46.3 seconds
      job_id: 6
      restarted: False
      labels:
        error: False
        message: None
      load-balancer:
        services:
          darkhttpd:
            error: False
            message: (project:hello-http, service:darkhttpd, backend:web, serverid:-fa8f94b1-6b2b-4cb4-a808-e9da46014c86, code:202, transaction_id:6792e6fe-a778-44a7-b23a-1b2c23fe5904, commit:1) server has been successfully updated.
        skydns:
          services:
            darkhttpd:
              error: False
              message: None

```

```

datacenter: http://127.0.0.1:8888
  entrypoint: main
  chain: centralita
  labels:
    - all
    - centralita
    - services
  projects:
    hello-http:
      state: DONE
      last_log: 2025-04-23_20h04m25s
      locked: False
      services:
        - {'name': 'darkhttpd', 'status': 0, 'jail': 'a6549318ce'}
      up:
        operation: COMPLETED
        output:
          rc: 0
          stdout: {'errlevel': 0, 'message': None, 'failed': []}
        last_update: 33.34 seconds
        job_id: 21
        restarted: False
        labels:
          error: False
          message: None
          load-balancer:
            services:
              darkhttpd:
                error: False
                message: (project:hello-http, service:darkhttpd, backend:web, serverid:f-
4b9e170-67bb-403e-88da-112c55b45fce, code:202, transaction_id:00e632ce-c215-4784-9e61-
9507d914ba6a, commit:1) server has been successfully updated.
          skydns:
            services:
              darkhttpd:
                error: False
                message: None
$ overlord get-info -f hello-http.yml -t autoscale --filter-per-project
datacenter: http://127.0.0.1:8888
  entrypoint: main
  chain: None
  labels:
    - all
    - desktop
    - services
    - vm-only
  projects:

```

```

hello-http:
  autoScale:
    last_update: 7.65 seconds
    operation: COMPLETED
    output:
      message: None
$ curl http://revproxy.overlord.lan
Hello, world!
UUID: 08951a86-2aef-4e85-9bfc-7fe68b5cc62d
$ curl http://revproxy.overlord.lan
Hello, world!
UUID: 5a06a89d-6109-438e-bc04-1ef739473994

```

Suppose the service in **provider** is down for any reason.

```

$ overlord get-info -f hello-http.yml -t projects --filter-per-project --use-autoscale-labels
datacenter: http://127.0.0.1:8888
entrypoint: main
chain: None
labels:
- all
- desktop
- services
- vm-only
projects:
  hello-http:
    state: DONE
    last_log: 2025-04-23_19h36m17s
    locked: False
    services:
      - {'name': 'darkhttpd', 'status': 0, 'jail': '0524bcf91b'}
    up:
      operation: COMPLETED
      output:
        rc: 0
        stdout: {'errlevel': 0, 'message': None, 'failed': []}
      last_update: 13 minutes and 37.64 seconds
      job_id: 32
      restarted: False
      labels:
        error: False
        message: None
      load-balancer:
        services:
          darkhttpd:
            error: False
            message: (project:hello-http, service:darkhttpd, backend:web, serve-

```

```
rid:0d67d160-61af-4810-b277-5fb9e20da8eb, code:202, transaction_id:a2ba93d7-6ce6-4a36-aab2-09be13a00c17, commit:1) server has been successfully updated.
```

```
skydns:
```

```
services:
```

```
darkhttpd:
```

```
error: False
```

```
message: None
```

```
datacenter: http://127.0.0.1:8888
```

```
entrypoint: main
```

```
chain: provider
```

```
labels:
```

```
- all
```

```
- provider
```

```
- vm-only
```

```
projects:
```

```
hello-http:
```

```
state: DONE
```

```
last_log: 2025-04-23_20h11m58s
```

```
locked: False
```

```
services:
```

```
- {'name': 'darkhttpd', 'status': 66, 'jail': '2c2d22d2a5'}
```

```
up:
```

```
operation: COMPLETED
```

```
output:
```

```
rc: 0
```

```
stdout: {'errlevel': 0, 'message': None, 'failed': []}
```

```
last_update: 1 minute and 13.9 seconds
```

```
job_id: 7
```

```
restarted: False
```

```
labels:
```

```
error: False
```

```
message: None
```

```
load-balancer:
```

```
services:
```

```
darkhttpd:
```

```
error: False
```

```
message: (project:hello-http, service:darkhttpd, backend:web, serverid:-fa8f94b1-6b2b-4cb4-a808-e9da46014c86, code:202, transaction_id:b19e7997-871c-4293-a8a9-51ce03f2bbaa, commit:1) server has been successfully updated.
```

```
skydns:
```

```
services:
```

```
darkhttpd:
```

```
error: False
```

```
message: None
```

```
datacenter: http://127.0.0.1:8888
```

```
entrypoint: main
```

```
chain: centralita
```

```

labels:
  - all
  - centralita
  - services
projects:
  hello-http:
    state: DONE
    last_log: 2025-04-23_20h04m25s
    locked: False
    services:
      - {'name': 'darkhttpd', 'status': 0, 'jail': 'a6549318ce'}
    up:
      operation: COMPLETED
      output:
        rc: 0
        stdout: {'errlevel': 0, 'message': None, 'failed': []}
      last_update: 8 minutes and 29.3 seconds
      job_id: 21
      restarted: False
      labels:
        error: False
        message: None
        load-balancer:
          services:
            darkhttpd:
              error: False
              message: (project:hello-http, service:darkhttpd, backend:web, serverid:f-
4b9e170-67bb-403e-88da-112c55b45fce, code:202, transaction_id:00e632ce-c215-4784-9e61-
9507d914ba6a, commit:1) server has been successfully updated.
            skydns:
              services:
                darkhttpd:
                  error: False
                  message: None

```

Without any intervention, let's see the magic.

```

datacenter: http://127.0.0.1:8888
entrypoint: main
chain: None
labels:
  - all
  - desktop
  - services
  - vm-only
projects:
  hello-http:
    state: DONE

```

```

last_log: 2025-04-23_19h36m17s
locked: False
services:
  - {'name': 'darkhttpd', 'status': 0, 'jail': '0524bcf91b'}
up:
  operation: COMPLETED
  output:
    rc: 0
    stdout: {'errlevel': 0, 'message': None, 'failed': []}
  last_update: 14 minutes and 30.7 seconds
  job_id: 32
  restarted: False
  labels:
    error: False
    message: None
    load-balancer:
      services:
        darkhttpd:
          error: False
          message: (project:hello-http, service:darkhttpd, backend:web, serve-
rid:0d67d160-61af-4810-b277-5fb9e20da8eb, code:202, transaction_id:a2ba93d7-6ce6-4a36-aab2-
09be13a00c17, commit:1) server has been successfully updated.
      skydns:
        services:
          darkhttpd:
            error: False
            message: None
datacenter: http://127.0.0.1:8888
entrypoint: main
chain: provider
labels:
  - all
  - provider
  - vm-only
projects:
  hello-http:
    state: DONE
    last_log: 2025-04-23_20h13m37s
    locked: False
    services:
      - {'name': 'darkhttpd', 'status': 0, 'jail': '2c2d22d2a5'}
    up:
      operation: COMPLETED
      output:
        rc: 0
        stdout: {'errlevel': 0, 'message': None, 'failed': []}
      last_update: 27.47 seconds

```

```

    job_id: 8
    restarted: False
    labels:
      error: False
      message: None
      load-balancer:
        services:
          darkhttpd:
            error: False
            message: (project:hello-http, service:darkhttpd, backend:web, serverid:-
fa8f94b1-6b2b-4cb4-a808-e9da46014c86, code:202, transaction_id:98ca3a6a-65e4-450b-a4c3-
4f135e36be37, commit:1) server has been successfully updated.
          skydns:
            services:
              darkhttpd:
                error: False
                message: None
datacenter: http://127.0.0.1:8888
entrypoint: main
chain: centralita
labels:
  - all
  - centralita
  - services
projects:
  hello-http:
    state: DONE
    last_log: 2025-04-23_20h04m25s
    locked: False
    services:
      - {'name': 'darkhttpd', 'status': 0, 'jail': 'a6549318ce'}
up:
  operation: COMPLETED
  output:
    rc: 0
    stdout: {'errlevel': 0, 'message': None, 'failed': []}
  last_update: 9 minutes and 22.37 seconds
  job_id: 21
  restarted: False
  labels:
    error: False
    message: None
    load-balancer:
      services:
        darkhttpd:
          error: False
          message: (project:hello-http, service:darkhttpd, backend:web, serverid:f-

```

```
4b9e170-67bb-403e-88da-112c55b45fce, code:202, transaction_id:00e632ce-c215-4784-9e61-9507d914ba6a, commit:1) server has been successfully updated.
```

```
skydns:
  services:
    darkhttpd:
      error: False
      message: None
```

The service is alive again.

Info, Metrics, and more...

Thanks to [AppJail](#), a lot of information can be obtained from jails. Overlord has a special deployment called **readOnly** that can be perfectly combined with the **get-info** command.

info.yml:

```
kind: readOnly
datacenters:
  main:
    entrypoint: !ENV '${ENTRYPOINT}'
    access_token: !ENV '${TOKEN}'
deployIn:
  labels:
    - all
$ overlord get-info -f info.yml -t projects --filter adguardhome
datacenter: http://127.0.0.1:8888
  entrypoint: main
  chain: centralita
  labels:
    - all
    - centralita
    - services
  projects:
    adguardhome:
      state: DONE
      last_log: 2025-04-07_17h32m40s
      locked: False
      services:
        - {'name': 'server', 'status': 0, 'jail': '2a67806954'}
$ overlord get-info -f info.yml -t jails --filter 2a67806954
datacenter: http://127.0.0.1:8888
  entrypoint: main
  chain: centralita
  labels:
    - all
    - centralita
    - services
  jails:
    2a67806954:
```

```

stats:
  cputime: 91
  datasize: 8400896 (8.01 MiB)
  stacksize: 0 (0 bytes)
  coredumpsize: 0 (0 bytes)
  memoryuse: 75104256 (71.62 MiB)
  memorylocked: 0 (0 bytes)
  maxproc: 4
  openfiles: 296
  vmemoryuse: 1367982080 (1.27 GiB)
  pseudoterminals: 0
  swapuse: 0 (0 bytes)
  nthr: 13
  msgqqueued: 0
  msgqsize: 0
  nmsgq: 0
  nsem: 0
  nsemop: 0
  nshm: 0
  shmsize: 0 (0 bytes)
  wallclock: 363548
  pcpu: 0
  readbps: 0 (0 bytes)
  writebps: 0 (0 bytes)
  readiops: 0
  writeiops: 0
info:
  name: 2a67806954
  network_ip4: 10.0.0.3
  ports: 53/tcp,53/udp,53/tcp,53/udp
  status: UP
  type: thin
  version: 14.2-RELEASE
cpuset: 0, 1
expose:
  - {'enabled': '1', 'name': None, 'network_name': 'ajnet', 'hport': '53', 'jport':
'53', 'protocol': 'udp', 'ext_if': 'tailscale0', 'on_if': 'tailscale0', 'nro': '3'}
  - {'enabled': '1', 'name': None, 'network_name': 'ajnet', 'hport': '53', 'jport':
'53', 'protocol': 'tcp', 'ext_if': 'jext', 'on_if': 'jext', 'nro': '0'}
  - {'enabled': '1', 'name': None, 'network_name': 'ajnet', 'hport': '53', 'jport':
'53', 'protocol': 'tcp', 'ext_if': 'tailscale0', 'on_if': 'tailscale0', 'nro': '2'}
  - {'enabled': '1', 'name': None, 'network_name': 'ajnet', 'hport': '53', 'jport':
'53', 'protocol': 'udp', 'ext_if': 'jext', 'on_if': 'jext', 'nro': '1'}
fstab:
  - {'enabled': '1', 'name': None, 'device': '/var/appjail-volumes/adguardhome/db',
'mountpoint': 'adguardhome-db', 'type': '<volumeufs>', 'options': 'rw', 'dump': '0', 'pass':
None, 'nro': '0'}

```

