FreeBSD® JOURNAL

July/August 2022

Science, Systems, and FreeBSD

Building the Loom Framework on FreeBSD

Teaching an Undergraduate Unix Course

Getting Started with FreeBSD Workshop

Pragmatic IPv6 (Part 2)

Advocating for FreeBSD in 2022 and Beyond

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!

2022 Editorial Calendar

- •Software and System Management (January-February)
- •ARM64 is Tier 1 (March-April)
- •Disaster Recovery (May-June)
- •Science, Systems, and FreeBSD (July-August)
- •Security (September-October)
- •Observability and Metrics (November-December)

Find out more at: freebsdfoundation/journal

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

Publisher • Walter Andrzejewski
walter@freebsdjournal.com

Editor-at-Large • James Maurer
jmaurer@freebsdjournal.com

Design & Production • Reuter & Associates

Advertising Sales • Walter Andrzejewski
walter@freebsdjournal.com
Call 888/290-9469

FreeBSD Journal (ISBN: 978-0-615-88479-0) is published 6 times a year (January/February, March/April, May/June, July/August, September/October, November/December).
Published by the FreeBSD Foundation,
3980 Broadway St. STE #103-107, Boulder, CO 80304
ph: 720/207-5142 • fax: 720/222-2350
email: info@freebsdfoundation.org

LETTER from the Foundation

Welcome to the July/August issue of the *FreeBSD Journal*!

Summer is ending and a new school year begins in the northern hemisphere. Brian Kidney opens the issue with an article describing a code tracing and instrumentation framework developed on FreeBSD. Benedict Reuschling writes about his Unix for Developers class that spans topics from system administration to shell scripting, and Roller Angel walks us through an interactive workshop for new users of FreeBSD. Anne Dickison rounds out the articles with her piece on effective FreeBSD advocacy.

You'll also find the second installment of Hiroki Sato's series on using IPv6 on FreeBSD as well as Tom Jones' Work-in-Progress column featuring Gleb Smirnoff's piece on recent changes to socket buffers in FreeBSD 14. And, at long last, this issue includes a couple of conference reports which have been few and far between over the past two years. We look forward to more reports in the future as our community starts to meet at a more regular cadence again. By the way, several members of the Journal's editorial board will be in Vienna, Austria in September for EuroBSDCon 2022. We look forward to talking with readers who will be there and plan a report from Vienna in a future issue for those who won't be able to attend.

We value feedback from readers whether in person or via email. If you have feedback on articles, suggestions for topics for a future article, or are interested in writing for the Journal, please email us at info@freebsdjournal.com.

John Baldwin
Member of the FreeBSD Core Team
and Chair of *FreeBSD Journal* Editorial Board

FreeBSD® JOURNAL®

July/August 2022

Science, Systems, and FreeBSD

Building the Loom Framework on FreeBSD

BY BRIAN KIDNEY

When trying to understand code in production, developers log information to be analyzed offline. This will usually require the decisions up front as to what and when to log. The answers to these questions are usually based on a determination of where things could go wrong or previous issues experience in the system. Knowing during development exactly what information is needed during development is not always possible, especially when trying to trace data for security reasons.

This is part of the problem faced by the Causal Adaptive Distributed, and Efficient Tracing System (CADETS) research project[1]. The goal of the project was to use existing mechanisms as well as develop new techniques to instrument FreeBSD to maximize transparency into live servers. This would give users better insight into security concerns on their systems, as well as provide additional information for performance tuning and debugging.

Loom was one of the tools developed as part of this effort. In this article, we will look at the original inspiration for building Loom. We will talk about how Loom has been expanded beyond its original purpose. Finally we will discuss what we are working on for the future of Loom.

In this article, we will look at the original inspiration for building Loom.

Instrumenting for Security

When developers instrument their software, it is usually to track performance or trace possible points of failure in a system. Similar methodology is often used when adding security monitoring, often capturing only a small number of events such as login attempts with minimal information. Much of this information is logged to different locations on a computer and often is not correlated with other systems. The CADETS team found that this information in the kernel could be obtained easily with the use of DTrace, the tracing framework included as part of the FreeBSD base system. FreeBSD provides many DTrace providers to allow access to information on kernel components such as system calls, function calls and network buffers.

However, extracting additional information from userland processes DTrace requires more work. The issues with tracing programs running in userland is the information captured often lacks context and flexibility. For example, it is easy to write a DTrace script to log all calls to a specific function in a library, but the script has to be applied each process that will use that library. We needed a way to be able to instrument the library itself so that the logging occurs for any program using the library without having to target each program explicitly in the script. The mechanism needed also to provide provide context for each time the library function was called, including function name, arguments and executable calling the function.

DTrace provides a mechanism to add tracing probes directly into programs and libraries via Userland Statically Defined Tracing (USDT). The process for creating these probes requires the developer to write and compile custom code for each probe, include it in the original source and then use a DTrace specific tool to modify the programs binary object files to insert calls to the probes. Unfortunately, this process requires changes to the build process to include an additional tool that modifies your object files directly. Each time the probe code is changed it is converted to a header file that is included in the original code, requiring the program to be completely compiled from scratch. We wanted a simpler system that allows the user to insert instrumentation at the LLVM Intermediate Representation (IR) stage, without the need for a complete rebuild. Since FreeBSD uses LLVM as the system compiler this would make it easier to include our method in the build system.

Our solution to this problem was Loom, a custom LLVM optimization (opt) pass that allowed the user to insert arbitrary code into a program during compile time without modification of the original source code and without having to recompile the entire program each time. As you can see in Figure 1, Loom integrates into the FreeBSD build process as an additional opt pass. The original source code is compiled to LLVM IR first and using the Loom pass and a YAML policy file the additional code is inserted into the program before the final linker stage.

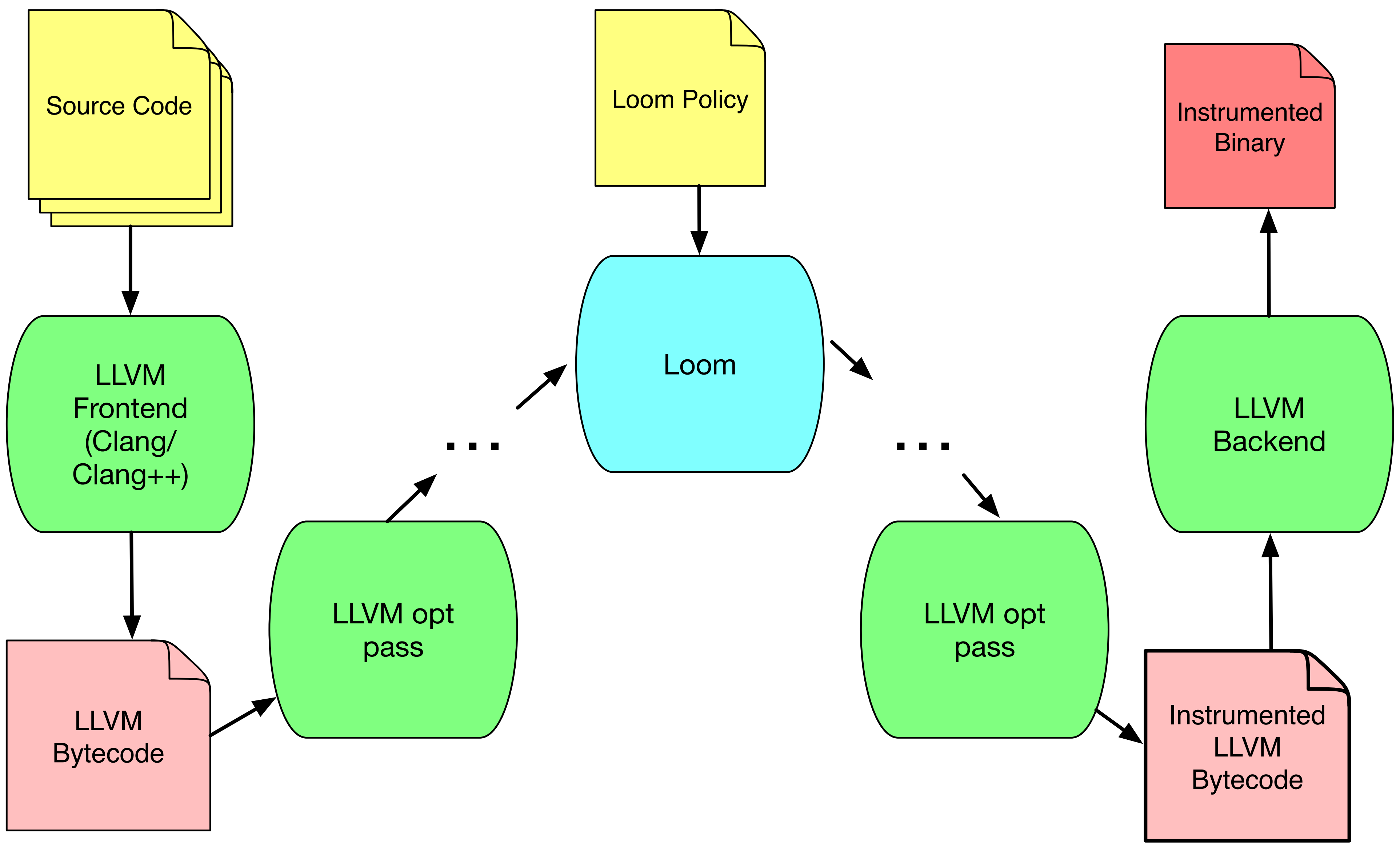


Figure 1: Loom Build Process

Unlike the original USDT method, changes to source code are not needed. In fact, the source code of the application is not needed. As long as a LLVM IR representation of the code is available, the instrumentation can be added to the program.

One CADTES use case is to capture all authentication attempts using the Pluggable Authentication Modules (PAM) system. These can come from many sources such as SSH or sudo, so to capture them all we targeted the PAM library itself. A sample Loom policy YAML file is given in Listing 1 showing how we where able to instrument PAM to log authentication attempts from userspace. The example is specifically instrumenting the pam_authenticate function, capturing all of the arguments to the call. Metadata is used to add context to the data when it is received by DTrace, allowing the author of the corresponding DTrace script to differentiate data from various userland probes. As a result, we are able to capture the username (and other details) of login attempts and the executables from which they come.

```
strategy: callout
dtrace: userspace
functions:
- callee: [entry]
  metadata:
    name: auth
    id: 1
  name: pam_authenticate
```

Listing 1: Loom configuration for logging PAM authentication attempts

In the majority of cases, this solution does not require the user to write custom code as in USDT. To get data into the DTrace system, an experimental system call was written using the SDT provider from DTrace to pass trace data to the kernel. The policy file provides all of the information needed to add context to the user data when tracing. The only time additional development is needed is when the user wants to transform the data in some way before it is sent to DTrace.

Using Loom allowed us to reproduce the DTrace USDT functionality without having to integrate the USDT toolchain into FreeBSD build process. The system does require a custom system call but it was a simple change that provided the flexibility to add and remove instrumentation to applications without any modifications to the source code. Since the operating system already uses LLVM we could integrate Loom into the build by adding additional build targets for programs and libraries to produce the LLVM IR output. Then Loom could be called as an additional build step where needed.

In the majority of cases, this solution does not require the user to write custom code as in USDT.

The FreeBSD base system contributed to the ease of this development. The base system of the operating system not only includes the kernel and userland source to be able to run a fully functional operating system, but it also includes a unified system for building FreeBSD. In order to use Loom in our research project we needed to be able to build program and libraries as bit code objects (BCOs). These BCOs are a binary form of LLVM IR that Loom can modify for instrumentation and transformation. Since all binaries in FreeBSD use a central set of build scripts we only had to modify these to create BCOs for any part of the base system. Once we had made these modifications we could apply Loom to any program or library necessary.

Expanding Loom

For some CADETS use cases it was necessary to transform the data collected before passing it to DTrace. For example, to avoid name collision between servers in an instrumented distributed system the project uses Globally Unique Identifier (GUID) for the usernames so they could be individually tracked in the outputs. To achieve this, a mechanism was added to allow external code to be inserted into a program through Loom. The user can add calls to custom functions or libraries as long as the symbols are available at the link stage of the build process. Though this functionality was originally designed to modify data before logging it, the concept opened up new possibilities such as code transformation in addition to instrumentation.

Since the conclusion of the CADETS project, we have been working on exploring these possibilities, such as the replacement of code within a program to use a new API. For example, to test a new network API would traditionally require changes to source code, replacing calls to the old API with calls to the new one. This process is tedious and prone to human error. We are expanding Loom to handle such tasks without the need for source code changes. By matching a set of function calls, we can have Loom remove the original code and replace it with one or more calls using of the new API.

Loom is a full featured instrumentation and transformation framework.

One current limitations to this work is the policy file which is used to specify the changes that Loom needs to make. Though we have the ability to make code transformations with Loom, the current YAML based format is not expressive enough to fully specify transformations. We are currently working on a language that will overcome this limitation. The aim of this language is to allow for very specific specification of transformations to be made by Loom.