```
labels:
  - {'value': '1', 'name': 'overlord.skydns'}
  - {'value': 'adguardhome', 'name': 'appjail.dns.alt-name'}
  - {'value': 'tailscale0', 'name': 'overlord.skydns.interface'}
  - {'value': 'adguardhome', 'name': 'overlord.skydns.group'}
nat:
  - {'rule': 'nat on "jext" from 10.0.0.3 to any -> ("jext:0")', 'network': 'ajnet'}
volumes:
  - {'mountpoint': 'usr/local/etc/AdGuardHome.yaml', 'type': '<pseudofs>', 'uid':
None, 'gid': None, 'perm': '644', 'name': 'adguardhome-conf'}
  - {'mountpoint': '/var/db/adguardhome', 'type': '<pseudofs>', 'uid': None, 'gid':
None, 'perm': '750', 'name': 'adguardhome-db'}
```

Future Work

There are more things Overlord can do for you than this document provides, see the [Wiki](#) for more examples.

Overlord is a recent project, there is a lot of room for improvement and future features will be added to improve its usability. If you would like to support the project, please [consider donating](#).

JESÚS DANIEL COLMENARES OVIEDO is a sysadmin, developer and ports committer and is the creator of Appjail and all the projects related to it (Director, Makejails, Reproduce, Overlord, etc.).



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 **freebsd.foundation.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.
freebsd.foundation.org/donate



NETGRAPH FOR THE REST OF US

Making FreeBSD's powerful networking framework accessible to everyone who runs jail and VM guests

BY DANIEL J. BELL

A few years ago, I wrote about leveraging FreeBSD in datacenters as a performant and cost-effective alternative to cloud options using tools in the FreeBSD base. As my infrastructure continued to grow, I explored ways to improve performance and management, and getting my network stack just right was one of my key goals. This led me to trying out Netgraph, and I liked it so much that I created an rc service-based tool to simplify its configuration for basic setups, `ngbuddy(8)`.

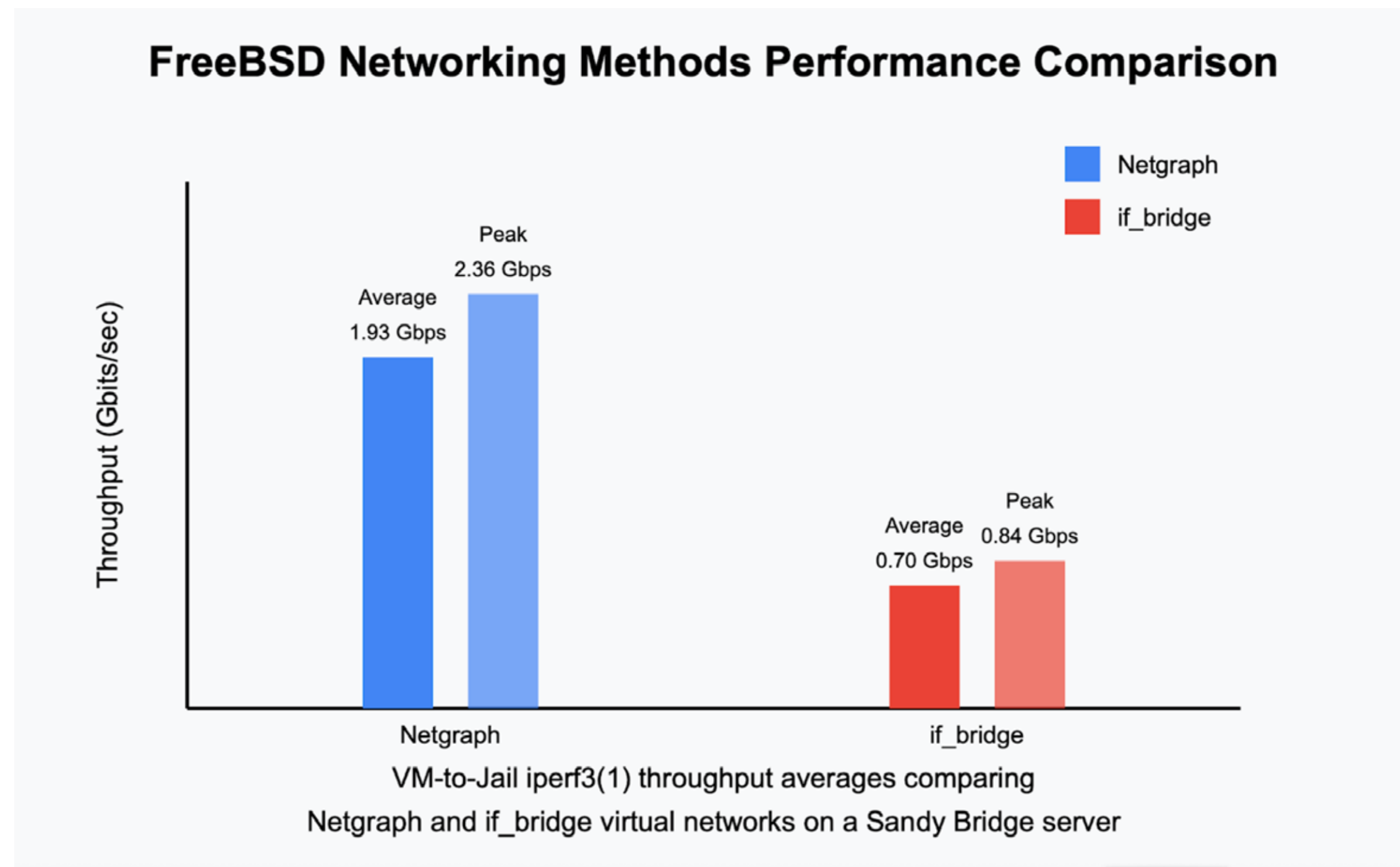
Netgraph, introduced to FreeBSD in 1999, is a networking toolkit that can be used for complex and powerful configurations and specialized networking workflows in a node and link structure. Think of it as a modular building system: nodes (the networking building blocks) connected by hooks (links between nodes or other network objects) to create multiple layers of switches, traffic shapers, or custom virtual networks. Most commonly it's used to hook VM and jail guests to a host's physical or virtual network, but Netgraph can layer these objects in interesting and complex ways. For instance, a FreeBSD pro I met at BSDCan uses a clever Netgraph setup to "mirror" traffic redundantly over multiple physical circuits in a RAID-like fashion, greatly improving stability and resilience for critical VoIP operations.

Most commonly it's used to hook VM and jail guests to a host's physical or virtual network.

I'll introduce the core concepts of Netgraph, explore tools like `ngbuddy` and `jng` that simplify its configuration, walk through manual setup for those who want to understand the underlying mechanics, and provide resources for further exploration. Whether you're new to FreeBSD networking or a know-it-all admin looking to delve deeper, you'll find practical approaches to leverage this powerful framework by giving Netgraph a try.

Why Netgraph Rocks (and Where It Doesn't)

The traditional approach to networking with VMs and jails in FreeBSD typically involves `if_bridge(4)`, `tap(4)`, and `epair(4)` interfaces. This standard choice is great and getting even better; there was a massive `if_bridge` speed improvement in 2020. But less than a year later, FreeBSD 13 introduced Netgraph support in `bhyve`, providing its users with this great alternative. There are good reasons to try Netgraph, but I probably wouldn't have used it if `if_bridge` reliably outperformed it. I ran several simple `iperf3` performance tests between a FreeBSD 14.3 jail and FreeBSD 14.3 `bhyve` VM, comparing both `if_bridge/tap/epair` and pure Netgraph communication, on several of my servers of varying age. Though my results are far from scientific, Netgraph was consistently faster regardless of hardware configuration.



Outside of the ideal conditions above, Netgraph has performance scaling limitations at high packet rates due to a bottleneck in its hook lookup mechanism. There is a promising fix in development that will hopefully make it into FreeBSD 15, but until then, this locking issue makes Netgraph the wrong choice for heavy packet-slinging applications like CDN frontends. However, for the virtualization density we're discussing — a few dozen mixed-use bhyve VMs or jails per host — Netgraph provides consistently reliable performance. In my production environments with similar configurations, I've observed excellent throughput and stability over multiple years of operation.

Netgraph offers other advantages beyond performance:

- **Cleaner configuration:** Fewer visible ifconfig devices and focused commands provide a calmer environment
- **Metrics:** Easy access to traffic statistics for monitoring and troubleshooting
- **Flexibility:** By using Netgraph as your networking suite, the door is opened to its full set of features
- **Visualization:** The graph structure makes it easy to understand your hosts' network topologies, and there are built-in tools to help visualize them

The main drawback of Netgraph is that it can be a bit daunting at first. Netgraph has over 40 FreeBSD manpages for all of its concepts, tools, and kernel modules, so it can be tough to figure out where to get started. However, there are really just a few things to know for most of our setups.

Learn Netgraph Basics: Nodes and Hooks

First, let's get in the mindset of nodes and hooks.

- **Nodes:** These are the processing modules that perform specific networking functions
- **Hooks:** Connection points on nodes that allow data to flow between them

To do basic networking with bhyve and jails, you should know about these node types:

- **ng_bridge:** A virtual switch that connects nodes together for communication
- **ng_ether:** A Netgraph representation of an existing Ethernet interface, e.g., for bridging virtual devices to a physical network
- **ng_eiface:** A virtual Ethernet interface created by Netgraph, also suitable for a jail's interface
- **ng_socket:** A connection for a user process (like bhyve) to interact with Netgraph

The power of Netgraph comes from how these components can be connected. For example, you can create a virtual switch (`ng_bridge`) that connects both physical interfaces (`ng_ether`) and virtual interfaces (`ng_iface`) used by jails or VMs. You can even layer bridges deeply, which is an elegant way to model complex network topologies on a single FreeBSD system.

Simplify Netgraph with ngbuddy

While Netgraph is powerful, the syntax for creating and managing nodes can be a little tricky. Its precision means that you must be explicit about some things that are done automatically when working with `ifconfig(8)` alone. I wrote `ngbuddy` (Netgraph Buddy) to automate this for mixed VM and jail environments and keep our changes persistent on boot.

In the following example, we'll start from a clean FreeBSD 14 install to create and use a "public" Ethernet-connected bridge and "private" host-only bridge, similar to the networking defaults in popular hypervisor managers. We'll also use `vm-bhyve`, which added Netgraph support in July 2022, to further simplify VM management.

While Netgraph is powerful, the syntax for creating and managing nodes can be a little tricky.

Set Up a Basic Virtual Network

We'll mimic the traditional approach to setting up a virtual network for jails or VMs using `if_bridge`, `tap`, and `epair` interfaces. With Netgraph and `ngbuddy`, we'll configure `ng_bridge` nodes instead, linking the "public" bridge to the host's interface that is associated with the default route.

First, install `ngbuddy` from ports:

```
# pkg install sysutils/ngbuddy
```

Then, enable the service to create a basic configuration:

```
# service ngbuddy enable
Adding default bridges.
ngbuddy_public_if: -> ix0
ngbuddy_private_if: -> nghost0
```

This creates two bridges:

- **public:** Connected to your physical interface (`ix0` in this example)
- **private:** Connected to a new virtual interface (`nghost0`) for host-only networking.

If you'd like different bridge names (I prefer more specific names, like "lanbridge"), edit the `ngbuddy` lines in `/etc/rc.conf` to your liking.

Start the service to create the Netgraph bridge nodes:

```
# service ngbuddy start
Created 3 links.
```

At this point, the "public" switch is ready to go and should work like any device connected to your host's network. For the "private" host-only network, you most likely need to configure additional networking components. This usually includes setting up NAT rulesets and a

DHCP server on the nghost0 interface. You can configure nghost0 like any standard interface using ifconfig commands or by adding appropriate entries to /etc/rc.conf.

Configure vm-bhyve

If you're using vm-bhyve 1.5.0 or newer, you can leverage ngbuddy to configure your virtual switches (be careful not to conflict with any existing vm-bhyve bridge names):

```
# service ngbuddy vmconf
```

This adds the following to your vm-bhyve configuration file, your defined \$VM_DIR/.config/system.conf:

```
switch_list="public private"
type_public="netgraph"
type_private="netgraph"
```

With this configuration in place, vm-bhyve is ready to go. When configuring your VMs, just include the proper switch name in each VM configuration, for example:

```
network0_switch="public"
```

Create Interfaces for Jails

Add or replace the following lines in your jail's configuration to automatically create an ng_iface device when the jail starts:

```
my_jail_name {
  if_name = "$name";
  $bridge = "public";
  vnet.interface = "$if_name";
  exec.prestart = "service ngbuddy jail $if_name $bridge";
  exec.prestop = "service ngbuddy unjail $if_name $name";
  ...
}
```

This example uses the jail name variable \$name as the interface name to keep things unique and consistent. You can choose any interface naming scheme that works for your environment, as long as each interface has a unique name.

Monitor Traffic

Monitoring traffic through your Netgraph nodes is simple with ngbuddy, which provides a simplified view of the **ngctl ... getstats** command for all interfaces it detects:

```
# service ngbuddy status
public
  vtnet0 (upper): RX 1.25 MB, TX 4.37 MB
  vtnet0 (lower): RX 4.37 MB, TX 1.25 MB
  jail1: RX 256.32 KB, TX 128.16 KB
private
  nghost0: RX 0B, TX 0B
  jail2: RX 0B, TX 0B
```

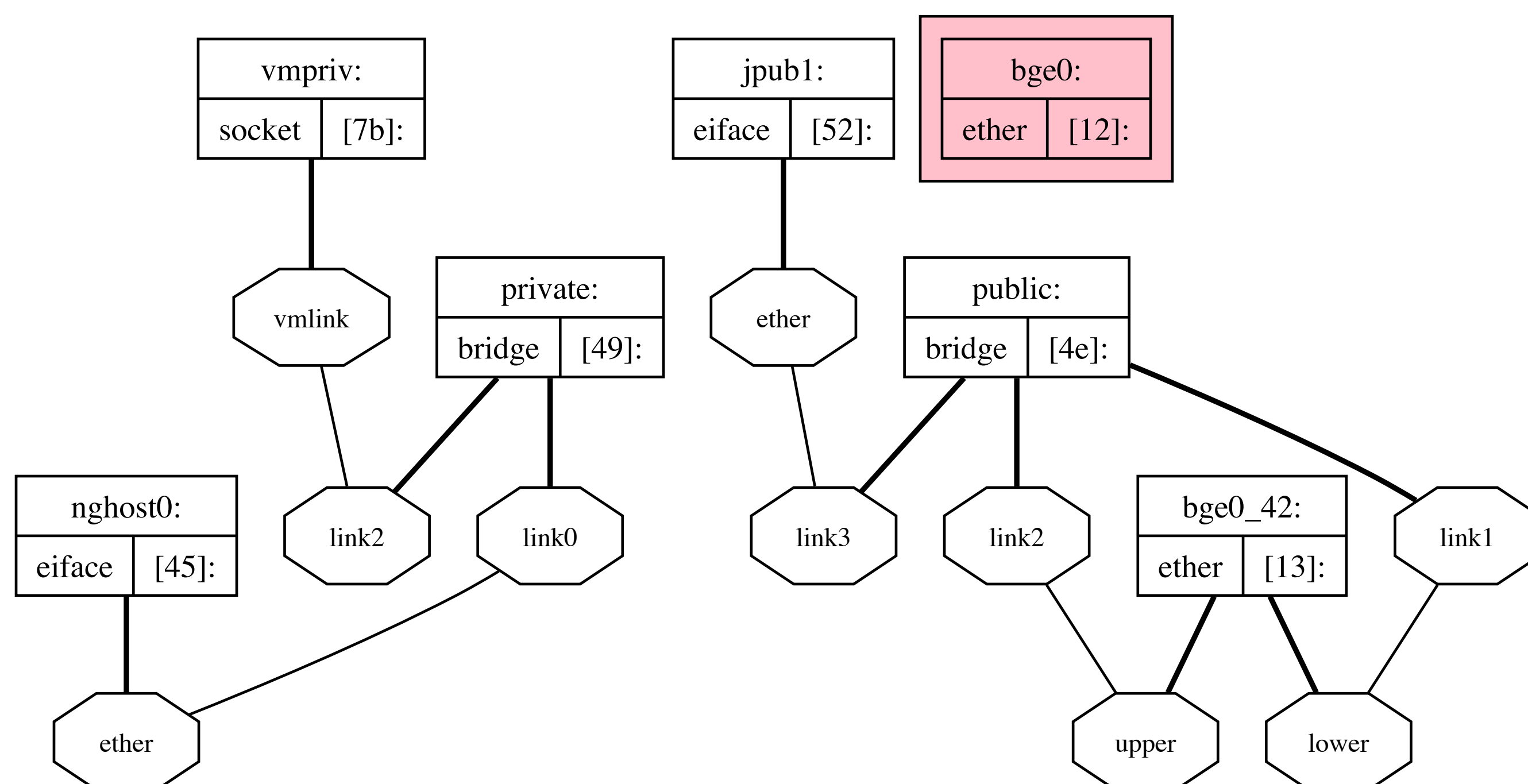
This provides a quick overview of traffic flowing through each interface, which can be invaluable for troubleshooting and performance monitoring. You can also use service **ngbuddy status** or **service ngbuddy vmname** to identify vm-bhyve nodes, which would otherwise appear as “unnamed” ng_socket devices in ngctl output.

Visualize Your Network

We can easily make a picture of the host’s virtual network using the ngctl dot command which produces directed graph text output that looks like this:

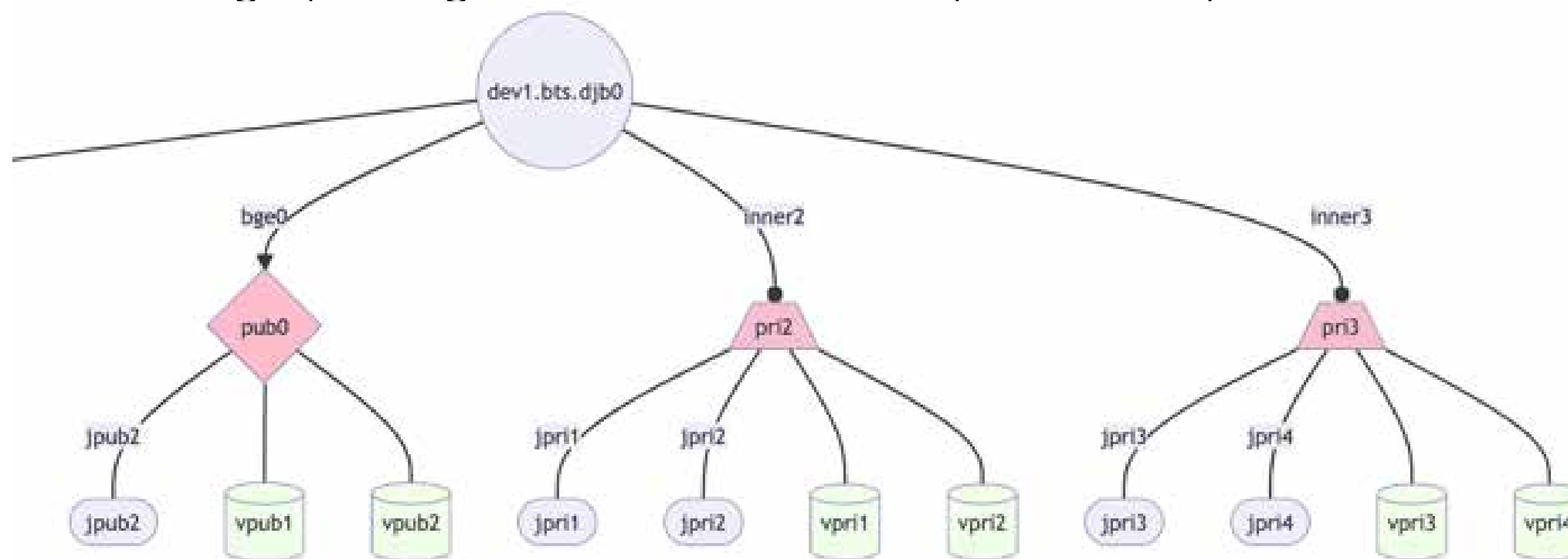
```
graph netgraph {
  edge [ weight = 1.0 ];
  node [ shape = record, fontsize = 12 ] {
    "45" [ label = "{nghost0:|{eiface|[45]:}}" ];
    "49" [ label = "{private:|{bridge|[49]:}}" ];
    "4e" [ label = "{public:|{bridge|[4e]:}}" ];
    "52" [ label = "{jpub1:|{eiface|[52]:}}" ];
    "12" [ label = "{bge0:|{ether|[12]:}}" ];
    "13" [ label = "{bge0_42:|{ether|[13]:}}" ];
    "7b" [ label = "{vmpriv:|{socket|[7b]:}}" ];
  };
  subgraph cluster_disconnected {
    bgcolor = pink;
    "12";
  };
  node [ shape = octagon, fontsize = 10 ] {
    "45.ether" [ label = "ether" ];
  };
  ...
```

This text can be converted directly into a visual SVG or PNG graph using the dot(1) command from the graphics/graphviz port. Here’s a cleaned-up output example of a simple Netgraph setup.



Though this output is great, ngbuddy includes a helper script called ngbuddy-mmd.awk which I wrote to turn this output into colorful and cleaner mermaid-js visualizations. My

company's documentation system supports mermaid-js, so with a bit of API magic I always have up-to-date Netgraph diagrams of all hosts in my fleet in my documentation database.



This diagram includes the hostname at the top, shapes for public and private bridges in the middle, and jails (ovals) and VMs (cylinders) at the bottom. These visualizations are particularly useful when you're running servers with various tenant networks and different functions.

Manage MAC Address Collisions

Regardless of which virtual networking technology I use, I've had terrible luck with MAC collisions. Even in my home lab with 5 or fewer FreeBSD hosts per network segment, I've had afternoons of scratching my head until discovering that some jails were fighting for the same MAC address. To help mitigate this issue and to ensure deterministic MAC address creation, which is very important for guest migrations, you can run the following on all of your ngbuddy hosts.

```
sysrc ngbuddy_set_mac=YES
sysrc ngbuddy_set_mac_prefix=02
```

This configuration ensures that ngbuddy will assign unique MAC addresses to interfaces it creates, using your specified prefix. The "02" prefix indicates a locally administered address, which is appropriate for virtual interfaces in an environment you control.

See the ngbuddy manpage for additional tips and features.

Use jng: No Ports Required

The jng tool remains the simplest way to get started using Netgraph with jails, and it comes built into FreeBSD. The script intelligently creates bridges when needed, which is perfect for many jail host setups. To get started, copy jng into your preferred directory:

```
install -m 755 /usr/share/examples/jails/jng /usr/local/sbin/jng
```

In your jail's configuration, we can invoke jng like the ngbuddy example. Note that jng needs the name of the physical interface you'd like to bridge.

```
my_jail_name {
    $if_uplink="em0";
    $if_name = "ng0_$name";
    vnet.interface = "$if_name";
    exec.prestart += "jng bridge $if_name $if_uplink";
    exec.poststop += "jng shutdown $if_name";
    ...
}
```

If you're interested in writing your own networking workflow, be sure to read the beautifully documented jng script.

Implement Netgraph the Old-Fashioned Way

Now it's time to explore why Netgraph configuration can be annoying and see what the above tools are doing under the hood. FreeBSD virtual networking is mostly done with ifconfig, and those objects are revealed there. With Netgraph, some of these objects are only visible with ngctl and other Netgraph-aware tools, which can be a bit confusing. Also, unlike working with ifconfig, since Netgraph hooks can have complex relationships with each other, we'll have to keep track of our hooks. Let's get started by manually creating the "private" and "public" ng_bridge examples we discussed before plus a host-only Netgraph ng_eiface device to communicate between the host and the virtual "private" bridge. In our example, we'll use "em0" as our real interface that we'd like to bridge.

```
# Load the Netgraph kernel modules
kldload ng_ether ng_bridge

# Create an ng_bridge and call it "private"
ngctl mkpeer ngeth0: bridge ether link0
ngctl name ngeth0:ether private

# Share traffic on em0 with linked Netgraph nodes
ngctl msg em0: setpromisc 1

# Create an ng_bridge device and name it "public"
ngctl mkpeer em0: bridge lower link1
ngctl name em0:lower public
ngctl connect em0: public: upper link2

# Turn off performance features that virtual switches hate
ifconfig em0 -lro -tso

# Create an virtual ethernet device ngeth0 (this will appear in ifconfig)
ngctl mkpeer eiface ether ether
```

The gist here is that we must set up each object, wire it with a unique link#, and then name it if we want any hope of keeping track of it. Let's make another ng_eiface device for a jail.