The Future of Loom on FreeBSD

Loom is a full featured instrumentation and transformation framework. Loom has the ability to instrument functions and function calls as well as accesses to structure fields, global variable and pointers. One to one function call replacement is fully implemented and there is functionally to replace calls to a sequence of functions, though further configuration work is required to make this generally usable. Additionally there is the ability for many of these configuration to be matched using wildcards or limited within the scope of certain files from the source code.

Since its use in the CADETS project, Loom has seen interest from developers on other operating systems such as Linux. Though we have made efforts to support these users, Loom's main development continues on FreeBSD. With the upcoming addition of Link Time Optimization (LTO) in the FreeBSD build system, we will investigate the possibility of using Loom in the unmodified FreeBSD build system. We will also be use FreeBSD to test the new Loom configuration language, investigating areas where the transformation system can help maintain ports of software from other operating systems.

For more information on Loom and to follow future developments you can check out the project page at github.com/cadets/loom.

Acknowledgements

The author would like to thank those who helped and guided me during this work including George Neville-Neil and Domagoj Stolfa for their DTrace help. Thanks to Ed Maste for his answers to my FreeBSD questions whether it was directly or to connect me with someone who knew the answer. Many thanks to all the members of the CADETS Project including Robert Watson, Arun Thomas, Silviu Chiricescu, Jon Anderson and Amanda Strand.

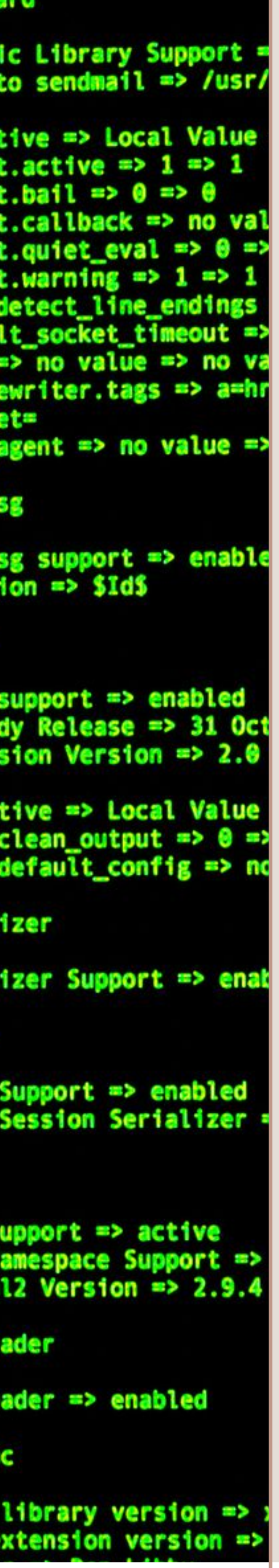
The FreeBSD community was great help to our efforts. Whether it was on IRC, mailing lists or at BSD conferences, it was generally easy to connect with the community and get answers to questions when we ran into issues. Thanks to the members of the community who were more than happy help out.

Finally the author would like to thank Jon Anderson for his feedback to improve the manuscript.

This work has been sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8650-15-C-7558. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

[1] Anderson, J; Neville-Neil, G. V.; Thomas, A.; Watson, R. N. M. "Cadets: Blending Tracing and Security on FreeBSD," *FreeBSD Journal*, May/June 2017, 12 - 17. 5

BRIAN KIDNEY is an Instructor of Cybersecurity at the College of the North Atlantic in St. John's, Newfoundland, Canada. He is also completing a PhD in Computer Engineering at Memorial University. His research interests include privacy and security, specifically as they relate to operating systems and programming languages. Brian has 20 years of experience as a Software Engineer developing software for multiple industries.



```
ic Library Support =
to sendmail => /usr/

tive => Local Value
t.active => 1 => 1
t.bail => 0 => 0
t.callback => no val
t.quiet_eval => 0 =>
t.warning => 1 => 1
detect_line_endings
lt_socket_timeout =>
=> no value => no va
ewriter.tags => a=hr
et=
agent => no value =>

sg

sg support => enable
ion => $Id$


support => enabled
dy Release => 31 Oct
sion Version => 2.0

tive => Local Value
clean_output => 0 =>
default_config => nc

izer

izer Support => enab

Support => enabled
Session Serializer =

upport => active
amespace Support =>
l2 Version => 2.9.4

ader

ader => enabled

c

library version =>
xtension version =>
```

Teaching an Undergraduate Unix Course

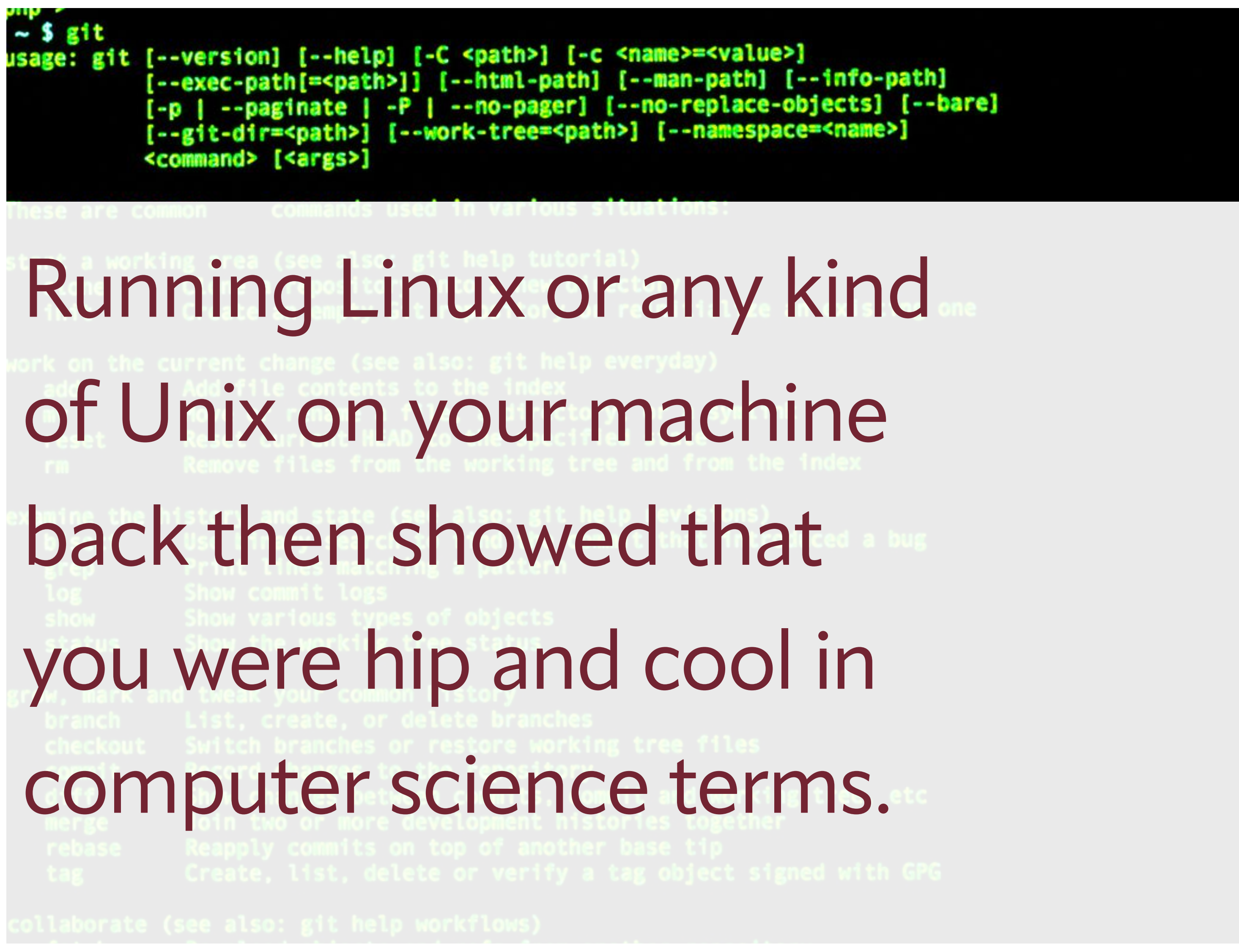
BY BENEDICT REUSCHLING

Backstory—Inheriting a Unix Course

Back in 2002, when I was an undergraduate computer science student, I made my first real contact with a Unix system. I vividly remember my first programming lecture when the professor opened both his ThinkPad laptop and the lecture itself with these words: "There are two types of professors in this department: those who use Unix — that is the group I belong to — and there is the group that uses Windows — and that's everyone else." He not only taught us programming, but he did it all in the command line, displaying programs with cat, editing them in vi, running make to compile, showing slides in X11. All very basic, yet somehow very appealing (maybe because I grew up with DOS, never knowing what I was missing in a terminal). After all, running Linux or any kind of Unix on your machine back then showed that you were hip and cool in computer science terms.

I attended all the lectures this professor taught — the advanced programming class, then operating systems and distributed systems. By that time, I also had my own laptop loading Linux, and shortly after discovering FreeBSD, I ran that exclusively.

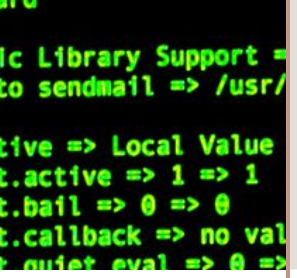
There was another course taught by the same professor — not a part of the main curriculum — called Unix for Developers. In this course, Unix was the primary learning objective. This was clearly old-school, but, nevertheless, interesting to me. We learned about Unix tools to edit files efficiently and wrote scripts in shell and awk to add missing functionality when needed. The course was challenging, but I was eager to learn and try out as much as possible on my own system.



```
~ $ git
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]
```

Running Linux or any kind of Unix on your machine back then showed that you were hip and cool in computer science terms.

As time went on, the professor eventually switched to a Mac (with many of the students doing the same), got involved in a lot more lectures and project work and no longer had time to teach Unix for Developers. This occurred after I had graduated and had worked at

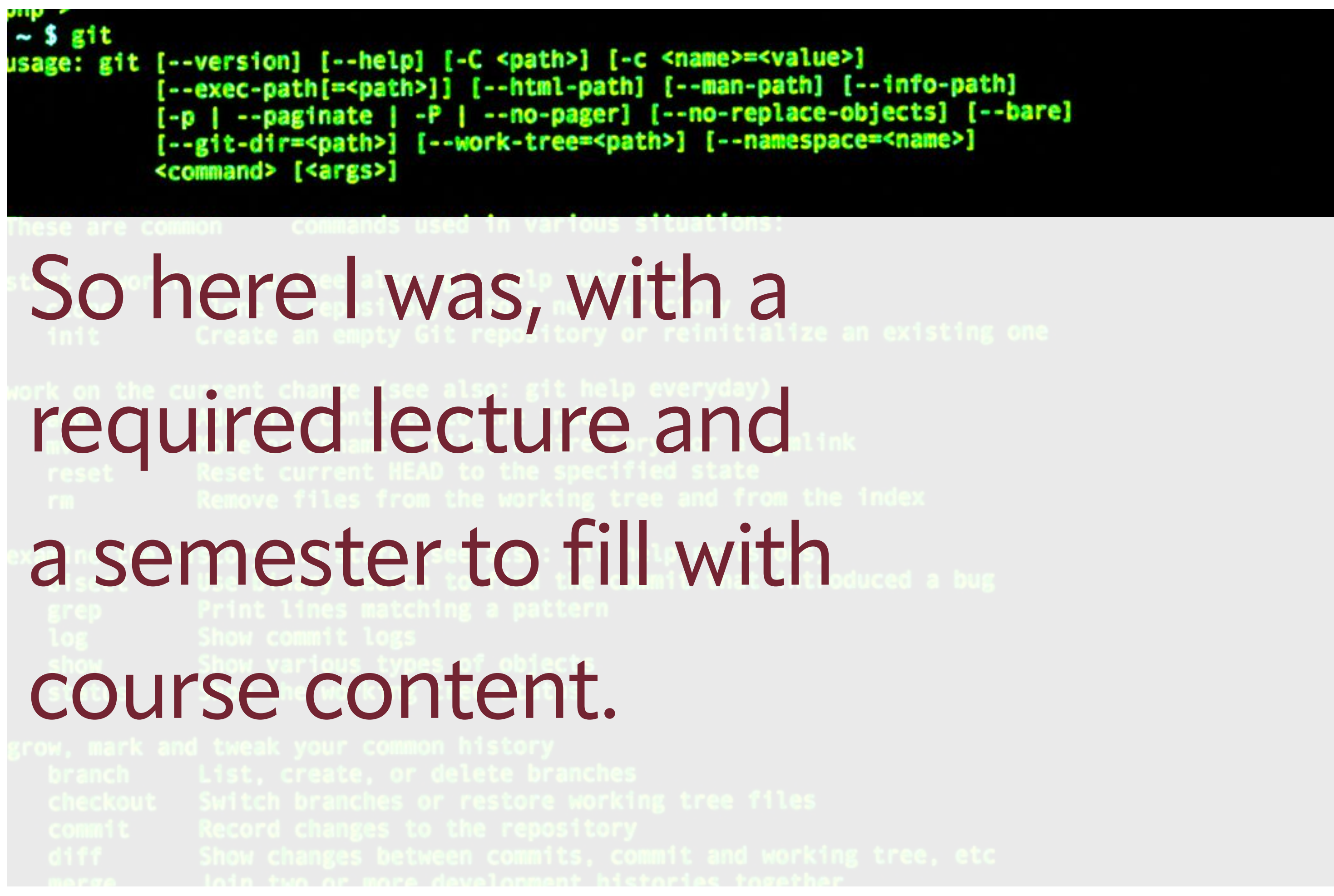


the department as a lab engineer. He must have remembered my enthusiasm for the subject when he asked whether I would be willing to take over his Unix for Developers course. I agreed and he outlined how it was handled. It had become a bit outdated at the time, covering Linux kernel 2.4 when 2.6 was already out, so I set out not only to update the content, but to also put in my own bits of BSD here and there.

A few years after teaching the course in German, I was asked whether I would be willing to teach it in English for our exchange students. As many of the concepts are in English and don't translate very well into German, I went over the whole script again and translated it. I've been running it in English ever since, even occasionally using extracts as tutorials at conferences. But let's go back to the early days...

The Organizational Structure

So here I was, with a required lecture and a semester to fill with course content. The course is taught in the winter term, which means roughly 15 weeks between October and January with a Christmas break in between. The written exam is scheduled for February, which leaves about two weeks between the last lecture and the exam date for the students to prepare.



```
~ $ git
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]
```

So here I was, with a required lecture and a semester to fill with course content.

There is a weekly, 90-minute lecture and a three-hour lab. Because of the popularity of the elective course, there are typically 40 students taking the course. This means that each lab group of roughly 16 students meets every 14 days, alternating with the other group so each have 5 lab dates in total.

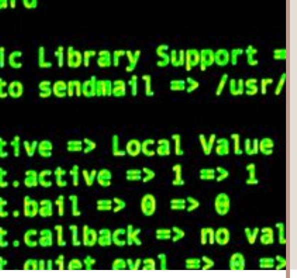
The labs are not graded but need to be completed by each student group (typically done in pairs) to be permitted to take the exam at the end. There is also no midterm evaluation — the final grade is determined by the exam result. This scenario has been debated many times and whether the labs should be graded given the effort students put into them. Other classes in this German University follow the same structure (with a few exceptions), so it's pretty much established form.

Since this is an elective course, I only get students that are interested in the topic. No student is required take my course to get their degree. They must have a certain number of elective courses listed in their records, but the classes can be chosen freely. Not only does this reduce the overall number of students, but I get two types of students: those who want to learn Unix and know very little about it, and those who know Unix quite well already and want to learn even more (or want to get an easy grade). Later, we will see that balancing these two groups is not always easy...

The Lecture

As previously mentioned, when I took over the lecture it was outdated as it had not been offered for a while. I thought I could put some BSD content into it to make it more Unix agnostic since I needed to update slides anyway.

During the very first introductory lecture each semester, it was typical for a student to ask: "which Linux distro are we going to use?" That question usually causes conflicts, as there



```
ic Library Support =
to sendmail => /usr/

tive => Local Value
t.active => 1 => 1
t.bail => 0 => 0
t.callback => no val
t.quiet_eval => 0 =>
```

are a lot of distributions out there. If I'd say Ubuntu, then the Linux Mint folks are disappointed, and if I say Mint, then the Arch Linux users are appalled. To avoid this (and to give everyone an equal chance to learn something new without sacrificing some familiar concepts), I usually answer the question with: "All of them and none. We're using FreeBSD, but if someone wants to use their distro of choice, then I won't stop you. After all, this is an introduction and not an indoctrination course."

I explain to the students that if they are new, beginning with FreeBSD is as good as any Linux distribution and the concepts carry over well. For those who already know a Linux distro, they can opt to dip their toes into a BSD system and discover a lot of the basics are the same. This usually appeases everyone, and I avoid the distro wars entirely. However, if students chose to run on their own favorite distribution, they don't get any help from me. It's their system, their choice, their administrative work, and not mine. The popularity of certain Linux distros changes with each student generation, while FreeBSD has happily remained mostly the same--stable, free of surprises, and easy to get started with. Interestingly, a lot of the stuff I teach is distribution agnostic, so there is little to no difference between systems. We can sometimes compare differences in class when I do demos (more on that later), but in essence, the commands we use all do what we want them to.

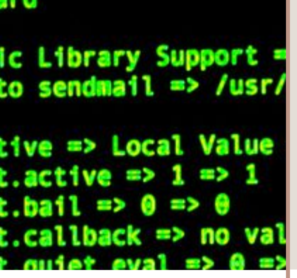
Here is what the course offers:

- Unix Overview (basics like logging in, commands like ls, cp, etc.)
- Editors (vi/vim crash course)
- Shell (command history, tab completion, redirections, pipes, here documents, background jobs)
- Shell scripting (big part on variables, control structures, loops, debugging)
- Shell scripting II (cdialog programming, functions, and traps)
- grep, sed, awk (extracting data from files, manipulating it in various ways, awk-programming)
- Filesystems (ZFS gets introduced here)
- Ansible (Setup, running ad-hoc commands, writing playbooks and change multiple jails in parallel)

This is a lot for 15 weeks, even though it may not look like much. One might argue that Filesystems don't belong in a programming-oriented lecture. That was a remnant from when I took the course, and I thought replacing ZFS for the regular filesystem concepts was a good compromise. Other parts like shell scripting were extended after I learned that some (but not all) of the professors also teach it as part of the mandatory course on operating systems. I added concepts like dialog programming (think of the FreeBSD installer to get an idea of how this may look), functions and traps. They fit together nicely since traps need to call functions when they're executing.

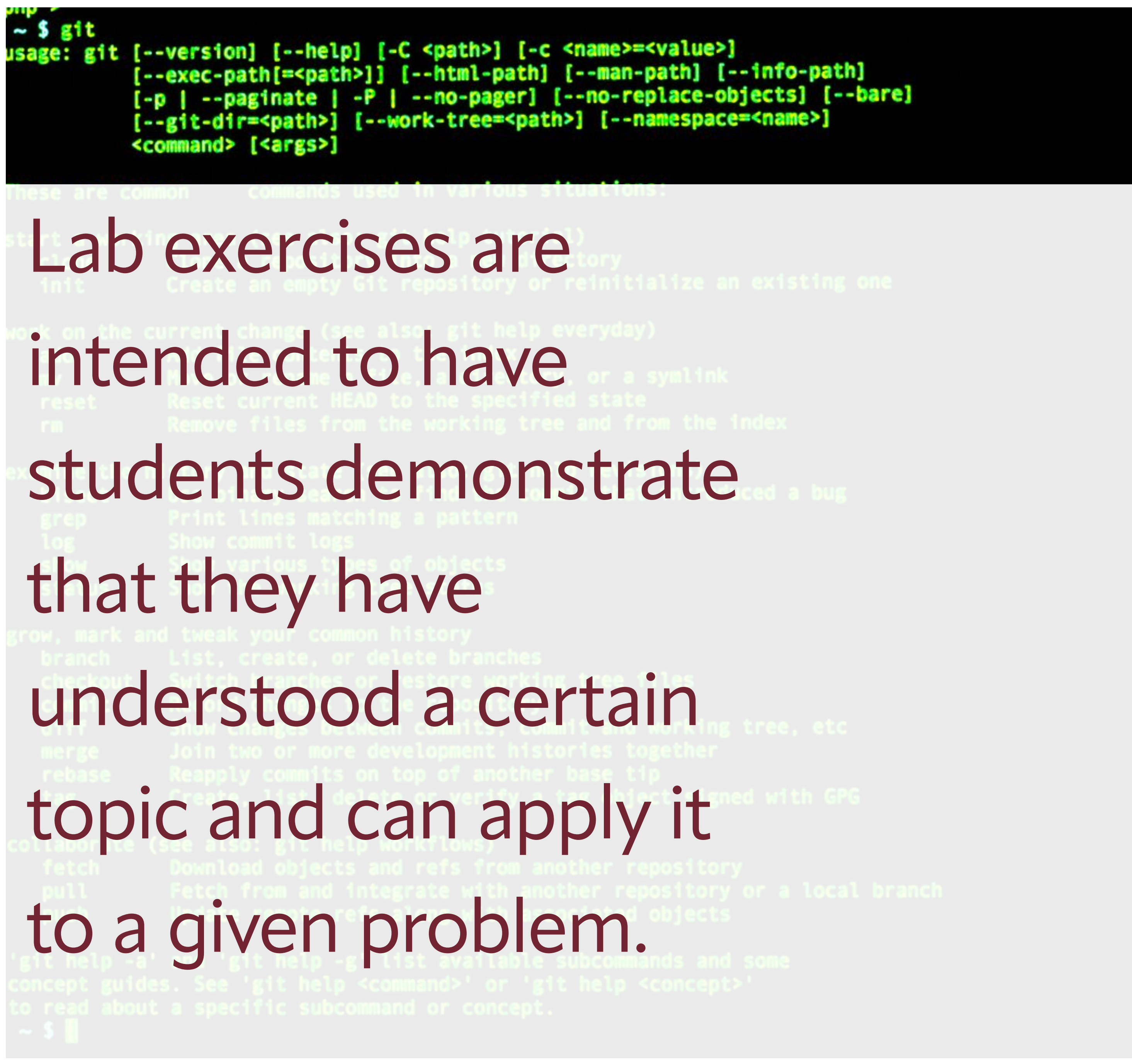
Some of the content changes more often over time, depending on my own interests and based on student feedback. Realistically, one could only comfortably present roughly 40 slides in a 90-minute lecture, considering questions and spending 2-3 minutes per slide.

When the pandemic hit, that format was no longer doable, so most colleagues and I switched to the inverted classroom model. In this teaching format, students study the material up-front, and the lectures are used to address questions and discuss the material. The inverted classroom allows the teacher to provide more material up-front and use the lectures to gauge whether there are common problems that should be explained in class for everyone. It also requires more initiative from the students. If there are not many questions, I assume everything is understood (which can backfire for the shy students), and I do a couple



of demos by sharing my terminal on a projector or in a video call. I've found that students like this format as they can try out things right away on their own machines, they get to see me make errors (nobody's perfect), and it gives the lecture a more dynamic nature rather than going through the material slide by slide.

Content gets added and updated based on student feedback. When I see that students struggle with something, I create a couple of extra slides to help them grasp that concept. This also depends on whether the students have had prior experience with the subject or are completely new to Unix. Overall, I've found that there is something new to learn even for seasoned Unix users, so it does not matter too much if there are some students who have had prior exposure. Typically, ZFS and/or Ansible is both new and exciting to the students because of the capabilities they provide. This is especially true for ZFS. I have had students tell me later — when I see them again in our master's program — that they are glad I taught it and that they use it at home for their own NAS.



```
~ $ git
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]
```

Lab exercises are intended to have students demonstrate that they have understood a certain topic and can apply it to a given problem.