```
# Create a new interface, then print its name
ngctl mkpeer public: eiface link3 ether
ngctl show -n public:link3 | cut -w -f3
```

Our new interface in this example will be ngeth1. We can configure it with ifconfig as normal. If our new interface becomes a VNET jail, it will still be able to communicate with our network via its Netgraph link with its bridge.

Though it probably doesn't take more keystrokes than standard FreeBSD virtual networking, Netgraph is definitely less suited for manual tinkering due to its more complex syntax and the need to track link numbers.

Continue Your Netgraph Journey

Netgraph represents one of FreeBSD's hidden gems—a powerful networking framework that, once understood, offers elegant solutions to complex networking challenges. While it may seem intimidating at first, tools like `ngbuddy` and `jng` make it accessible to administrators of all skill levels.

Whether you're managing a small home lab or a complex production environment, Net-graph provides the flexibility and power to create exactly the network topology you need. The performance benefits, cleaner configuration, and visualization capabilities make it worth considering for your virtualization infrastructure. I encourage you to explore this powerful subsystem and discover how it can simplify your virtual networking setup.

Resources

- [Netgraph man page](#) (man 4 netgraph)
- [ngbuddy on GitHub](#) (includes full jail.conf examples)
- [FreeBSD Handbook: Jails and Containers](#)
- [FreeBSD Handbook: Virtualization](#)
- [Call For Testing: Jail & bhyve Users' Calls](#)

DANIEL J. BELL is the founder of Bell Tower, a hybrid cloud service provider in New York City focusing on cloud efficiency, regulatory compliance, and advanced storage using ZFS. Daniel has contributed to open-source projects and given conference talks, including talks on ZFS and Netgraph at BSDCan 2024.

Write For Us!

Contact Jim Maurer
with your article ideas.
(maurer.jim@gmail.com)





FreeBSD WiFi Development

Part 1 – Experimenting with WiFi

BY TOM JONES

I have been working on WiFi in FreeBSD for the last 6 months through a project sponsored by the FreeBSD Foundation. The main result of this has been a port from OpenBSD of the `ixw` driver for Intel 80211ac/ax cards. Through this project, I have been exposed to most of the WiFi stack, and it has made clear to me that we need many more people working on WiFi and an easier route to starting to do development.

WiFi arrived 25 years ago and has become so core to our lives that people's first question when entering a new place is "Is there WiFi?"

FreeBSD lacks WiFi drivers. The last 5 years have seen the introduction of a `linuxkpi` layer with support for Wifi and by 2025 these drivers are starting to manage IEEE 802.11ac speeds in the hundreds of megabits. WiFi 7 devices on the market today do 2Gbit.

FreeBSD is behind, there is no argument. We need more people to work on WiFi. If you are interested in doing operating system development, I can think of nothing better than getting involved with improving the FreeBSD WiFi stack. It isn't going to be the easiest thing to do — but the thing about being behind is that we have a lot of open tasks of all difficulties and we need more contributors to catch up.

This series explains a path to working on WiFi in FreeBSD. The raw nuts and bolts of building drivers and watching LLVM warm up your room are covered well elsewhere. In this first article of three, I will explain terms and demonstrate how to configure WLAN interfaces on FreeBSD for testing. This minimum setup is enough to start finding bugs. In following articles I will discuss what WiFi drivers do and how they interact with the larger `net80211` stack in FreeBSD.

Terminology

Networking is about communication - communication is the effective transmission and accurate reception of a message. I've worked in developing and writing internet standards for a decade, and still, for communications experts we aren't the best at completing the communication story. From an IETF background, WiFi is that weird IEEE stuff concerned with making electrons dance.

WiFi arrived 25 years ago
and has become so core
to our lives that people's
first question when
entering a new place is
"Is there WiFi?"

To communicate well, we need to agree on what terms mean, which reminds me of a conversation in the University of Oslo Cafeteria many years ago with a German colleague.

Me: *I'm pretty sure it is why-phi you know, wireless fidelity, like high fidelity.*
Him: *No, it is wee-fee like hee-fee*

I am going to use a lot of acronyms and terms in these articles; it just can't be helped. Searching the Internet is your friend when it comes to figuring these out. Of the two I just used IETF (Internet Engineering Task Force) and IEEE (Institute of Electrical and Electronics Engineers) only IEEE will feature heavily here. Sadly throughout documentation and code there are variations of IEEE, net80211, and IEEE80211 repeatedly, there are a variety of terms used in the FreeBSD source code for WiFi infrastructure.

The WiFi Alliance (the people who sell certification) has made a mess between the standards (from IEEE) and the brand names. Using the standards is more correct, **IEEE80211n** (or **11n**, or **n**) is clearer for me than using WiFi 4. Either term might help you figure out what things are called, what a product does, or what someone is asking you about. FreeBSD is going to tend toward standard names, or even document revisions if we are lucky, rather than the marketing names.

Getting up to speed with concepts in WiFi is also going to be as difficult as it is with names. I suggest some reading, Matthew Ghast's first book on WiFi (*802.11 Wireless Networks*) is a great introduction to all the main concepts at a suitable introductory level. Don't mind the age, the foundations are still the same, the numbers are just bigger, and the modulations are more complex.

.....

The WiFi Alliance
(the people who sell
certification) has made
a mess between the
standards (from IEEE)
and the brand names.

WiFi	A collection of standards by the IEEE and marketing names by the WiFi alliance. Used colloquially it is "that thing your laptop uses to go on the internet" and that is good enough for us. Anyone correcting your terminology too intensely isn't your friend.
IEEE80211	IEEE 802.11 is the family of standards that defines WiFi.
net80211	Or the stack. The code in FreeBSD that implements the IEEE80211 state machine.
Band	A frequency range that a client or access point may use. (these are fractal 2.4GHz 'band', 2462 MHz band)
Channel	An RF frequency and parameters. Sometimes used interchangeably with 'band'
Station	Your device, other clients on a network (also a mode)
Access Point	The device running the network you connect to (also a mode)
Monitor	A mode which causes the network adapter to capture all packets on a band.

Network Adapter	The device with all the radios which enables you to do WiFi (also, a network card – though a USB stick isn’t a card, might be a network interface, but best to avoid confusion between the hardware and the software model).
Driver	Code in FreeBSD that speaks to a network card either directly or via firmware
Firmware	Code that runs on the network card and does some of the work for you. We normally only mention it when it needs to be loaded by the OS or a driver.
MAC	The part of the 80211 machine which handles sharing the medium
Full MAC	Used in reference to drivers – full mac devices implement the MAC layer and much of the work that net80211 does can be skipped.
HT	High Throughput (also called 80211n)
VHT	Very High Throughput (also called 80211ac)
EHT	Extremely High Throughput (also called 80211ax)

There are also some core concepts you need to be familiar with for this article. I think, for most people, this is how their day-to-day network access works, but it is good to recognize that technologies some of us grew up with (me) or were introduced in our lifetimes (a bunch of people I’ve just made feel old), have existed for a decade longer than some readers have been alive.

The most common WiFi network found in people’s homes is an access point (AP), which acts as the gateway to the larger Internet for stations (or clients). In many deployments, like your home, the access point is a router that handles forwarding traffic from clients to the internet (maybe via a modem), assigning addresses, and a host of other network tasks.

To join a network, a station goes through a series of states while communicating with the access point. In summary, it will:

- scan
- probe
- receive beacons
- authenticate
- associate
- negotiate encryption keys
- acquire an IP address

All but the final steps here are WiFi-specific, in some ways they model you finding a cable, a port, and plugging your computer into the router. The final step happens at the IP layer, and we use the same tools as in wired networks.

All these phases and states are handled in the **net80211** stack and device drivers. Who does what depends on hardware support, some features are handled by firmware on the network adapter. When they can’t be done in hardware, the **net80211** stack can implement most things itself, some features that use radio functionality can’t be emulated in software.

As we do WiFi development on FreeBSD, we need to be aware of what the hardware and **net80211** layers are doing. Above is a short summary, but there are many other states and authentication modes. **IEEE80211** is approaching 30 and it has a lot of history.

Experimenting with WiFi on FreeBSD

Before we start reading code and making changes, we should discuss how to do management operations on WLAN adapters in FreeBSD.

The **net80211** stack provides an abstraction on top of network adapters which gives us virtual interfaces. Throughout the code, they are called VAPs (see the `ieee80211_vap` man page for a full description of their functionality).

This means that we first must create a WLAN interface from a device before we can use it.

This may seem clunky compared to the ease of management of an interface in OpenBSD (`ifconfig iwx0 up`), but it enables virtual functionality to be implemented on top of a single adapter. If the hardware supports it, you might be able to be an access point and a station at the same time, or a station and in monitor mode simultaneously.

Usually, this management is handled for you by some configuration in `rc.conf`, the installer adds lines like these:

```
wlans_iwlfwifi0="wlan0"
ifconfig_wlan0="WPA SYNCDHCP"
```

The first line tells the **rc** system to create an interface from the **iwlfwifi0** adapter called **wlan0**, the second line is a usual **ifconfig** line you would see for a wired interface.

Kernel development can be helped by controlling the creation of devices so let's first look at how we can manually create interfaces.

Manually Creating Interfaces

WLAN devices that are registered with **net80211** as devices can be listed by reading the **net.wlan.devices sysctl**.

```
$ sysctl net.wlan.devices
net.wlan.devices: iwx0 rtwn0
```

On my laptop, you can see I have an **iwx**-based card attached (**iwx0** on PCIe) and a **rtwn** card (**rtwn0** on USB).

From these devices, I can create interfaces using `ifconfig` like so:

```
# ifconfig wlan create wlandev iwx0
wlan0
# ifconfig wlan create wlandev rtwn0 wlanmode ap
wlan1
# ifconfig wlan create wlandev rtwn0 wlanmode monitor
wlan2
```

In the first example, we create **wlan0** without any arguments. Station mode is the default when creating a device without any `wlanmode` argument. WLAN devices in FreeBSD operate in one and only one mode set at creation time, the possible modes for any given device are governed by hardware and driver support.

The modes a driver supports are listed on the corresponding man page. If we compare **iwm**, **iwlfwifi**, and **iwx** (which support some of the same hardware) we will see that **iwm** and **iwlfwifi** at the time of writing can only operate in station mode, whereas **iwx** can operation in station and monitor mode. The Intel hardware that **iwx** supports can work as a host AP, but in a very limited way (only on the 2.4GHz band) and so support isn't yet implemented.

The **rtwn** driver supports station, adhoc, host AP, and monitor mode operation. The full list of modes supported by **net80211** is documented in the `ifconfig(8)` man page:

wlanmode mode

Specify the operating mode for this cloned device. mode is one of sta, ahdemo (or adhoc-demo), ibss (or adhoc), AP (or hostap), wds, tdma, mesh, and monitor. The operating mode of a cloned interface cannot be changed. The tdma mode is actually implemented as an adhoc-demo interface with special properties.

Right after configuration **ifconfig** will show us an interface with lots of information unpopulated:

```
$ ifconfig wlan0
wlan0: flags=8802<BROADCAST,SIMPLEX,MULTICAST> metric 0 mtu 1500
options=0
ether e4:5e:37:af:13:5b
groups: wlan
ssid "" channel 1 (2412 MHz 11b)
regdomain FCC country US authmode OPEN privacy OFF txpower 30
bmiss 10 scanvalid 60 bgscan bgscanintvl 300 bgscanidle 250
roam:rssi 7 roam:rate 1 wme bintval 0
parent interface: iwx0
media: IEEE 802.11 Wireless Ethernet autoselect (autoselect)
status: no carrier
nd6 options=29<PERFORMNUD,IFDISABLED,AUTO_LINKLOCAL>
```

We have quite a few extra parameters in the **ifconfig** output compared to a wired device, let's look at some of these.

```
ssid "" channel 1 (2412 MHz 11b)
```

As we are a station, we have a nominated SSID (network name to connect to), or here nothing. We are on channel 1 with the frequency and the channel mode listed.

```
regdomain FCC country US authmode OPEN privacy OFF txpower 30
```

The regulatory domain and country have been set to a default but should update when we bring the interface up to match my regulator domain and country.

```
bmiss 10 scanvalid 60 bgscan bgscanintvl 300 bgscanidle 250
roam:rssi 7 roam:rate 1 wme bintval 0
```

We have some driver parameters that control scanning, moving between networks, and multimedia extensions (which are used for quality-of-service information not using your adapter to play MP3s).

```
parent interface: iwx0
media: IEEE 802.11 Wireless Ethernet autoselect (autoselect)
```

Finally, we have the parent interface (handy if you are doing a lot of WiFi) and the current media mode. This should normally be autoselect, but you might be forcing this to debug or get better throughput.

If we look at the VAPs created on the rtwn interfaces, they are similar but different, which is due to their different mode.

```
$ ifconfig wlan1
wlan1: flags=8802<BROADCAST,SIMPLEX,MULTICAST> metric 0 mtu 1500
    options=0
    ether 74:da:38:33:c0:62
    groups: wlan
    ssid "" channel 1 (2412 MHz 11b)
    regdomain FCC country US authmode OPEN privacy OFF txpower 30
    scanvalid 60 wme dtimperiod 1 -dfs bintval 0
    parent interface: rtwn0
    media: IEEE 802.11 Wireless Ethernet autoselect <hostap> (autoselect <hostap>)
    status: no carrier
    nd6 options=29<PERFORMNUD,IFDISABLED,AUTO_LINKLOCAL>

$ ifconfig wlan2
wlan2: flags=8802<BROADCAST,SIMPLEX,MULTICAST> metric 0 mtu 1500
    options=0
    ether 74:da:38:33:c0:62
    groups: wlan
    ssid "" channel 1 (2412 MHz 11b)
    regdomain FCC country US authmode OPEN privacy OFF txpower 30
    scanvalid 60 wme bintval 0
    parent interface: rtwn0
    media: IEEE 802.11 Wireless Ethernet autoselect <monitor> (autoselect <monitor>)
    status: no carrier
    nd6 options=29<PERFORMNUD,IFDISABLED,AUTO_LINKLOCAL>
```

When we are done, or if we accidentally created a device in the wrong mode, we could remove it using `ifconfig`:

```
# ifconfig wlan0 destroy
```

Using Interfaces

I lied earlier, well sort of. The FreeBSD WiFi stack has two main components, but a lot of WiFi state is driven by two userspace programs, `wpa_supplicant` and `hostapd`.

Station mode with `wpa_supplicant`

`wpa_supplicant` was originally a program for managing WPA (wireless protected access) encryption state for a device in station mode. It has grown to be a full userspace WiFi management interface. It is one of a few implementations and it is common to see `wpa_supplicant` used on Linux for wireless configuration.

`hostapd` is the host AP userspace daemon from the same project — it corresponds to `wpa_supplicant`, but rather than doing client tasks it does host tasks.

We can use `ifconfig` to manage a WLAN interface, the following command will bring up our station interface and configure it to join the “Test” SSID.

```
# ifconfig wlan0 ssid "Test" up
```

This allows a FreeBSD station to start trying to associate with an AP called "Test". The FreeBSD **net80211** stack doesn't directly support the WPA state machine, so for most networks you will need to use **wpa_supplicant**. The second line from our example `rc.conf` above handles starting **wpa_supplicant** on an interface (and enabling DHCP).

To manually run **wpa_supplicant** we need a configuration file with our networks, the **wpa_supplicant.conf** man page has some great examples, but a bare minimum configuration file looks like this:

```
ctrl_interface=/var/run/wpa_supplicant
ctrl_interface_group=wheel

network={
    ssid="Open Network"
    key_mgmt=NONE
}
```

We can then start **wpa_supplicant** manually like so:

```
# wpa_supplicant -i wlan0 -c /etc/wpa_supplicant.conf -D bsd
```

The default invocation will run **wpa_supplicant** in the foreground and adding **-B** will run it in the background. The logged messages from **wpa_supplicant** are available via the control interface.

The **wpa_passphrase** can be used to add a WPA-protected network to the **wpa_supplicant** configuration file:

```
$ wpa_passphrase "Closed Network" superpassword
network={
    ssid="Closed Network"
    #psk="superpassword"
    psk=852c26a07d84c48e4bfeec71289214a39bcd9d881bc66aedef6a2d11372f59752
}
```

It only generates the required configuration to add to **wpa_supplicant.conf**, to not have to learn a lot of configuration syntax.

FreeBSD ships with **wpa_cli** to interface with **wpa_supplicant**. From **wpa_cli** we can list networks, connect (select), reconfigure, and disconnect. Here is an example session joining a WPA-protected network.

```
$ wpa_cli
wpa_cli v2.11
Copyright (c) 2004-2024, Jouni Malinen <j@w1.fi> and contributors

This software may be distributed under the terms of the BSD license.
See README for more details.

Selected interface 'wlan0'

Interactive mode
```

```

> list_networks
network id / ssid / bssid / flags
0      Open Network      [DISABLED]
1      Closed Network     [DISABLED]
> select_network 1
OK
<3>CTRL-EVENT-SCAN-RESULTS
<3>WPS-AP-AVAILABLE
<3>Trying to associate with 20:05:b6:fa:13:f1 (SSID='Closed Network' freq=5180 MHz)
<3>Associated with 20:05:b6:fa:13:f1
<3>WPA: Key negotiation completed with 20:05:b6:fa:13:f1 [PTK=CCMP GTK=CCMP]
<3>CTRL-EVENT-CONNECTED - Connection to 20:05:b6:fa:13:f1 completed [id=0 id_str=]
disable_network disconnect
> disconnect
OK
<3>CTRL-EVENT-DISCONNECTED bssid=20:05:b6:fa:13:f1 reason=3 locally_generated=1
<3>CTRL-EVENT-DSCP-POLICY clear_all

```

AP mode with hostapd

hostapd is less used, and the ability to bring up your own AP in which you can do full kernel debugging and packet capture can be a very helpful debugging tool.

An example configuration for our example "Closed Network" might look like the following:

hostapd.conf:

```

ctrl_interface=/var/run/hostapd
ctrl_interface_group=wheel

interface=wlan1

# hw_mode=g
channel=8
ieee80211d=0
ieee80211n=0
wmm_enabled=0

# the name of the AP
ssid="Closed Network"
# 1=wpa, 2=wep, 3=both
auth_algs=1
wpa=2
wpa_key_mgmt=WPA-PSK
rsn_pairwise=CCMP
wpa_passphrase="superpassword"

```

We can run **hostapd** in the foreground like so:

```
# hostapd hostapd.conf
```

With hostapd running we have most of an access point — the WiFi bit is easy! We also need to give `wlan1` an address and usually a process runs to provide dynamic addressing.

We can give the interface an IP address like normal:

```
# ifconfig wlan1 inet 192.168.2.1/24 up
```

For dynamic addressing, we need to install the `dhcpcd` port and create a configuration file. A minimal configuration file can be found in the `dhcpcd.conf(5)` man page and might look like this:

```
/usr/local/etc/dhcpcd.conf:
```

```
subnet 192.168.2.0 netmask 255.255.255.0 {
    range 192.168.2.100 192.168.2.200
}
```

We can then enable and start `dhcpcd`:

```
# service enable dhcpcd
# service start dhcpcd
```

Monitor mode

The final mode VAP we created in our example was a monitor mode VAP. While we can capture on VAPs in other modes, the interface is not run in promiscuous mode, the packets we receive are those that arrive with an address for our interface. Monitor mode allows us to receive all the packets on a channel (hardware support depending on different rates).

A simple proof of concept is to use `tcpdump` with the `-y` link level headers flag.

```
# tcpdump -L -i wlan2
Data link types for wlan0 (use option -y to set):
  EN10MB (Ethernet)
  IEEE802_11_RADIO (802.11 plus radiotap header)
```

I struggle to remember the order of the underscores in this variable, but `tcpdump` will show you the supported link-level headers for an interface type with the `-L` flag and the interface.

```
# sudo tcpdump -i wlan2 -y IEEE802_11_RADIO
tcpdump: data link type IEEE802_11_RADIO
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on wlan2, link-type IEEE802_11_RADIO (802.11 plus radiotap header), snapshot
length 262144 bytes
14:33:48.656430 3757399us tsft 1.0 Mb/s 2412 MHz 11g -76dBm signal -95dBm noise Data
IV:c1ed Pad 20 KeyID 1
14:33:50.657087 5759270us tsft 1.0 Mb/s 2412 MHz 11g -72dBm signal -95dBm noise
14:33:50.796280 5895802us tsft 1.0 Mb/s 2412 MHz 11g -76dBm signal -95dBm noise Beacon
(HomeWifi) [1.0* 2.0* 5.5* 11.0* 6.0 9.0 12.0 18.0 Mbit] ESS CH: 2, PRIVACY
14:33:53.151514 8251009us tsft 1.0 Mb/s 2412 MHz 11g -74dBm signal -95dBm noise Beacon
(HomeWifi) [1.0* 2.0* 5.5* 11.0* 6.0 9.0 12.0 18.0 Mbit] ESS CH: 2, PRIVACY
14:33:53.970729 9070213us tsft 1.0 Mb/s 2412 MHz 11g -74dBm signal -95dBm noise Beacon
(HomeWifi) [1.0* 2.0* 5.5* 11.0* 6.0 9.0 12.0 18.0 Mbit] ESS CH: 2, PRIVACY
```

```

14:34:10.183336 25285260us tsft 6.0 Mb/s 2437 MHz 11g -62dBm signal -95dBm noise Beacon
(a2-enc) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS CH: 6, PRIVACY
14:34:10.204045 25306099us tsft 11.0 Mb/s 2437 MHz 11g -68dBm signal -95dBm noise Beacon ()
[1.0* 2.0* 5.5* 11.0* 6.0 9.0 12.0 18.0 Mbit] ESS CH: 6
14:34:10.253438 25356305us tsft 11.0 Mb/s 2437 MHz 11g -58dBm signal -95dBm noise Beacon ()
[1.0* 2.0* 5.5* 11.0* 6.0* 9.0 12.0* 18.0 Mbit] IBSS CH: 6, PRIVACY
14:34:10.253441 25356305us tsft 11.0 Mb/s 2437 MHz 11g -58dBm signal -95dBm noise Beacon ()
[1.0* 2.0* 5.5* 11.0* 6.0 9.0 12.0 18.0 Mbit] ESS CH: 6
14:34:10.253445 25356305us tsft 11.0 Mb/s 2437 MHz 11g -58dBm signal -95dBm noise Beacon
(HM-CM-$tste, HM-CM-$tste, kette) [1.0* 2.0* 5.5* 11.0* 6.0 9.0 12.0 18.0 Mbit] ESS CH: 6,
PRIVACY
14:34:10.285355 25387273us tsft 6.0 Mb/s 2437 MHz 11g -63dBm signal -95dBm noise Beacon
(a2-enc) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS CH: 6, PRIVACY
14:34:10.297973 25399988us tsft 11.0 Mb/s 2437 MHz 11g -68dBm signal -95dBm noise Beacon
(HM-CM-$tste, HM-CM-$tste, adkette) [1.0* 2.0* 5.5* 11.0* 6.0 9.0 12.0 18.0 Mbit] ESS CH: 6,
PRIVACY
14:34:10.355834 25458704us tsft 11.0 Mb/s 2437 MHz 11g -58dBm signal -95dBm noise Beacon ()
[1.0* 2.0* 5.5* 11.0* 6.0* 9.0 12.0* 18.0 Mbit] IBSS CH: 6, PRIVACY
14:34:10.355836 25458704us tsft 11.0 Mb/s 2437 MHz 11g -58dBm signal -95dBm noise Beacon ()
[1.0* 2.0* 5.5* 11.0* 6.0 9.0 12.0 18.0 Mbit] ESS CH: 6

```

This `tcpdump` output is focused on the IEEE80211 radio frames. In-depth analysis will require a different tool such as Wireshark.

Testing Traffic

Now that we have all the pieces to build a purely FreeBSD AP station and debug the traffic from the air, we should talk about how to test things.

We can use our AP station and monitor mode to verify and investigate the packets that are sent during an association and sending traffic.

The first stage of testing is getting the station onto the network to the point where we can send pings to the AP. Using `wpa_supplicant` from the station we can select the network and request that `wpa_supplicant` associate for us. `wpa_supplicant` will print messages as it does this, and it will document the hand packets.

Once on the network, the station needs to request an IP address. If this doesn't happen automatically then running:

```
# dclient wlan0
```

will normally kick it to working. If you get a dhcpd lease (an address) then the next step is to try and ping the AP:

```
# ping 192.168.2.1
```

Any failures here are places to start debugging. You will likely have configuration issues (I can't promise the examples are perfect, but they are tested). Debugging why you can't connect isn't the most enjoyable WiFi development experience, but it is one we have all encountered.

Once we have an address, simple throughput tests are usually a good first metric. If you are testing patches from a branch, it might be enough to build a kernel with the patches and run a network throughput test. I like `iperf3` for this sort of testing, it allows you to test

throughput to a host on your network or the Internet. Traffic from **iperf** on a station to **iperf** on the app will give you a good idea of what throughput is possible with that combination of hardware.

If your system is stable enough and can run a web browser, I find testing against fast.com to be helpful too.

These options test very different limitations, the **iperf** test will give you an understanding of the throughput available to a station on your WiFi network and the fast.com test will tell you what is possible from your network to the internet. You may get a significantly lower number with fast.com. If it is higher than the **iperf** test, then something is funky.

You will see differences in throughputs between TCP and UDP tests. UDP can better saturate WiFi radio. Testing both is good, but for a quick measurement, TCP throughput is fine.

Ready for Work

This article has covered the background and terminology you need to start hacking on WiFi in FreeBSD, but we haven't managed to look at any code. Setting up a test network with the examples here is a core part of doing WiFi development and if you are eager and don't want to wait for Part 2, I am sure that if you try this network with enough hardware, you will start to find bugs.

In the next installment, we will look at the life cycle of a WiFi driver, the core functions it needs to have to do its work, and the interfaces to net80211 that can do a lot of the work from the driver.

TOM JONES is a FreeBSD committer interested in keeping the network stack fast.



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 freebsd.foundation.org

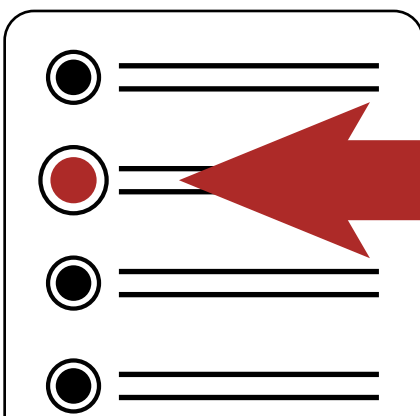
Help Create the Future. Join the FreeBSD Project!

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

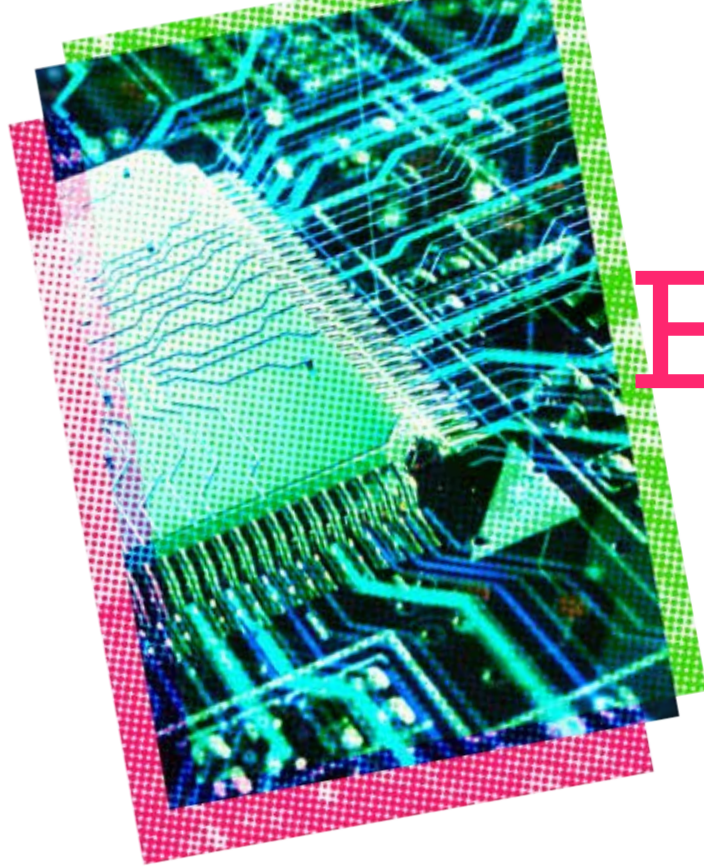
Not only is FreeBSD easy to install, but it runs a huge number of applications, offers powerful solutions, and cutting edge features. The best part? It's FREE of charge and comes with full source code.

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



Contents



Embedded FreeBSD

Embedded FreeBSD: Custom Hardware

BY CHRISTOPHER R. BOWMAN

When I started this journey using FPGAs and FreeBSD, I chose the [Digilent Arty Z7-20](#) because it was among the least expensive of the Zynq-based boards and it had expandability. It has both a set of pins in the physical form factor of the Arduino shield and a set of connectors conforming to the [PMOD](#) standard. In hindsight, I think I would choose a board that sacrificed the Arduino shield connector (which I've never used) for more [PMOD](#) interfaces. There are [lots](#) of PMOD devices and many of them are pretty cheap. For this column, I'm going to use the [PMOD SSD](#).

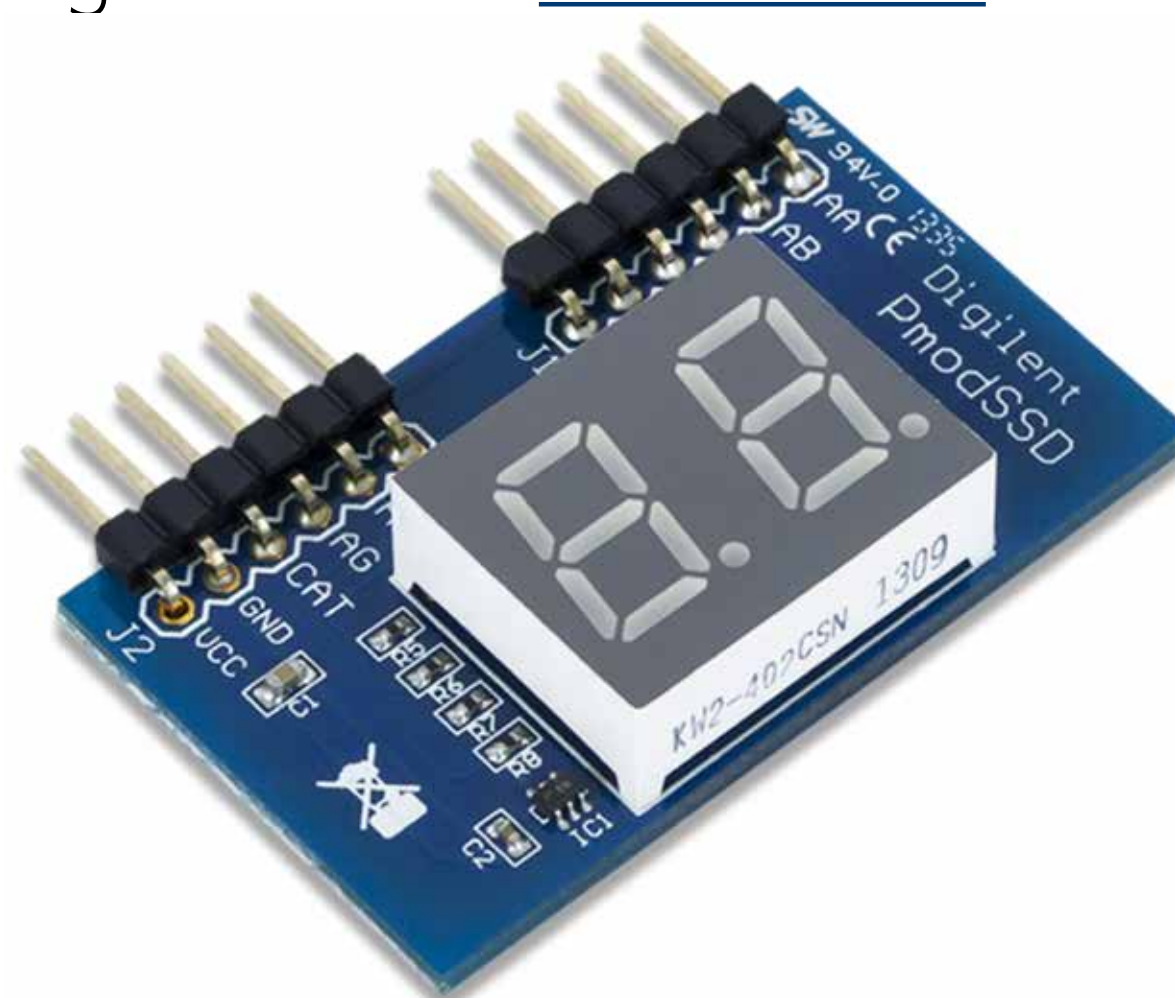


Figure 1 PMOD SSD

This is a dual Seven Segment Display (SSD, not to be confused with a Solid-State Disk) that you can plug into two of the PMOD connectors on the ArtyZ7 board. If we look at the [schematic](#) it's clear that simply setting a couple of pins high or low will configure the LEDs that are lit up, but then there is this line "Because only one digit can be lit at a particular time, users that want to use both digits to display a particular value will need to alternately light up the two digits at least every 20 milliseconds (50 Hz)." You can only have one digit or the other light up at a time. To display from both simultaneously, you need to change the AA-AF pins and then toggle the C pin every 20ms. I haven't tried it but I doubt I'd be able to call the GPIO system 50 times a second reliably. This looks like a good use case for hardware. It's not a difficult circuit. We'll have a pair of registers, and we will switch connecting each register to the pins at 50Hz: two registers, a counter, and some MUXes. Instead of registers, we could use 14 GPIO pins and just put the counter and MUXes in the hardware. That's certainly the quick and simple solution, but at some point, you're going to want to build hardware with a conventional register interface. So, let's build a simple memory-mapped register device and hang it off the AXI bus. It will be a simple use of the AXI bus but it will provide a powerful example for more complex uses. Because of this choice, we'll need to build our own driver, but that will be a good example too.

This project has both hardware and software components as well as documentation. You can get a copy using:

```
# git clone http://github.com/axi_mm_ssd
```

Things are a little more complicated now, some components are meant to be built on a Linux machine or VM (the FPGA bitstream), others are meant to be built under FreeBSD (everything else). I'm using my **bhyve** setup from an earlier column for the Linux side so I can use a single machine and don't have to continuously reboot each time I make a hardware change requiring a rebuild of the FPGA. The hardware should be built under Linux (unless you've gotten the Xilinx/AMD Vivado tool to run native under FreeBSD using Linux emulation, in which case, could you tell me how you did that?) As GNU **make** is the standard **make** under Linux, there is a GNU-style **make** file. You'll also see that the hardware build script is much more complicated.

I constantly struggle to find a good way to script my hardware builds for Vivado. This time I worked out much of what I needed to do in the GUI and used the Vivado command **write_project_tcl** from within Vivado to write out a project creation script which I've modified and augmented. I'm not super happy with it, but it seems to work reliably for me.

Using Vivado's GUI and IP creation I created a simple AXI slave example which helpfully generated Verilog for a complete AXI4-lite memory-mapped slave interface. I took this example and modified it a bit for my purposes adding the counter and MUXes and routing the pins. Then I used the GUI to connect that AXI4-lite slave to the AXI bus and assign it an address using **assign_bd_address**. You can just call **make** which will do all this via a script.

Let's start by looking at the hardware that's going to be built. If you go into the documentation directory in the repo and build the documentation using **make**, you'll see a section describing the hardware interface. Briefly, we're going to build a memory-mapped device with 3 registers. The first will have a constant read-only value that will positively identify the device. I find this useful when you're starting from zero and you're not really sure what you're doing from a hardware or software perspective. If nothing works the way you expect it but you see that magic number in memory at least you know you're on the right track. I'm also looking at ways to use the **git** hash and the build date to tag my designs so that when a circuit loads in the fabric I know exactly which version of my design loaded. Not so much a problem when you're dealing with a fully debugged board from a commercial vendor, but a huge help when you're doing it all by yourself, adding bugs in the hardware and the software both of which you're just learning how to write. Unlike building a chip, we don't have to create a \$5 million set of masks for each hardware spin, so we can afford this overhead and take it out if it's an issue when everything is working. The other two registers will each control the display of a digit, with each having a bit for each segment of its digit. Each digit can have all the segments set or cleared simply by writing a value to the memory address of the register controlling that digit.

Now that we've talked about the hardware and how to build it and since this is the *FreeBSD Journal* let's take a closer look at the software side. In **software/driver/freebsd/kld/ssd.c** you'll find a fairly straightforward KLD that adds a few **sysctl** entries that end up writing the memory-mapped registers which drive the seven-segment displays. This KLD is a fairly straightforward example as you might find in Joseph Kong's excellent ["FreeBSD Device Drivers: A Guide for the Intrepid."](#) There are a couple of interesting things that differentiate this driver from those you might find on ubiquitous AMD64 PC workstations with their standard PCIe bus. Here is the probe function:

```
static int
ssd_probe(device_t dev)
{

//    device_printf(dev, "probe of ssd\n");
    if (!ofw_bus_status_okay(dev))
        return (ENXIO);

    if (!ofw_bus_is_compatible(dev, "crb,ssd-1.0")){
        return (ENXIO);
    }

    //device_printf(dev, "matched ssd\n");
    device_set_desc(dev, "AXI MM seven segment display");
    return (BUS_PROBE_DEFAULT);
}
```

Note the `ofw_*` functions, we talked about these in the last column and discussed how they're used to read properties from the FDT/DTS files. This time around we'll create our own entries in our FDT overlay to describe our device type, address, and register set. Our overlay source is in `software/driver/freebsd/kld/arrtyz7_ssd_overlay.dts` and the interesting part looks like this:

```
&{/axi} {
    axissd: ssd@043c00000 {
        compatible = "crb,ssd-1.0";
        reg = <0x43c00000 0x0004>;
    };
};
```

As we described in the last column, the **compatible** line identifies our device, via the vendor and version separated by a comma. In this case, I built the device, so I used my initials as the vendor. This project is pretty much done so I'm unlikely to modify it, but if I did modify it and produced a V2 or subsequent version, I could allow the driver to differentiate those versions in case the register layout or programming paradigm was different. This one driver could handle multiple different versions of a similar device and would know which was present based on the version in the FDT.

The **reg** line tells the kernel where the device registers are located in physical address space and how much address space is used for device registers. In our case, the device was placed at physical address `0x43c00000` and there are four registers. While the interface only requires three registers and the documentation describes three, the Verilog actually implements four registers.

If we turn our attention to the attach function it looks as follows:

```
static int
ssd_attach(device_t dev)
{
    struct ssd_softc *sc;
```

```

    device_printf(dev, "attaching ssd\n");
    sc = device_get_softc(dev);
    sc->dev = dev;

    int rid;

AXI_MM_SSD_LOCK_INIT(sc);