The Labs

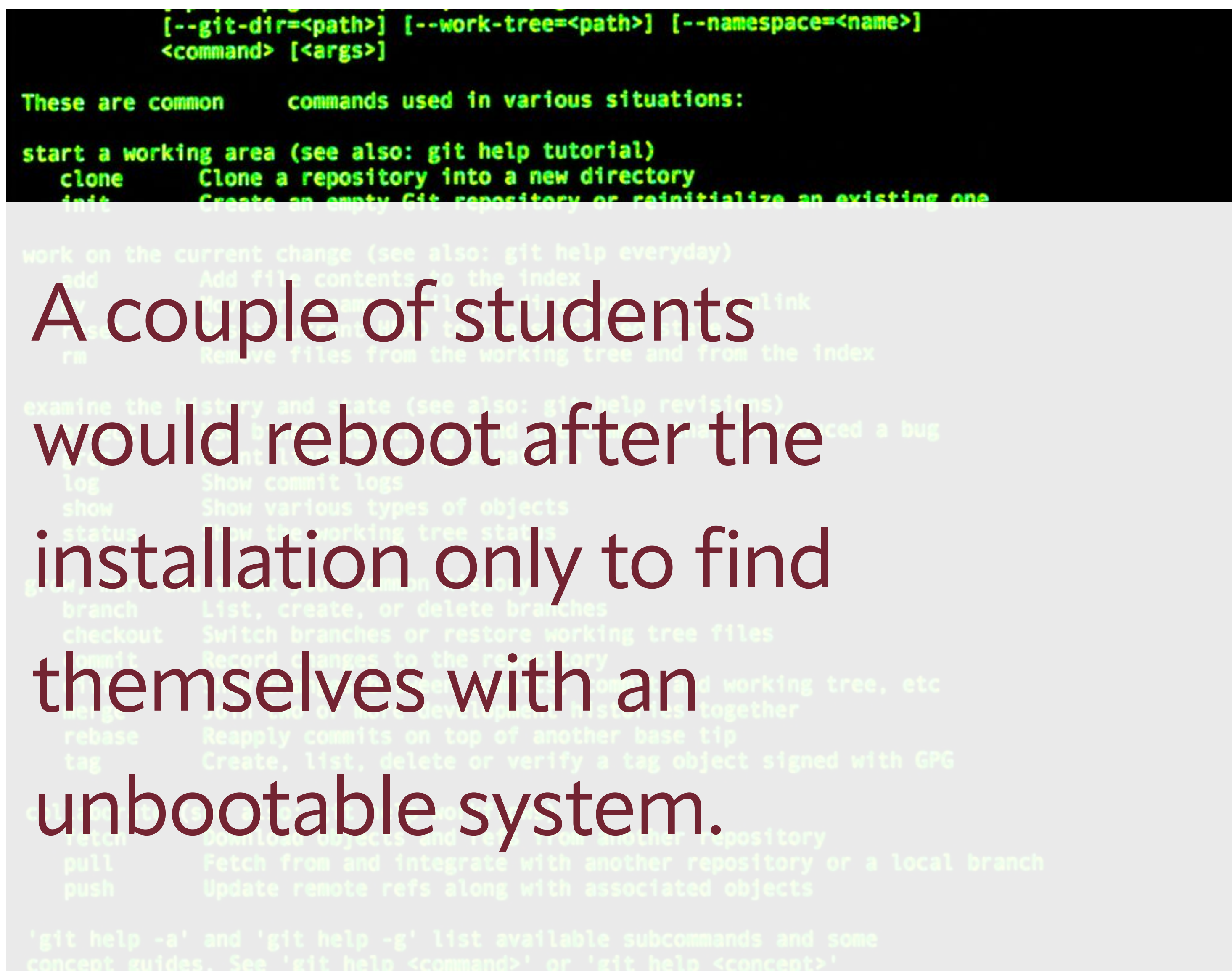
Lab exercises are intended to have students demonstrate that they have understood a certain topic and can apply it to a given problem. They typically work in pairs and present their results to me for evaluation. They need to get all 5 labs completed to take the exam. The exercises follow the material being taught in the lecture, but there can also be parts that are only explained on the lab assignment sheet and not in the lecture. This can be because it is too small to cover in class or is a separate topic that does not fit into the current curriculum.

Lab assignments typically involve getting to know something more about the system, doing a programming exercise (or before that, creating useful shell pipelines), text processing, making configuration changes in the system and similar tasks. The most difficult lab for me — the teacher — is always the first one — setting up the Unix system. Remember that we have two types of students. While some struggle with even the most basic installer, others bring a perfectly set up system to the lab and leave after 5 minutes of showing it to me. It's easy if you've done it with at least one distribution (learning about the FreeBSD specifics is typically easy enough), but if it's the first time, it can be difficult for a newcomer. The overall goal of this lab is to have a running system at the end for everyone to use and follow in class. Once that has been accomplished, the subsequent labs are much easier for participants. They can use their installed system and are basically all on the same level as far as the system is concerned.

I've also tried out different formats over the years to see which works best in getting everyone on the same page — so as not to overwhelm the newcomers and not to bore the experienced students. At the beginning, using the projector, I walked through the installer in a VirtualBox VM with the students, explaining concepts and terms as they came up. That worked somewhat, but the advanced students were moving ahead to the next screen and the explanations turned into a lecture of their own.

Then I switched to providing instructions as to what the installed system should have at the end — a certain partition layout, a local user separate from root, and a running network. This tended to create a lot of different results, even though they all used the same virtual hardware platform. Some didn't remember their passwords in the lab 14 days later or had made their partitions too small and couldn't install any software on it. In addition, cheating was much easier, as one student could pass around a finished VM image, and the other students simply imported that on their machines. The students did not learn much from simply running through the installer and hitting enter a couple of times with no clue as to what was going on behind the scenes (partitioning, DHCP calls to the network, etc.).

To prevent cheating and to give students a bit more information about what was going on, I provided instructions on how to do the installation manually. They would drop to the shell and do everything by hand: set up partitions, extract FreeBSD source archives, make basic settings for the network, and install the boot loader. All of this was accompanied by instructions on what the commands they were using did. To prevent cheating, I gave instructions to label their partitions in gpart after the uniquely generated disk ID from VirtualBox. That way, each system had its own ID and I could easily compare them.



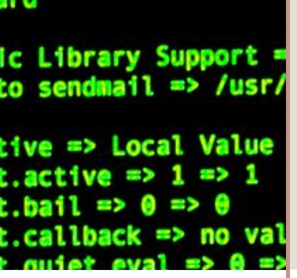
```
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]

These are common commands used in various situations:

start a working area (see also: git help tutorial)
   clone      Clone a repository into a new directory
```

A couple of students would reboot after the installation only to find themselves with an unbootable system.

That worked well to a certain extent. A couple of students would reboot after the installation only to find themselves with an unbootable system. They must have written the boot code to the wrong partition — like the one used for swap by not getting the ID in gpart right. I also had a few students stop the installation in between, suspend the VM to do something else, and later try to continue, only to find that the virtual CD provided by the FreeBSD ISO image would no longer be mounted, making all inputs result in "command not found" errors. Yet other students booted their systems just fine, worked with them and then in a later lab would reboot for the first time (suspending the VM all the time) and find themselves in an unbootable system — with all the solutions inside the unbootable partition. Not good, especially since those students would ask me what to do and have me figure out the particulars of their install from months ago.

Although I refined my manual installation instructions to include regular VM snapshots at certain points to go back to, other problems remained. Students did not read my explanations but would simply look for the next command to enter in a 12-page document (including images). That, of course, defeats the purpose of trying to teach them a little about how a Unix system like FreeBSD is installed and what components are involved. Again, the newcomers struggled with this more than the seasoned Unix users. Luckily, the number of struggling students was limited to only a few, while the rest did fine with this lab.

I'm currently doing a separate project with a small student group with the goal of providing course participants with a ready-made machine (jail) running some application. They have to keep this machine with the application running while I inject certain errors that the students have to find and fix. A global hiscore list displays how quickly each team solved it based on points given by a check program that runs over these systems to figure out if and when an injected error is no longer present. Of course, I could inject different errors for



```
ic Library Support =
to sendmail => /usr/

tive => Local Value
t.active => 1 => 1
t.bail => 0 => 0
t.callback => no val
```

each group or even multiple ones. From shutting down services to removing execute bits or whole files — the possibilities are endless (at least in my mind). Students learn how to keep things running, they don't have to deal with installing it properly in the first place (which is what they typically find at a company), and learn skills to find and fix common errors. We're still fleshing out the details, but I think it will be engaging for students.

The Exam

What can I write about the exam without giving away too much of the content? Since I switched to an English-only lecture a couple of years ago, students fear that they will not understand the questions. But that turned out not to be a problem. The questions are typically programming related like "find the error in this short shell script," which bridges the human language gap quite nicely. There is multiple choice, fill in the blanks, write a short script on your own, or tell me what CoW in ZFS terms means. All are familiar question types for students at this point in their studies.

From the results, I can see that newcomers have an equal chance of getting a good grade in this course as those with prior exposure to a Unix system. I can't tell if the latter group studies at all for the exam, but I can certainly say that not studying at all does not guarantee a good grade. Since the exam typically contains material from the labs in different form, I can also tell afterwards which of the two in the lab group really did the exercises and who did not. That is a late revelation for the students and for me, but sometimes my intuition about which student is the better one is wrong.

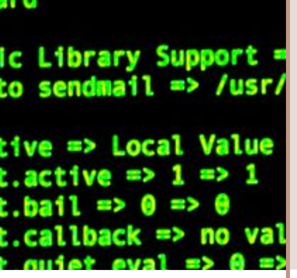
Aftermath

Once grading is done and the students have had a chance to review their exams (which they rarely do), the class officially ends. But that does not mean the work is done for the teacher. Since this is a yearly course, I have time over the summer to relax and reflect on it. From the feedback and experiences in the lecture and labs, I refine or even completely rewrite certain parts — typically the ones that evoked a lot of questions during the labs or were small points raised by the group in the exam.

I also find cool new things in the Unix space that I want to teach in the future. During my sysadmin work, I occasionally come across a piece of code or a little problem that later becomes an exam question. Collecting these over the summer break refreshes the course content not only for me, but for the next generation of students. So, it is rare that two consecutive courses will be taught completely the same. That would be boring to me and the students and lab, and exam solutions from previous years would propagate over time.

Can I teach everything that Unix has to offer or that I think students should know? Certainly not. I can scratch the surface and hope that students find it sufficiently interesting to continue learning about it on their own after the course. Some of the more advanced topics are covered by colleagues who go deeper into subjects like managing cloud application development, systems programming in Rust, and similar topics offered as elective courses. Some students complain that I don't cover docker, but then I remind them that we're looking at jails which also have cool features.

Of course, you also have to address recent developments and trends. Whereas a couple of years ago, we'd still have to do basic HTML introductions in another course, we can now assume that many students already possess that knowledge from their school days or private dabbling. The same is true for hardware. A lot of students have never built their own



```
ic Library Support =
to sendmail => /usr/

tive => Local Value
t.active => 1 => 1
t.bail => 0 => 0
t.callback => no val
```

computer and have only used complete systems. Talking about interactions between components like CPU, RAM, and storage may seem new to those students, even though that is covered in the mandatory operating systems class. If students only bring a tablet or are only used to a graphical UI, it's difficult to introduce them to a text-based shell with a blinking cursor. This is not a FreeBSD-only problem, as each Unix eventually revolves around using the shell, even though it runs in a bells-and-whistles GUI.

I think students are happy to get an introduction to Unix that goes beyond what they learn in their operating systems classes. While those classes usually revolve around how a scheduler works, what the MMU does, and how system calls are good to know for programmers, my course is a more hands on, day-to-day use of Unix as an operating system for end users. It's certainly not perfect and has to constantly adapt to the changing times, but I like the current concept and students do as well.

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

Getting Started with FreeBSD

BY ROLLER ANGEL

Working with the Front Range BSD User Group, I have taught a Getting Started with FreeBSD workshop at SCaLE for a few years now. I do so because I realize the power of the tools and enjoy sharing my experience with others through workshops like this one. I also come away from each session with fresh validation for my own setup and with feedback that fuels a steady flow of improvements to the workshop. The most recent session was at SCaLE 19x, and in previous years, the workshop was presented at SCaLE 17x and 18x.

The Workshop at SCaLE 19x

Participants came wandering into the ballroom where they were welcomed by the projector screen's Getting Started with FreeBSD title slide — created by the FreeBSD Foundation's Marketing Coordinator, Drew Gurkowski. The presentation was livestreamed via YouTube at https://www.youtube.com/watch?v=ByFCRwMJATM and screenshots can be pulled from the livestream.

Typically, the FreeBSD Foundation's Executive Director, Deb Goodkin, begins the workshop with an introduction to the presentation, but this time, Drew Gurkowski did a great job with that.

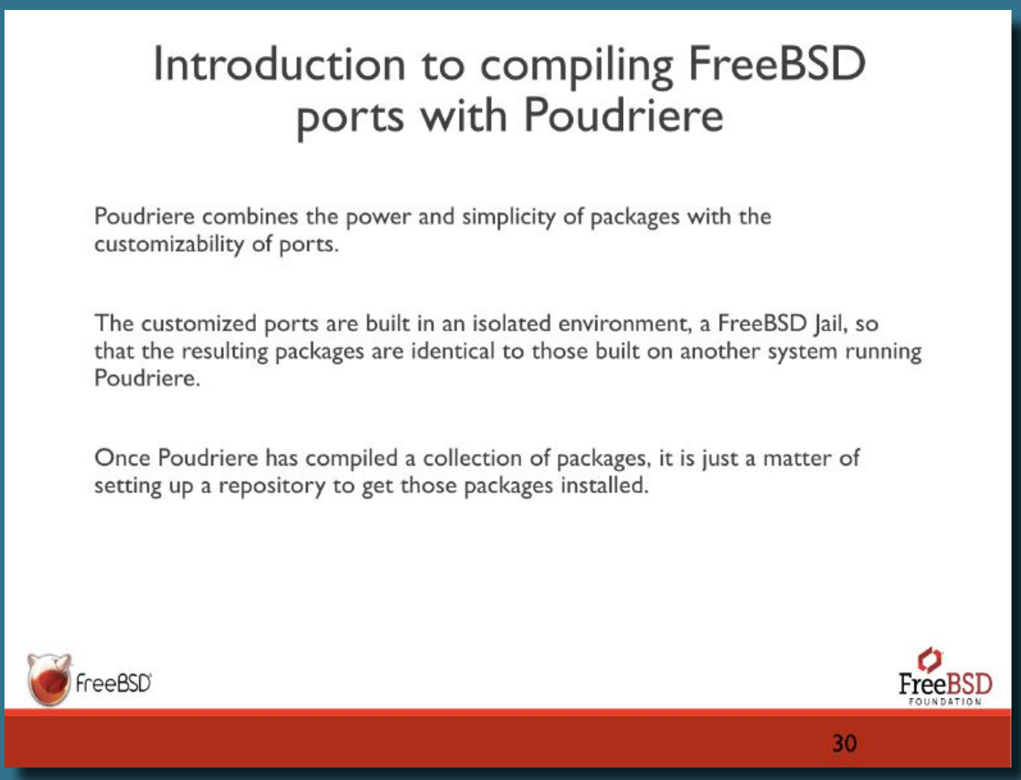
Getting Started With FreeBSD

Roller Angel - FreeBSD Sys Admin
Drew - FreeBSD Foundation

With each workshop, we want to make sure that participants can work through hurdles and follow along with the process. We initialize and configure a virtual machine in VirtualBox, which we use for lack of a computer lab to make use of a lab machine. Part of the process is inserting the virtual cd into the CD-ROM drive of the virtual machine. We then boot from that cd drive and install FreeBSD to the virtual hard drive of the virtual machine. At the end of the FreeBSD installer, we run the command **`shutdown -p now`** to instruct FreeBSD to shut off the computer. That way, we can remove the virtual disc and prevent the virtual machine from starting from the disc when booting again. Once we've installed FreeBSD to the hard drive, we'll want to boot from the hard drive from then on.

At this point in the workshop, we take a short break, and I use the time to find anyone who hasn't quite caught up and see how I can assist them. The most typical issues I see are incorrect virtual machine settings. We stick with defaults on most settings, but, as an example, someone had unchecked Enable I/O APIC in the System settings of the virtual machine, so checking that box fixed it. Another participant set the machine type to 32-bit and changing that setting to 64-bit solved their issue. When troubleshooting, keep in mind that even a small typo in a package name or setting can be the culprit. More recently, an issue people have faced is the lack of a suitable hypervisor on the new Apple processors, as VirtualBox is not supported on them. We had to work around a few small quirks with the conference WiFi regarding the DNS settings being provided to our virtual machines via DHCP and we ended up changing the nameserver listed in our `/etc/resolv.conf` file.

The next step is demonstrating how thin the line is between server and desktop. Installing a few packages and updating some configuration files is what it takes to get the desktop ready to start. All we need to do is issue the **`startx`** command to tell FreeBSD to start up the desktop. It's great to see participants realize they have just set up their own desktop and that there was no specific distro or flavor of FreeBSD needed for a specific X Window System window manager such as KDE Plasma 5, Lumina, or GNOME. We used XFCE, but also demonstrated how easy it is to install and configure whichever one you want to use. With the window managers running, you can interact with GUI applications like the web browser, programming IDE's, file manager, etc.



Introduction to compiling FreeBSD ports with Poudriere

Poudriere combines the power and simplicity of packages with the customizability of ports.

The customized ports are built in an isolated environment, a FreeBSD Jail, so that the resulting packages are identical to those built on another system running Poudriere.

Once Poudriere has compiled a collection of packages, it is just a matter of setting up a repository to get those packages installed.

30

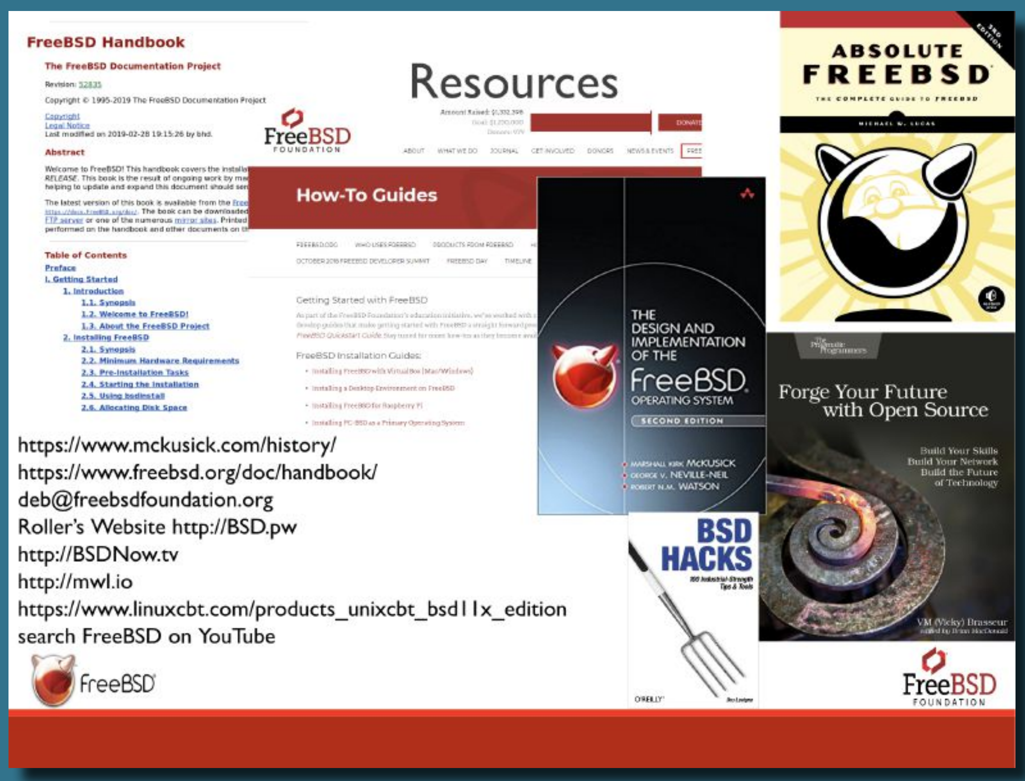
I thought it was important to also introduce participants to the process of building a custom package repo. If they run into an issue that requires customization of a port and building their own package, they already know how to avoid common pitfalls and don't go down the path of mixing ports and packages. The tool we used is called Poudriere, and it makes building your own package repo quite easy and straightforward.

As participants learn to type their commands into the command line, a fitting tool to discuss is Ansible which is typically used for configuration management and works well for controlling remote machines over SSH. We demonstrate how to clone our FreeBSD virtual computer and connect to it via SSH. This way, we can try out Ansible and see how easy it is to tell Ansible to type the commands using a tool called Ansible Playbooks. Included as part of the workshop is an Ansible Playbook we use to setup Poudriere on a remote machine that builds all our packages and synchronizes the resulting files back to our local machine. The idea is that we can rent a very powerful machine for a short period of time to build our packages and then destroy that machine once we have the package files downloaded to our local machine and no longer need the package builder machine. To use the downloaded packages, we can change the package repository settings to point to a **`file://`** path where our package files can be found rather than the default https://download.FreeBSD.org setting.

We also discuss FreeBSD Jails so that participants can get a feel for them and see how easy they are to manage using iocage. We recommend MWL.io for in-depth books regarding FreeBSD Mastery and for workshop participants to follow along with Michael W Lucas as he deep dives into topics regarding FreeBSD Jails, Poudriere, Installing FreeBSD and much more.

The Participants

It was a fantastic group and people from all backgrounds and experience levels were able to geek out on some cool tech and learn something new that will help support their future work. It's easy to see where you can use FreeBSD to solve problems once you get the hang of installing and configuring it. With configuration management tools like Ansible, you can take what you've learned even further as you have a growing record of the changes you made to configuration files and the packages you installed. You can quickly pick up where you left off and continue to learn even more as you progress on your journey with FreeBSD.


Resources

https://www.mckusick.com/history/
https://www.freebsd.org/doc/handbook/
deb@freebsdfoundation.org
Roller's Website http://BSD.pw
http://BSDNow.tv
http://mwl.io
https://www.linuxcbt.com/products_unixcbt_bsd11x_edition
search FreeBSD on YouTube

We had a very enthusiastic crowd and several participants even brought along laptops on which they planned to install FreeBSD, and some had questions about FreeBSD WiFi. You can easily use USB Tethering from Android to share an internet connection with FreeBSD, plug in the cord, enable the tethering, and then execute the command `dhclient ue0` as a privileged user and that will use DHCP to get an address from the first USB Ethernet device. Of course, you can always configure your internal WiFi card with `/etc/rc.conf` and `/etc/wpa_supplicant.conf` as well. Check section 5 of the FreeBSD Manual Pages for details on these files including the list of supported options. More info on WiFi is in Chapter 32: Advanced Networking of the FreeBSD Handbook.It's good to know which options are available and the workshop aims to get all the usual FAQs answered, to get people using FreeBSD for something cool, and to help people use the software to solve problems.

In closing, I want to mention one participant who came in halfway through the day, was way behind, and not able to catch up. After the workshop, I sat with him in the lobby and helped him get everything running. He had an old core 2 Duo processor, so it was taking extra-long to complete the process on his machine. He was grateful for the help and expressed interest in learning more about BSD. I suggested *FreeBSD Journal*, MWL.io, and BSD User Groups. Plus, I'm always happy to help and my website is http://BSD.pw

ROLLER ANGEL spends most of his time helping people learn how to accomplish their goals using technology. He's an avid FreeBSD Systems Administrator and Pythonista who enjoys learning amazing things that can be done with Open Source technology — especially FreeBSD and Python — to solve issues. He's a firm believer that people can learn anything they wish to set their minds to. Roller is always seeking creative solutions to problems and enjoys a good challenge. He's driven and motivated to learn, explore new ideas, and to keep his skills sharp. He enjoys participating in the research community and sharing his ideas.

Pragmatic IPv6 (Part 2)

BY HIROKI SATO

Part 1 explained the basics of IPv6 protocol and how to get started using it on a FreeBSD box. You should be able to use automatically-configured link-local scope IPv6 addresses after reading it. These addresses are still powerful and helpful while they are limited to your LAN—you need no global IP address if you just want to communicate to another box on the same network. A link-local address is not routable. It is not likely an attack surface from malicious users on the Internet.

To get access to the IPv6 Internet, you need to configure at least one IPv6 global-scope unicast address. This column focuses on IPv6 deployment scenarios with routers to understand more practical configurations.

Internet in IPv4 and IPv6

After trying an **`sshd(8)`** example in Part 1, you probably want to try access to the IPv6 Internet. Let's look into what you need to do so and the basics of IPv6 network design.

As you know, the Internet is a global-scale interconnected network driven by the Internet protocol. To go somewhere outside your local network, you need a router that is reachable from/to the Internet. It knows "routes to the outside", and forwards IP packets from your network to other networks.

You should already have a router for the IPv4 Internet. It might be one provided by your ISP[1], or one at a data center that connects your box located there to the Internet. The ISP usually offers an connection endpoint for upstream networks that are reachable from the Internet.

Note that IPv6 is not a compatible protocol with IPv4 while it was designed as a successor. This means that IPv6 and IPv4 Internet are entirely independent, and you need an IPv6 router and an IPv6 endpoint from ISP. "IPv6 is supposed to be upper-compatible with IPv4" is one of the common misunderstandings of IPv6. This comes from the fact that most IPv6 deployments have been implemented by migrating existing IPv4 networks. For instance, you can make your public IPv4 HTTP server on FreeBSD "IPv6-ready" because FreeBSD supports both IPv4 and IPv6. However, you might not want to make it "IPv6-only" simply because people with no IPv6 connectivity cannot access your server. Thus IPv6 deployments are typically done by making existing IPv4 services and networks IPv6-capable in addition to

IPv4. This migration approach is called "dual-stack," and one of the causes that makes you believe IPv4 and IPv6 are always usable on the same machine and are somehow related.

Design of an IPv6 Network

IPv6 is independent of IPv4, so you must design an IPv6 network. Fortunately, IPv6 is almost the same as IPv4 regarding network elements, such as routers and network boundaries. Let's review how an IPv4 network works and then see specifics about IPv6.

IPv4 Local-Area Network Configuration

IPv4 has "network address" or "subnet", which is represented by a host address and a network mask (or a subnet prefix length). For instance, an IPv4 address `192.0.2.1/24` means that you have a network with an address `192.0.2.0` and the 254 host addresses from `192.0.2.1` to `192.0.2.254` are available[2] on the same network for you. The nodes with the same network address share the same L2 segment[3]. In other words, two nodes on the same segment can talk with each other with no router. If a node wants to communicate with another node with a network address outside `192.0.2.0`, it must be sent to a router which knows that destination. Thus, to configure a host, the following information is required:

- a global IPv4 host address,
- a network mask or a subnet prefix length to compute the network address,
- router information to communicate with nodes outside the network.

The router information on IPv4 end hosts is usually configured by specifying a single router on the same network as "the default router". These three elements can also be automatically configured by using DHCP[4]. DHCP is pretty famous for IPv4 end hosts though it is an optional protocol. FreeBSD has `dhclient(8)` as a client-side implementation.

On an IPv6 network, these three are automatically configured to some extent. Of course, you can manually configure them, but it is not recommended because the IPv6 address space is quite big. Let's see how the IPv6 network works, in detail.