    /* Allocate memory. */
    rid = 0;
    sc->mem_res = bus_alloc_resource_any(dev,
        SYS_RES_MEMORY, &rid, RF_ACTIVE);
    if (sc->mem_res == NULL) {
device_printf(dev, "Can't allocate memory for \
device\n");
        ssd_detach(dev);
        return (ENOMEM);
    }
#define MAGIC_SIGNATURE 0xFEEDFACE
#ifdef CHECKMAGIC
int32_t value = RD4(sc, AXI_MM_SSD_SGN);
    if (value != MAGIC_SIGNATURE) {
        device_printf(dev, "MAGIC_SIGNATURE 0xFEEDFACE \
not found! value = %x\n", value);
        ssd_detach(dev);
        return (ENXIO);
    }
#endif
    axi_mm_ssd_sysctl_init(sc);
    device_printf(dev, "ssd attached\n");

    return (0);
}

```

This code first initializes a lock. I'll come back and speak about this more later. Next, the driver allocates bus memory. I believe that by setting **rid=0** we're asking the bus DMA system to allocate the first bus resource associated with this device. I haven't actually traced this code, but I think since we have **reg = <0x43c00000 0x0004>**; in our FDT overlay this block of memory is the resource that gets allocated. I'm guessing if we have multiple such lines, they would be successive resource IDs. If you know whether I got this right, I'd be happy to hear.

Since this is the very first device I've built and the first driver I've created for it, I've added some **ifdef**'d code to read the first register and look for my magic number: **0xFEEDFACE**. The RD4 macro is defined in the file like so:

```

#define RD4(sc, off) bus_read_4((sc)->mem_res, (off))

```

This passes the bus resource we received from **bus_alloc_resource_any()** to **bus_read_4()** and returns the value stored in our register at the offset from the resource base. By using the bus resource system, we avoid having to hard code our register addresses in

our driver. This makes it very easy to change the FDT/DTS if I move the hardware address in my design. It also means I don't have to translate the physical address to a kernel virtual address. As I learned the hard way, kernel addresses don't map directly to physical addresses (and the magic number in the first register helped me figure this out.)

With this code, I can be pretty sure that if I read the first register and get back the correct hard-coded value I'm actually talking to the device I designed. When this is your first time with a whole toolchain for building devices and attaching them to the AXI bus and you don't really understand it all, and you didn't know that kernel physical addresses aren't virtual addresses, this can be *incredibly* reassuring. If I was a real software guy, I might even try adding a custom resource in my FDT/DTS overlay that specified the expected value that would read back instead of hardcoding **0xFEEDFACE**. You can see how that could be very useful. A different constant could signify a different version of the device with a different register layout or semantics and this would allow me to verify that the FDT/DTS matched the device the driver is expecting. For now, I'll leave this as an exercise for the reader (patches welcome).

We can now probe and attach to our device and be fairly certain the device is actually there. Let us turn our attention to how we use it. It's not clear to me what the software interface to such a device should look like from user space. The Unix philosophy seems to be that everything is a file, but this device just doesn't seem like a file. I could create a device file in `/dev` that I could open and then I could define **ioctl** functions that would allow me to set the values of the two registers controlling the seven-segment displays. This would allow me to use file permissions to control access to the displays. I chose a different but similar approach. I chose not to use a filesystem device node and instead of **ioctls** I used **sysctls**.

Just before **ssd_attach()** returns it creates two **sysctls**. One gets and sets the value of one of the seven-segment digits and the other **sysctl** sets the value of the other digit. The **sysctls** should show up in the **sysctl** tree as **dev.ssd.0.tens** and **dev.ssd.0.ones**. Depending on your orientation to the board, these may seem reversed to you.

axi_mm_ssd_sysctl_init() registers the two **sysctls**, and the code for one of these is as follows:

```
static int
axi_mm_ssd_proc0(SYSCTL_HANDLER_ARGS)
{
    int error;
    static int32_t value0 = 1;
    struct ssd_softc *sc;
    sc = (struct ssd_softc *)arg1;

    AXI_MM_SSD_LOCK(sc);

    value0 = RD4(sc, AXI_MM_SSD_SR2);

    AXI_MM_SSD_UNLOCK(sc);

    error = sysctl_handle_int(oidp, &value0,
                             sizeof(value0), req);
    if (error != 0 || req->newptr == NULL)
        return (error);
}
```

```

    WR4(sc, AXI_MM_SSD_SR2, value0);

    return (0);
}

```

This code uses the lock created in our attach function and reads from the device register for the first digit using the RD4 macro discussed above. Given I'm only doing a single read or write I'm not sure I need it, but I used the lock to ensure that competing processes trying to set the register will not interfere. Perhaps I should have had the lock wrap the whole procedure, but I'm new to kernel programming and still not sure what functions are safe to call with a lock (If you know I'd love to hear from you). `sysctl_handle_int` will take the old value, publish it back to userland, and return the new value to be written into our register using WR4. John Baldwin has written an excellent [article](#) on the `sysctl` system if you're looking for more information on how it works.

Now that we have a driver that can probe, attach, and set our two hardware registers we just need to build our kld that contains our driver, build our overlay, install it in `/boot/dtb/overlays`, and stitch it into the `loader.rc` script as we did in the last article. Reboot the system, and after it boots, load the bitstream and then the kld, `dmesg` should show you the successful probe messages.

That's all the important pieces of the driver. Let's see how it works in practice. In the git repo, you'll find a small C shell script in `software/app/test.csh`. It looks like this:

```

#!/bin/tcsh
set digits = (126 48 109 121 51 91 31 112 127 115)
set delay = $argv[1]
foreach tens ($digits)
    sysctl dev.ssd.0.tens=$tens
    foreach ones ($digits)
        sysctl dev.ssd.0.ones=$ones
        sleep $delay
    end
end
end

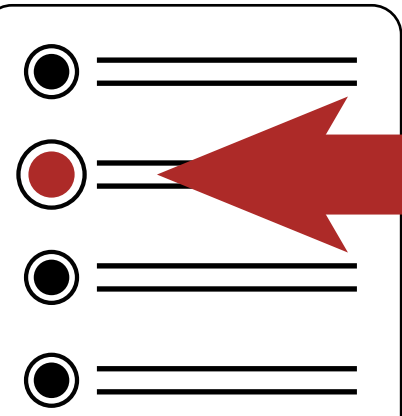
```

There is an array with the values that correspond to the bits that need to be set in one of our registers to encode the numerals 0-9. The script takes a delay value and counts from zero to 99 pausing after each digit changes.

I spent a ridiculous amount of time watching that display count. While in and of itself it's not a hugely useful device, it is a complete soup-to-nuts example of how to communicate to hardware via memory-mapped registers and have that impact on the world outside the chip.

I hope you've found these columns useful. I'd appreciate your comments or feedback. 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.



Static Pacing in FreeBSD

BY RANDALL STEWART AND MICHAEL TÜXEN

Previous articles described FreeBSD's underlying support for pacing via its High Precision Timing System (HPTS) as well as a software pacing methodology within the RACK stack called Dynamic Goodput Pacing (DGP) which will dynamically pace at an optimal rate for the current network condition. There is one more pacing methodology that is provided by the RACK stack that has yet not been described: static pacing.

Static pacing is much simpler and was one of the first pacing methods added to the RACK stack mainly to test out and improve the pacing facilities (i.e. HPTS). Even though it was initially intended as a testing methodology it can, in some instances, be used by an application in specific situations. When using static pacing, the pacing rate is not computed by the RACK stack. Instead, the application must provide the pacing rate by using socket options. Therefore, code in the application is needed to make use of static pacing.

When using static pacing, the pacing rate is not computed by the RACK stack.

Sending of TCP Segments

Three events trigger the sending of TCP segments:

1. The application provides user data to be sent to the TCP stack via **send()** system calls.
2. A TCP timer (for example, the retransmission or delayed acknowledgment timer) runs off.
3. A TCP segment is received.

The number of segments, which can be sent in such an event is mainly controlled by two mechanisms:

1. The flow control protecting a slow receiver against a fast sender. The receiver drives this by advertising the number of bytes the sender is allowed to send in the **SEG.WND** field in the TCP segments sent by the receiver.
2. The congestion control protecting a slow network against a fast sender. The sender drives this by computing the number of bytes, which are allowed to be sent. This number is called the congestion window **CWND**.

The sender only sends what is allowed by the flow control and by the congestion control. There are various algorithms to perform congestion control. One congestion control,

which was the default congestion control in FreeBSD for a long time, is New Reno. New Reno operates in one of two phases:

1. Slow start: this is the initial phase and the one after a timer-based retransmission. In this phase, the **CWND** grows exponentially.
2. Congestion avoidance: this is the phase in which the TCP connection should operate most of the time. In this phase, the **CWND** grows linearly.

At the beginning of a TCP connection, **CWND** is set to the initial congestion window which is controlled by the **sysctl**-variable **net.inet.tcp.initcwnd_segments**. The default is 10 TCP segments.

There are two methods used by a TCP endpoint to retransmit user data, which it considers to be lost. One is after the retransmission timer expires. In the other case, the TCP endpoint operates in recovery, because a loss of TCP segments has been detected.

The above description shows that a TCP endpoint might send bursts of TCP segments when the flow and congestion control allow this.

The following **packetdrill** script illustrates the behavior of FreeBSD when the application provides user data for 10 TCP segments right after the establishment of the TCP connection. **packetdrill** is a testing tool for TCP stacks and its scripts contain the system calls provided by the application and the TCP segments sent and received by the TCP stack. The lines start with a timestamp given in seconds. If the time information starts with a **+**, it is relative to the one before. So, **+0.100** means that the event happens 100 ms after the one before.

There are various algorithms to perform congestion control.

```
--ip_version=ipv4

0.000 `kldload -n tcp_rack`
+0.000 `kldload -n cc_newreno`
+0.000 `sysctl kern.timecounter.alloweddeviation=0`

+0.000 socket(..., SOCK_STREAM, IPPROTO_TCP) = 3
+0.000 setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
+0.000 setsockopt(3, IPPROTO_TCP, TCP_FUNCTION_BLK, {function_set_name="rack",
                                          pcbcnt=0}, 36) = 0
+0.000 setsockopt(3, IPPROTO_TCP, TCP_CONGESTION, "newreno", 8) = 0
+0.000 bind(3, ..., ...) = 0
+0.000 listen(3, 1) = 0
+0.000 < S      0:0(0)                win 65535 <mss 1460,sackOK,eol,eol>
+0.000 > S.     0:0(0)                ack      1 win 65535 <mss 1460,sackOK,eol,eol>
+0.050 < .      1:1(0)                ack      1 win 65535
+0.000 accept(3, ..., ...) = 4
+0.000 close(3) = 0
+0.100 send(4, ..., 14600, 0) = 14600
+0.000 > .      1:1461(1460)         ack      1 win 65535
```

```

+0.000 > . 1461:2921(1460) ack 1 win 65535
+0.000 > . 2921:4381(1460) ack 1 win 65535
+0.000 > . 4381:5841(1460) ack 1 win 65535
+0.000 > . 5841:7301(1460) ack 1 win 65535
+0.000 > . 7301:8761(1460) ack 1 win 65535
+0.000 > . 8761:10221(1460) ack 1 win 65535
+0.000 > . 10221:11681(1460) ack 1 win 65535
+0.000 > . 11681:13141(1460) ack 1 win 65535
+0.000 > P. 13141:14601(1460) ack 1 win 65535
+0.050 < . 1:1(0) ack 2921 win 65535
+0.000 < . 1:1(0) ack 5841 win 65535
+0.000 < . 1:1(0) ack 8761 win 65535
+0.000 < . 1:1(0) ack 11681 win 65535
+0.000 < . 1:1(0) ack 14601 win 65535
+0.000 close(4) = 0
+0.000 > F. 14601:14601(0) ack 1 win 65535
+0.050 < F. 1:1(0) ack 14602 win 65535
+0.000 > . 14602:14602(0) ack 2

```

It is assumed that the round-trip time (RTT) is 50 ms. The script shows that the `send()`-call triggers a burst of 10 TCP segments to be sent at about the same time.

Static Pacing

Static pacing is a method for mitigating the burstiness of traffic. It replaces a large burst with a sequence of smaller bursts. The size of the smaller bursts is called the pacing burst size. The time between these smaller bursts is specified by giving a pacing rate.

Consider a sending rate of 12,000,000 bits/sec, which corresponds to 1,500,000 bytes/sec. When packets with a size of 1500 bytes are sent, this means sending one packet every millisecond. Or sending two packets every two milliseconds, and so on.

By specifying a pacing rate and the pacing burst size, the time between the pacing bursts can be computed such that the sending rate is the given pacing rate.

For static pacing in the RACK stack, the application provides individual pacing rates used for slow start, congestion avoidance, and recovery and the pacing burst size. For the packet size, the size of the IP header, the size of the TCP header, and the TCP payload are considered. The size of link-layer headers and trailers is not considered.

Static pacing is controlled by using `IPPROTO_TCP`-level socket options in the source code of an application. Three of these socket options are used to control the pacing rate in different states of the congestion control algorithm. There is one specific socket option to set the pacing burst size and one to enable and disable the static pacing.

The size of the smaller bursts is called the pacing burst size.

These socket options are shown in the following table.

Socket option name	Type of value	Description
TCP_RACK_PACE_RATE_CA	uint64_t	This option will set the static pacing rate in bytes per second when the congestion control algorithm is in congestion avoidance.
TCP_RACK_PACE_RATE_SS	uint64_t	This option will set the static pacing rate in bytes per second when the congestion control algorithm is in slow start.
TCP_RACK_PACE_RATE_REC	uint64_t	This option will set the static pacing rate in bytes per second when the congestion control algorithm has entered recovery and is working to recover lost packets.
TCP_RACK_PACE_ALWAYS	int	This option enables or disables pacing.
TCP_RACK_PACE_MAX_SEG	int	Sets the pacing burst size.

There are four important notes to make:

1. Setting a socket option (as any other system call) can fail. The application must check for it. One reason that the setting of any of the above socket options fails is that the TCP stack currently used on the socket is not RACK. The default FreeBSD does not support static pacing. In addition to that, enabling static pacing can fail if the overall system limit regarding the number of TCP connections using pacing is reached. This limit is controlled by the `sysctl`-variable `net.inet.tcp.pacing_limit`.
2. When setting the first pacing rate, the rate is not only set for the mode specified by the socket option but for all three modes (congestion avoidance, slow start, recovery).
3. Setting the pacing rate is not enough to enable static pacing. Static pacing must explicitly be enabled by using the `TCP_RACK_PACE_ALWAYS` socket option.
4. When not setting the pacing burst size, the default value of 40 will be used.

The following `packetdrill` script illustrates the pacing of the RACK stack with a pacing rate of 12,000,000 bit/sec in slow start and a pacing burst size of 1.