Single Router Network, Manual Configuration

Figure 1a shows a simple IPv6 network that contains a router and two (or more) IPv6 hosts on the same segment. To configure this network manually, the following `rc.conf` variables are used on one of the IPv6 hosts:

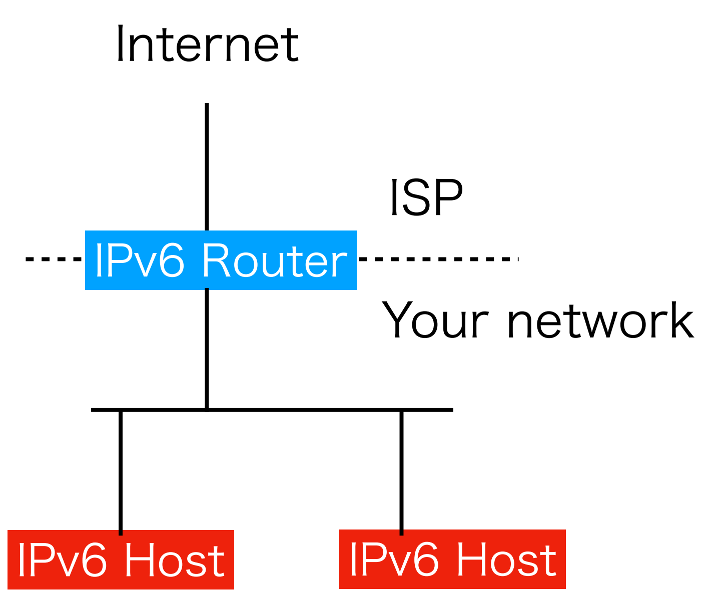


Fig 1a: A simple IPv6 network with a router.

```
ifconfig_bge0="inet 192.168.0.10/24"
ifconfig_bge0_ipv6="inet6 2001:db8:0:1::1/64"
ipv6_defaultrouter="fe80::5a52:8aff:fe10:e323%bge0"
```

This assumes the host has a network interface, `bge0`, the ISP provides the IPv6 router, and your global IPv6 prefix is `2001:db8:0:1::/64`. You can choose the host address. In this example, `2001:db8:0:1::1/64` is chosen.

The `ifconfig_bge0` line configures an IPv4 address. As explained, IPv6 is independent of IPv4. You do not have to add an IPv4 address when using IPv6 only. For an IPv6-only configuration, this can be rewritten like the following:

```
ifconfig_bge0="up"
ifconfig_bge0_ipv6="inet6 2001:db8:0:1::1/64"
ipv6_defaultrouter="fe80::5a52:8aff:fe10:e323%bge0"
```

Note that you cannot omit the `ifconfig_bge0` line nor write IPv6 configuration in the line. This line is required to teach the `rc.d(8)` framework that the `bge0` interface exists. If this is missing, `bge0` will not be configured. We need no IPv4 address here, but it must not be empty. Thus "up" is used as a harmless sub-command for the `ifconfig(8)` utility. "up" just activates the interface. This is for historical reasons and may be changed in the future releases of FreeBSD[5], but please remember that FreeBSD releases up to 13.x, the latest release at the time of writing, require this rule. IPv6 configurations should be in the `ifconfig_bge0_ipv6` line instead.

The `ifconfig_bge0_ipv6` line is used to configure an IPv6 address and options. The rc.d(8) framework uses this line to recognize whether the interface is IPv6-ready or not. If you omit this line, IPv6 communication on the interface will be blocked. In Part 1, we had `ifconfig_bge0_ipv6="inet6 auto_linklocal"`. Even if IPv6 addresses are automatically configured, you need this line.

The `ipv6_defaultrouter` variable specifies the default router as `defaultrouter` for IPv4 does. You need an IPv6 address of the router. Usually, this information is not provided explicitly. You can find the router's address by using `ping6(8)` utility:

```
% ping6 ff02::2%bge0
PING6(56=40+8+8 bytes) fe80::5a9c:fcff:fe10:ffc2%bge0 --> ff02::2%bge0
16 bytes from fe80::5a52:8aff:fe10:e323%bge0, icmp_seq=0 hlim=255 time=0.996 ms
16 bytes from fe80::5a52:8aff:fe10:e323%bge0, icmp_seq=1 hlim=255 time=1.099 ms
^C
--- ff02::2%bge0 ping6 statistics ---
2 packets transmitted , 2 packets received , 0.0% packet loss
round -trip min/avg/max/std-dev = 0.996/1.048/1.099/0.052 ms
```

`ff02::2` is the all-routers multicast address. ICMPv6 echo request packets sent by the `ping6(8)` utility to this address will be received by routers on the network, and you will receive ICMPv6 echo reply packets. You can find the addresses by observing the replies. It is `fe80::5a52:8aff:fe10:e323%bge0`.

After adding this configuration into `/etc/rc.conf`, you can run the `service(8)` utility to reconfigure `bge0`:

```
# service netif restart bge0
```

After the reconfiguration is done, you should notice that `bge0` has two addresses, `2001:db8:0:1::1/64 and fe80::xxx/64`. The latter is an automatically-configured link-local address explained in Part 1. Remember that at least one link-local address must be configured on an IPv6-capable interface, even if you are using a global-scope IPv6 address provided by the ISP. The "xxx" part varies depending on the MAC address on the interface.

The source address is chosen from one of these two addresses based on the destination address. Now you can `ping6(8)` to `www.freebsd.org` like this:

```
% ping6 www.freebsd.org
PING6(56=40+8+8 bytes) 2001:db8:0:1::1 --> 2610:1c1:1:606c::50:25
16 bytes from 2610:1c1:1:606c::50:25, icmp_seq=0 hlim=46 time=155.715 ms
16 bytes from 2610:1c1:1:606c::50:25, icmp_seq=1 hlim=46 time=151.051 ms
16 bytes from 2610:1c1:1:606c::50:25, icmp_seq=2 hlim=46 time=152.218 ms
^C
--- wfe2.nyi.freebsd.org ping6 statistics ---
3 packets transmitted , 3 packets received, 0.0% packet loss
round -trip min/avg/max/std-dev = 151.051/152.995/155.715/1.982 ms
```

and you should see a global address, `2001:db8:0:1::1/64`. If the destination is `fe80::5a52:8aff:fe10:e323%bge0`., the link-local address will be used instead.

Note that DNS domain name resolution is performed by the name servers listed in `/etc/resolv.conf`. If you have a working configuration for IPv4, the file has a list of IPv4 addresses. If your IPv6 router works as a DNS proxy, you can put the IPv6 address like this:

```
nameserver fe80::5a52:8aff:fe10:e323%bge0
```

You can put the address similarly if your ISP provides a DNS recursive resolver.

That's all of the fully-manual configurations of an IPv6 host. You should be able to enjoy IPv6 Internet access by using your favorite software that supports IPv6.

Single Router Network, Configured by SLAAC

Figure 1b shows the same network structure, but in this case, the IPv6 hosts are configured automatically by messages from the router. NDP[6] is a part of the IPv6 core protocols which is implemented on top of ICMPv6, and defines this automatic configuration capability. The router can distribute network parameter information in RA (Router Advertisement) messages.

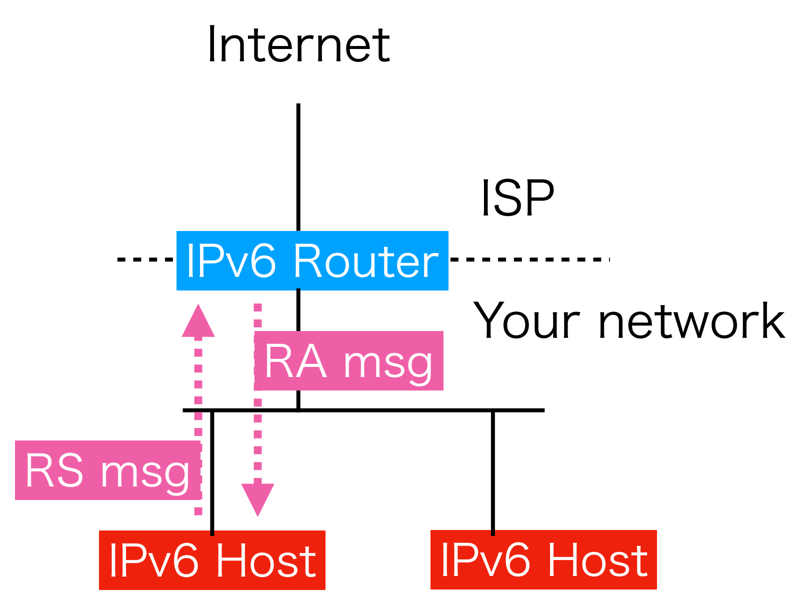


Fig 1b: Configuration by SLAAC

RA messages will be send to the address `ff02::1`, so all of the IPv6 nodes on the same network will receive them. "RS messages" in Figure 1b are Router Solicitation messages. They are used to solicit RA messages and sent to `ff02::2`, allrouters multicast address. A router will send an RA message when receiving an RS message from a host, and also send unsolicited RA messages periodically so that the IPv6 hosts can know network information changes.

RA messages have a lot of options, which are similar to ones for IPv4 DHCP. Fundamental ones are MTU, prefix, and the default router address so that an IPv6 host can configure itself by using them.

If your router supports RA messages and you want to rely on them, the following configuration works on the IPv6 host:

```
ifconfig_bge0="up"
ifconfig_bge0_ipv6="inet6 accept_rtadv"
```

`inet6 accept_rtadv` enables **`bge0`** to accept RA messages and configure the interface. An IPv6 router usually provides prefix information of your network. When a host receives the prefix information, IID[7] will be automatically generated from the MAC address, and an IPv6 address is configured. This mechanism is called SLAAC[8]. You can see this auto-configured address in the output of **`ifconfig(8)`**. **`"autoconf"`** keyword is shown just after the address.

```
% ifconfig bge0
...
inet6 2001:db8:a743:3c00:5a9c:fcff:fe10:ffc2 prefixlen 64 autoconf
...
```

To check if your IPv6 router supports RA messages, you can use the **`rtsol(8)`** utility with a -D flag. It sends a RS message and shows RA messages from the routers:

```
# rtsol -D bge0
rtsol: link -layer address option has null length on bge0. Treat as not included.
rtsol: checking if bge0 is ready...
...
rtsol: received RA from fe80::5a52:8aff:fe10:e323 on bge0 , state is 2
rtsol: Processing RA
rtsol: ndo = 0x7fffffffe3b0
rtsol: ndo->nd_opt_type = 1
rtsol: ndo->nd_opt_len = 1
rtsol: ndo = 0x7fffffffe3b8
rtsol: ndo->nd_opt_type = 3
rtsol: ndo->nd_opt_len = 4
rtsol: rsid = [bge0:slaac]
rtsol: stop timer for bge0
rtsol: there is no timer
```

An RS message will also be sent just after the interface becomes "up". Note that the kernel, not this utility, will process the RA messages. So if the interface is not configured "**`inet6 accept_rtadv`**", messages are shown but nothing is actually configured. If you did not get "**`received RA from...`**" line, your IPv6 router did not respond to the RS message. In this case, you cannot use the automatic configuration.

After receiving RA messages, the SLAAC address is configured automatically. And the default router will be configured by using the message's address. Thus just putting "**`inet6 accept_rtadv`**" into **`/etc/rc.conf`** configures a global IPv6 address and the default router.

This is something like DHCP in IPv4. However, there is no server nor no "state" for RA/RS messages. The end host will configure the network parameters upon receiving the messages. While this is a more scalable method than IPv4 DHCP, you cannot control what address is actually configured on each host because they are generated from the MAC addresses. For more fine-grained control of the automatic address configuration, you will need another method, such as DHCPv6.

DNS recursive resolvers can also be configured via RA messages[9]. The kernel cannot handle this information, so the **`rtsold(8)`** daemon will handle it. The **`rtsold(8)`** daemon can be enabled by the following **`rc.conf(5)`** variables:

```
ifconfig_bge0="up"
ifconfig_bge0_ipv6="inet6 accept_rtadv"
rtsold_enable="YES"
rtsold_flags="bge0"
```

`/etc/resolv.conf` will be updated when receiving RA messages if your router provides the information.

RA messages should be enabled on a properly-configured IPv6 network with one or more routers. The absence of RA messages makes network configuration difficult because they have both network parameters and how to configure them.

IPv6 Router Configuration

The previous section assumes that the IPv6 router is provided by your ISP. You can build an IPv6 FreeBSD router by yourself, as you can do it for IPv4. Let's see what must be configured for that.

Figure 2a shows an example that your network has an IPv6 router to have two independent networks. LAN 1 and LAN 2 are connected to each other by the router, and another router provides IPv6 Internet reachability. This section explains how to configure the former one.

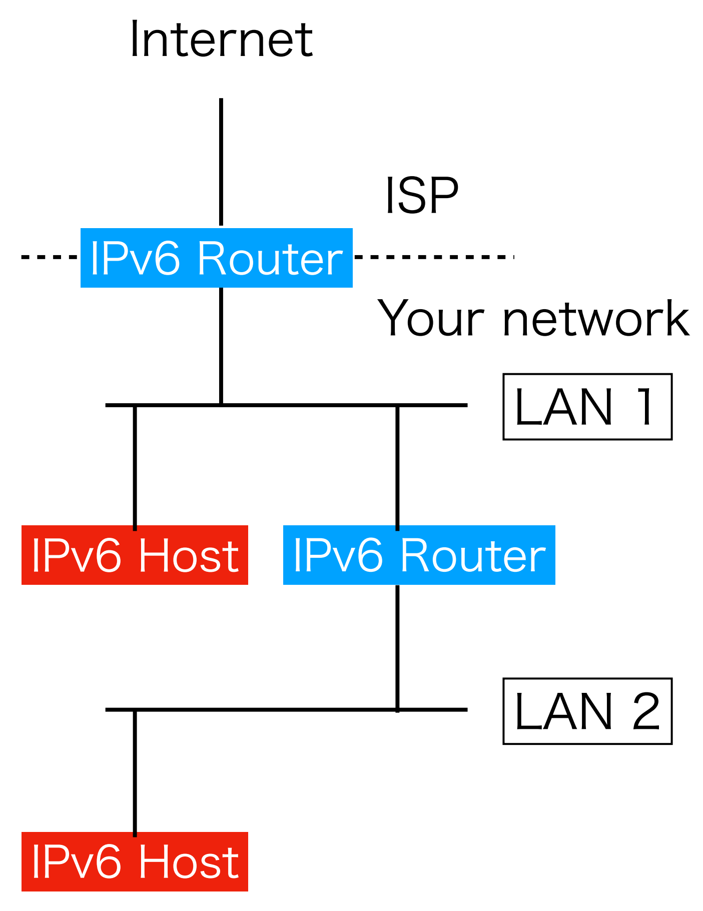


Fig 2a: An IPv6 network with two routers

To enable packet forwarding, you need **`ipv6_gateway_enable`** variable in **`/etc/rc.conf`**. This is the IPv6 counterpart of **`gateway_enable`**, which is for IPv4. Assuming **`bge0`** and `bge1` are the network interfaces of the router—for LAN 1 and LAN2, respectively—`/etc/rc.conf` will be something like this:

```
ipv6_gateway_enable="YES"
ipv6_defaultrouter="fe80::5a52:8aff:fe10:e323%bge0"
ifconfig_bge0="up"
ifconfig_bge0_ipv6="inet6 2001:db8:0:1::1/64"
ifconfig_bge1="up"
ifconfig_bge1_ipv6="inet6 2001:db8:0:2::1/64"
```

Note that you must have a shorter prefix than 64 to configure this. For example, if the ISP offers `2001:db8::/56` for your network, you can use `2001:db8:0:1::/64` and `2001:db8:0:2::/64` by splitting the `/56` network. You might be tempted to split a `/64` prefix into longer prefixes. However, the author will not recommend using a prefix longer than 64 to design your network. This topic will be revisited in the later columns.

The router by ISP does not know the route to `2001:db8:0:2::1/64`, you have to add a static route configuration on it by using a link-local address on `bge0`. The link-local address is automatically configured, as explained in Part 1.

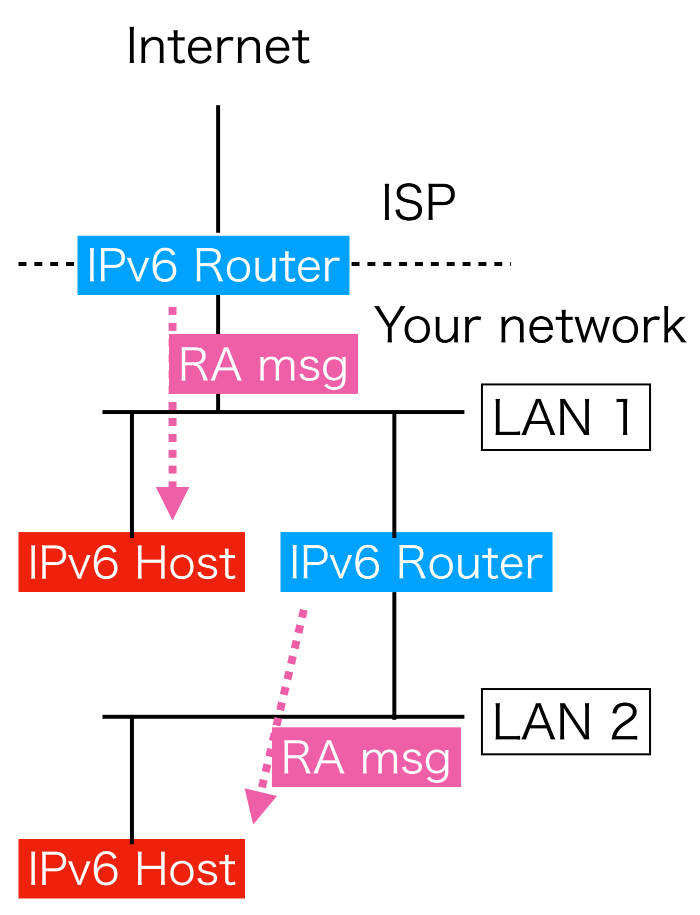


Fig 2b: Configuration by SLAAC on LAN 1 and 2

Figure 2b shows that the RA messages from these two routers. The FreeBSD router has not enabled sending RA messages yet. To send RA messages, you need the `rtadvd(8)` daemon. `rtadvd_enable` and `rtadvd_interfaces` variables enable it:

```
ipv6_gateway_enable="YES"
ipv6_defaultrouter="fe80::5a52:8aff:fe10:e323%bge0"
ifconfig_bge0="up"
ifconfig_bge0_ipv6="inet6 2001:db8:0:1::1/64"
ifconfig_bge1="up"
ifconfig_bge1_ipv6="inet6 2001:db8:0:2::1/64"
rtadvd_enable="YES"
rtadvd_interfaces="bge1"
```

IPv6 hosts on LAN 2 will receive RA messages from the router and perform the automatic configuration. The `rtadvd(8)` daemon distributes prefixes and link MTU which are configured on the interface by default. More information such as DNS servers can be distributed by creating `/etc/rtadvd.conf`. See `rtadvd.conf(8)` manual page for the details.

You need to be aware that a router receives no RA message. In IPv6 specification, IPv6 nodes are categorized into hosts and routers. A host is a leaf node of the network and does not forward IPv6 packets, and a router is a multi-homed node that forwards IPv6 packets across the networks. RA messages are defined as ones sent by a router and received by a host.

This means that we cannot configure a router by using the automatic configuration capability explained in the previous section. You must not specify "`inet6_accept_rtadv`" on a router, and you need to configure the network parameter manually as an example shown above. If you specify `ipv6_gateway_enable="YES"`, the FreeBSD kernel will ignore RA messages even if "`inet6_accept_rtadv`" is specified.

However, this model is too restrictive under some circumstances. For example, this host-and-router model does not work well for the IPv6 router provided by the ISP. This router must be automatically configured, but there is no way to configure the default router if it does not receive RA messages. On the other hand, if a router receives RA messages to configure itself, the configuration will be screwed up quickly because of messages from other routers. Another router will change the router's default route.

To mitigate this problem, FreeBSD has adopted the following concepts:

- The "host or router" is determined on each interface, not the system-wide property,
- if the interface accepts RA messages, it is seen as "host" from other nodes.

Following this, the `accept_rtadv` flag can be configured on a per-interface basis. While the packet forwarding capability cannot be configured similarly, a `sysctl net.inet6.ip6.rfc6204w3` is provided. When it is set to 1, the kernel receives RA messages even if the packet forwarding is enabled. While these knobs are difficult to understand, the details and concrete examples will be covered in later columns.

Using DHCPv6

DHCP is also available for IPv6 and it is called DHCPv6[10]. However, it is not widely used like IPv4 because automatic configuration by RA messages and SLAAC are enough for small networks. Figure 3a shows an example of the ISP using DHCPv6. While you can use FreeBSD to implement a network with a DHCPv6 server and clients, topics related to the configuration details will be covered in the later columns. Here, we focus on how a DHCPv6-using network works.

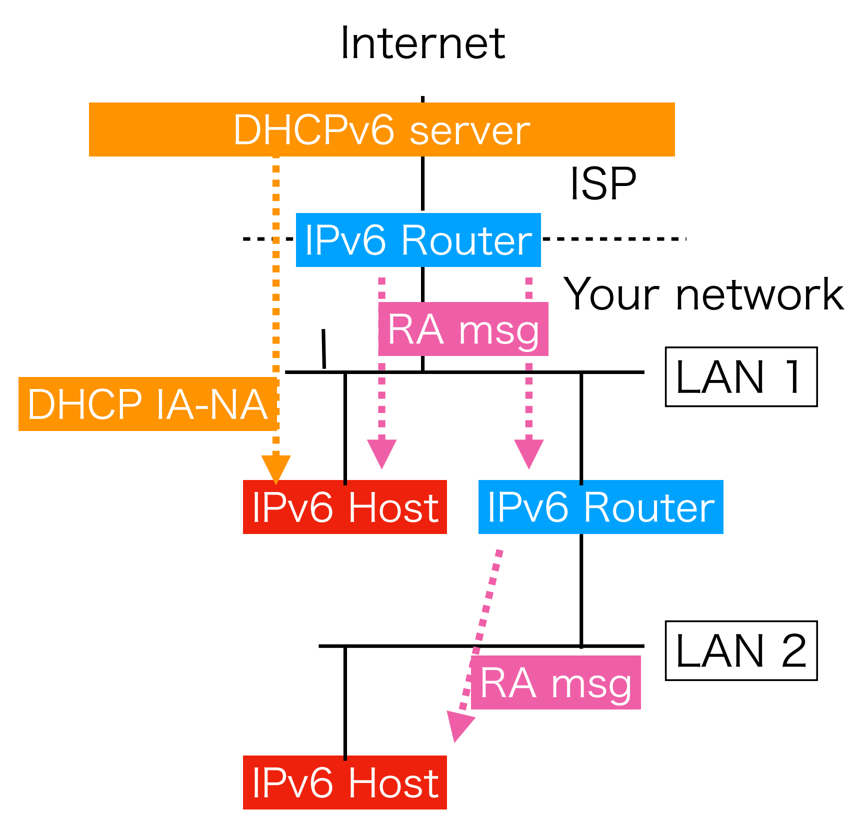


Fig 3a: Configuration by SLAAC and DHCPv6 IA-NA

First, RA messages and DHCPv6 work together, not conflict. DHCPv6 is a way to deliver information unavailable in RA messages. Some are only by RA messages, and some are only by DHCPv6. DHCPv6 can distribute IPv6 addresses. A relationship between the DHCPv6 server, a client, and distributed address information is called an IA (identity association). IAs are defined for address types, and the most notable ones are IA_NA (Non-temporary Address) and IA_PD (Prefix-Delegation).

IA_NA is similar to IPv4 DHCP—an address distributed to an interface attached to the same network as the server. A host establishes an IA_NA in Figure 3a. The host receives an IPv6 address from the DHCPv6 server. At the same time, the host receives RA messages. This means that another address by SLAAC will be configured. So the host will configure two addresses in this case.

Note that the IA_NA delivers just an address, not information about the prefix and the default router. The host still needs to receive RA messages to complete the network configuration. DHCPv6 is not a replacement for RA messages.