```
--ip_version=ipv4

0.000 `kldload -n tcp_rack`
+0.000 `kldload -n cc_newreno`
+0.000 `sysctl kern.timecounter.alloweddeviation=0`

+0.000 socket(..., SOCK_STREAM, IPPROTO_TCP) = 3
+0.000 setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
+0.000 setsockopt(3, IPPROTO_TCP, TCP_FUNCTION_BLK, {function_set_name="rack",
                                                                    pcbcnt=0}, 36) = 0
+0.000 setsockopt(3, IPPROTO_TCP, TCP_CONGESTION, "newreno", 8) = 0
+0.000 bind(3, ..., ...) = 0
+0.000 listen(3, 1) = 0
+0.000 < S      0:0(0)                win 65535 <mss 1460,sackOK,eol,eol>
+0.000 > S.    0:0(0)                ack    1 win 65535 <mss 1460,sackOK,eol,eol>
```

```

+0.050 < .      1:1(0)      ack      1 win 65535
+0.000 accept(3, ..., ...) = 4
+0.000 close(3) = 0
+0.000 setsockopt(4, IPPROTO_TCP, TCP_RACK_PACE_RATE_SS, [1500000], 8) = 0
+0.000 setsockopt(4, IPPROTO_TCP, TCP_RACK_PACE_MAX_SEG, [1], 4) = 0
+0.000 setsockopt(4, IPPROTO_TCP, TCP_RACK_PACE_ALWAYS, [1], 4) = 0
+0.100 send(4, ..., 14600, 0) = 14600
+0.000 > .      1:1461(1460) ack      1 win 65535
+0.001 > .    1461:2921(1460) ack      1 win 65535
+0.001 > .    2921:4381(1460) ack      1 win 65535
+0.001 > .    4381:5841(1460) ack      1 win 65535
+0.001 > .    5841:7301(1460) ack      1 win 65535
+0.001 > .    7301:8761(1460) ack      1 win 65535
+0.001 > .    8761:10221(1460) ack      1 win 65535
+0.001 > . 10221:11681(1460) ack      1 win 65535
+0.001 > . 11681:13141(1460) ack      1 win 65535
+0.001 > P. 13141:14601(1460) ack      1 win 65535
+0.042 < .      1:1(0)      ack    2921 win 65535
+0.002 < .      1:1(0)      ack    5841 win 65535
+0.002 < .      1:1(0)      ack    8761 win 65535
+0.002 < .      1:1(0)      ack 11681 win 65535
+0.002 < .      1:1(0)      ack 14601 win 65535
+0.000 close(4) = 0
+0.000 > F. 14601:14601(0)    ack      1 win 65535
+0.050 < F.      1:1(0)      ack 14602 win 65535
+0.000 > .    14602:14602(0)  ack      2

```

The required code changes to enable static pacing are highlighted in yellow and the resulting changes in behavior on the wire are in grey. It should be mentioned that only the timing of the TCP segments has changed, not the TCP segments themselves. There is now a delay of one millisecond between the sending of the outgoing TCP segments containing the user data.

Using the same pacing rate in slow start but a pacing burst size of 4 is illustrated by the following **packetdrill** script:

```

--ip_version=ipv4

0.000 `kldload -n tcp_rack`
+0.000 `kldload -n cc_newreno`
+0.000 `sysctl kern.timecounter.alloweddeviation=0`

+0.000 socket(..., SOCK_STREAM, IPPROTO_TCP) = 3
+0.000 setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
+0.000 setsockopt(3, IPPROTO_TCP, TCP_FUNCTION_BLK, {function_set_name="rack",
                                                    pcbcnt=0}, 36) = 0
+0.000 setsockopt(3, IPPROTO_TCP, TCP_CONGESTION, "newreno", 8) = 0
+0.000 bind(3, ..., ...) = 0
+0.000 listen(3, 1) = 0

```

```

+0.000 < S      0:0(0)                win 65535 <mss 1460,sackOK,eol,eol>
+0.000 > S.     0:0(0)                ack      1 win 65535 <mss 1460,sackOK,eol,eol>
+0.050 < .      1:1(0)                ack      1 win 65535
+0.000 accept(3, ..., ...) = 4
+0.000 close(3) = 0
+0.000 setsockopt(4, IPPROTO_TCP, TCP_RACK_PACE_RATE_SS, [1500000], 8) = 0
+0.000 setsockopt(4, IPPROTO_TCP, TCP_RACK_PACE_MAX_SEG, [4], 4) = 0
+0.000 setsockopt(4, IPPROTO_TCP, TCP_RACK_PACE_ALWAYS, [1], 4) = 0
+0.100 send(4, ..., 14600, 0) = 14600
+0.000 > .      1:1461(1460) ack      1 win 65535
+0.000 > .     1461:2921(1460) ack      1 win 65535
+0.000 > .     2921:4381(1460) ack      1 win 65535
+0.000 > .     4381:5841(1460) ack      1 win 65535
+0.004 > .     5841:7301(1460) ack      1 win 65535
+0.000 > .     7301:8761(1460) ack      1 win 65535
+0.000 > .     8761:10221(1460) ack      1 win 65535
+0.000 > .    10221:11681(1460) ack      1 win 65535
+0.004 > .    11681:13141(1460) ack      1 win 65535
+0.000 > P. 13141:14601(1460) ack      1 win 65535
+0.042 < .      1:1(0)                ack    2921 win 65535
+0.000 < .      1:1(0)                ack    5841 win 65535
+0.004 < .      1:1(0)                ack    8761 win 65535
+0.000 < .      1:1(0)                ack 11681 win 65535
+0.004 < .      1:1(0)                ack 14601 win 65535
+0.000 close(4) = 0
+0.000 > F. 14601:14601(0)            ack      1 win 65535
+0.050 < F.      1:1(0)                ack 14602 win 65535
+0.000 > .     14602:14602(0)          ack      2

```

As expected, four TCP segments are now sent every four milliseconds. Therefore, the timing of the TCP segments highlighted in grey is affected.

Additional Considerations

The last example would thus put in the 12Mbps pacing rate for all congestion control states with a burst size restriction of four segments and then finally turn on pacing. This means that the RACK stack would send four segments, wait for four milliseconds, and then send four more packets and continue repeating that sequence until all packets were sent.

It is important to note that the RACK stack will not wait at a send opportunity for a full four segments, so if there were less than four segments available in the socket buffer, the stack would send out what it could at that time and start an adjusted pacing timer that would space that burst at precisely the pace rate that was requested. Another important consideration is congestion control and flow control; pacing can be slower than the set rate if the stack hits the congestion control or flow control limit. So, the RACK stack may send only one, two, or three TCP segments not due to the lack of user data, but due to the congestion control or flow control restricting how much can be sent. Static pacing may yield reduced throughput due to the inability of it to “make up” for lost bandwidth beyond its set pacing rate after the competing traffic ceases.

There are at least four other interactions to be considered by a developer using static pacing. One is the interaction with Proportional Rate Reduction (PRR). PRR is invoked by the RACK stack whenever the stack is in recovery. In effect, it limits the sending to roughly one segment for every other inbound acknowledgment. This means that the data is sent both subject to a pacing time and subject to whether acknowledgments from the peer are received. In most cases with static pacing, this interaction between static pacing and PRR is not wanted. Therefore, there is an `IPPROTO_TCP`-level socket option `TCP_NO_PRR` to disable PRR. The socket option value is of type `int`.

Yet another recovery action that the RACK stack takes is rapid recovery; this is a feature that allows the RACK stack to recover packets faster based on loss and transmission time and not just three duplicate acknowledgments. However, this may change, during recovery, how fast data is sent and thus modify your expected pacing behavior. The `IPPROTO_TCP`-level socket option `TCP_RACK_RR_CONF` can be used to modify this behavior. Allowable values of type `int` for this socket option are 0, 1, 2, and 3. The default value is 0, which means that the RACK stack has full control to use its rapid recovery feature. The values provide subtle control over recovery by the caller. However, for static pacing, where only the rates provided should be used, the value should be set to 3.

Setting the pacing burst size has a CPU impact when TCP segment offloading (TSO) is enabled. Using smaller pacing burst sizes increases the CPU load. For example, sending 40 TCP segments with a pacing burst size of 40 will only require one TSO operation, whereas a pacing burst size of 4 will require 10 TSO operations.

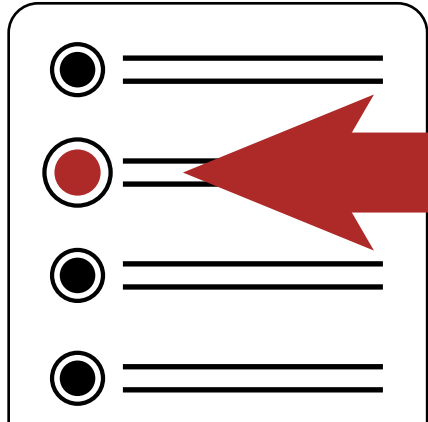
Another thing to keep in mind when setting the pacing burst size is the delay of acknowledgments. Most TCP stacks enable this feature. This means that a TCP stack waits for two (or more) segments or a timeout of the delayed acknowledgment timer before sending out an acknowledgment. The timeout is normally set somewhere between 40 to 200 milliseconds (the specification of delayed acknowledgment calls for 200 but many operating systems have shortened it). So, if the pacing burst size is set to 1 instead of 4, in the above example, an interaction between the delayed ack timer and the pacing timer can happen resulting in pacing at a substantially lower rate than anticipated. Therefore, the pacing burst size should not be set below 2 to avoid such an interaction.

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.

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



Yet another recovery action that the RACK stack takes is rapid recovery.





Events Calendar

BSD Events taking place through November 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.



EuroBSDCon 2025

September 25-28, 2025

Zagreb, Croatia

<https://2025.eurobsdcon.org/>

This yearly conference gives the exceptional opportunity to learn about the latest news from the BSD world, witness contemporary deployment case studies, and personally meet other users and companies using BSD-oriented technologies. EuroBSDCon is also a boilerplate for ideas, discussions, and information exchange, which often become programming projects.



OpenZFS User and Developer Summit 2025

October 25-28, 2025

Portland, OR

https://openzfs.org/wiki/OpenZFS_Developer_Summit_2025

This year, the User Summit will explore a wide range of topics designed to support and connect the OpenZFS community.

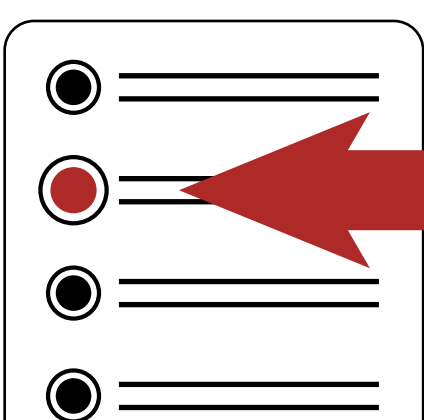


November 2025 FreeBSD Vendor Summit

November 6-7, 2025

San Jose, CA

The Summit provides commercial FreeBSD users with the unique opportunity to meet face-to-face with developers and contributors to get features requested, problems solved, and needs met. It also opens up discussion on improving and enhancing the operating system.



Contents