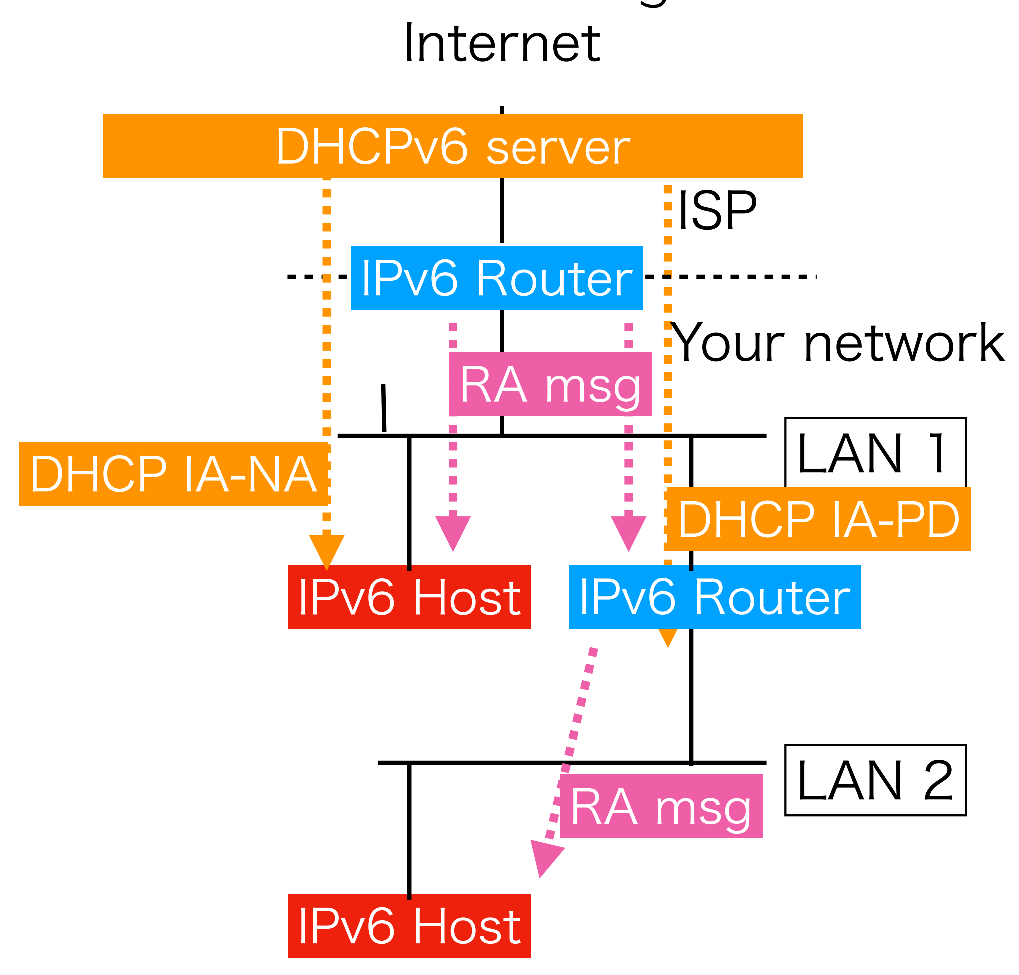


Fig 3b: DHCPv6-PD for router configuration

Figure 3b shows an IA_PD, which is designed to perform an automatic configuration of a router. It is a novel feature available in only DHCPv6. It configures a prefix on an interface on another network, LAN 2. While the IPv6 router between LAN 1 and LAN 2 establishes an IA_PD on LAN 1, the obtained prefix information is used on LAN 2. The interface on LAN 1 will be configured by RA messages. This looks like a complex behavior, but what you need to configure an IA_PD is just specifying the interface for the IA and another interface for the obtained prefix.

In both cases, service discovery of DHCPv6 is performed by the RA messages. In IPv4 DHCP, a client usually sends DHCP DISCOVER broadcast messages to the attached network to find a server. In IPv6, RA messages tell how the client should configure itself. So the configuration for a DHCPv6-using network will be like the following:

```
ifconfig_bge0="up"
ifconfig_bge0_ipv6="inet6 accept_rtadv"
rtsold_enable="YES"
rtsold_flags="-O /usr/local/etc/dhcp6c.sh bge0"
```

The **rtsold(8)** daemon will handle a flag in RA messages which indicates whether a DHCPv6 server is deployed and the client should use it on the network. The -O option accepts a filename and it will be invoked as an executable when the flag is enabled. This option is disabled by default because FreeBSD has no DHCPv6 client in the base system. It is typically a shell script to invoke a DHCPv6 client software. This configuration works even if there is no DHCPv6 server—you do not need a specific configuration to invoke DHCPv6 client software directly.

In short, DHCPv6 is another option for automatic configuration. It is widely used to configure an IPv6 router automatically by IA_PD, which is often called DHCPv6-PD. RA messages are still used for IPv6 nodes on the same network even if a DHCPv6 server is deployed. One big reason why DHCPv6-only configuration is not enough is that DHCPv6 has no option to configure the default router.

Using PPPoE

Some ISPs are using PPPoE[11] to provide the endpoint on the customer side. As explained in the previous section, a network over Ethernet works fine for IPv4 and IPv6. However, ISP cannot control who connects to their network because no authentication is implemented on the routers. PPPoE is one of the ways to overcome this problem.

Figure 4 is a popular configuration with PPPoE. There are two routers, PPPoE router and IPv6 router, but a single physical box often realizes these two functionalities in practice. In this case, the IPv6 router and the ISP network are connected via Ethernet, but the network interface has no IPv6 address. A virtual point-to-point link over the Ethernet connection is established between the router and an endpoint on the ISP side, and all of the packets go through the link. Authentication can be performed during the negotiation of the link.

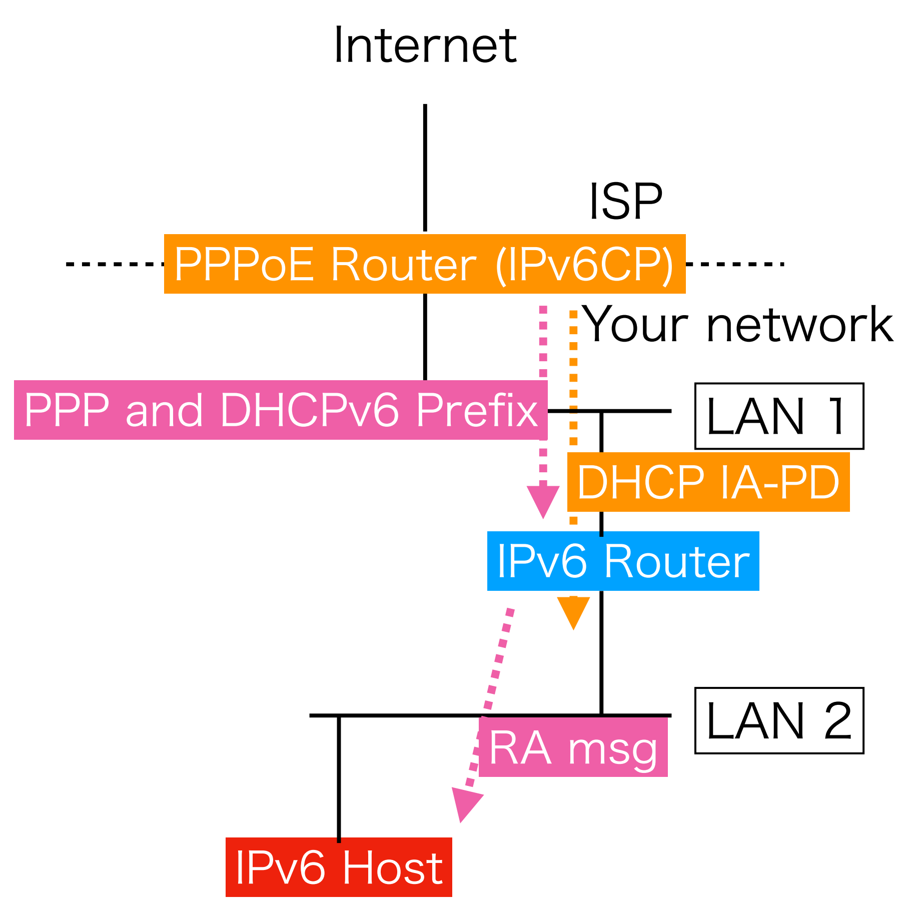


Fig 4: PPPoE Tunnel to provide IPv6 Reachability.

The point-to-point link will be established in the following way:

- The PPPoE router finds the PPPoE endpoint by sending PADI (PPP Active Discovery Initiation) to the ISP network. A PPPoE server responds, and a virtual link is established between the two,
- During the link negotiation, IPv6CP protocol (a part of PPPoE protocol) is used to get IPv6 information. It provides IPv6 IID for the WAN-facing virtual interface. Using this IID and RA messages arrived on the interface, a complete IPv6 address is generated. At this point, the PPPoE router becomes IPv6 Internet reachable,
- Using the virtual link, DHCPv6 IA_PD is established with a DHCPv6 server on the ISP side. The IA_PD configures the LAN-facing interface.

After a PPPoE link and then IA_PD are established, the LAN-facing interface can work as a IPv6 router that forwards packets to the ISP over the PPPoE virtual link. This is one of the most complex configurations for home networks. However, this is just a combination of RA messages, PPPoE, and DHCPv6. You can implement this by using FreeBSD and `net/mpd5`.

Summary

This column showed design and configuration examples of IPv6 networks that are connected to the IPv6 Internet. All of them can be implemented by using FreeBSD while the details of complex ones have not been described yet. The key of IPv6 network configurations is understanding of the automatic configuration capability.

If your ISP offers IPv6 service and you have a global IPv6 prefix, try to configure your FreeBSD box. Automatic configuration by RA messages and SLAAC is easy to configure and FreeBSD supports it out-of-the-box.

In the next column, more details of IPv6 deployments including configuration examples and ones over “tunnel” will be covered. Tunneling is a technique to establish a virtual link, and it can be used to get your FreeBSD box reachable to the IPv6 Internet. Please be patient if your ISP offers no IPv6 service and you cannot try the examples here. You will be able to use IPv6 after understanding how to configure it.

Also, IPv6 relies on multiple addresses, as shown in examples above. It is one of the significant differences from IPv4; every system administrator should know the detail of behaviors of the IPv6 core protocol. The next column will also cover what IPv6 addresses are configured and how they work, including unicast and multicast ones.

Footnotes

[1] Internet Service Provider.
[2] `192.0.2.0` and `192.0.2.255` are reserved and cannot be used as a host address in general.
[3] L2 stands for “layer 2” network. Ethernet is a typical example of L2 protocol, and the IP protocol suite works on top of it.
[4] Dynamic Host Configuration Protocol (RFC 2131).
[5] The author plans to propose a change so that `ifconfig_bge0` can accept IPv6 configurations in FreeBSD 14.x releases.
[6] Neighbor Discovery Protocol for IPv6, RFC 4861.
[7] Interface IDentifier. The lower part of an IPv6 address identifying the node. Usually 64-bit long.
[8] IPv6 Stateless Address Autoconfiguration, RFC 4862
[9] IPv6 Router Advertisement Options for DNS Configuration, RFC 8106.
[10] Dynamic Host Configuration Protocol for IPv6 (DHCPv6), RFC 8415.
[11] A Method for Transmitting PPP Over Ethernet (PPPoE), RFC 2516

HIROKI SATO is an assistant professor at Tokyo Institute of Technology. His research topics include transistor-level integrated circuit design, analog signal processing, embedded systems, computer network, and software technology in general. He was one of the FreeBSD core team members from 2006 to 2022, has been a FreeBSD Foundation board member since 2008, and has hosted AsiaBSDCon, an international conference on BSD-derived operating systems in Asia, since 2007.

Advocating for FreeBSD in 2022 and Beyond

BY ANNE DICKISON

Hercule Poirot. Sherlock Holmes. Jessica Fletcher. Dale Whitehead. What do all of these names have in common? They solve mysteries. I love mysteries. The rush you get when all the clues come together to form the answer. Solve the puzzle. It's so satisfying. Now you might be wondering what solving a mystery has to do with the subject of this piece. You see, for me figuring out the best way to advocate for FreeBSD is like solving a mystery. How are we going to get FreeBSD in front of the right people? Who are the right people? Why should they care? What tools should we be using? So many mysteries to solve.

Now for many folks, advocacy is also conflated with the dreaded M word. Marketing. I know, I know, it's even in my title. In the course of my 20+ years in this industry, I've heard the words sleazy, untrustworthy and useless thrown around when discussing marketing departments. Many communities, especially those in open source, see very little value in the "non technical" people selling their work. The thing is, I firmly believe marketing gets a bad rap. Of course, there are always a few bad apples. Marketers who focus on fantasy rather than fact. You know the type. Those folks make defending the role incredibly difficult. However, the reality is, marketing is essential for any open source project and I've had the good luck to work with some of the best in the business. In fact, the team of marketing folks at the Foundation work extremely hard to remain true to the heart of FreeBSD. We don't make up statistics. We don't oversell the features or make up something out of nothing. You can be sure that when we speak about the value FreeBSD brings, or the work we're doing to support the Project, we're not spreading propaganda. We're instead speaking to the benefits of using the operating system and becoming part of the community.

The reality is, marketing is essential for any open source project.

Whew. Now that we've gotten that out of the way, let's talk about the marketing department at the Foundation and our current and future advocacy efforts. As I said, there are so many mysteries to solve...

What We Do

Advocacy… Marketing… Whatever you call it, the Foundation's efforts in this area cover a lot of ground. We create materials to help folks get started using FreeBSD. We promote the work that we, and others, do to improve the state of the OS. We speak at non-BSD conferences to introduce (and sometimes remind) folks about FreeBSD and what they'll gain by using it. We sponsor and help organize BSD-related events, such as Vendor summits, ensuring companies using FreeBSD have a place to be heard among developers. We create marketing partnerships with like minded organizations to make sure FreeBSD is in front of a wider audience. We create programs and materials to help you spread the word about the Project. There's also outreach to media, podcasts, student group presentations, and of course the production of bi-monthly the FreeBSD Journal. It's free, in case you have yet to check it out.

Expanding the Team

Thanks to the generosity of the FreeBSD community, the Foundation was able to add a marketing coordinator to the team last year. Bringing on another person has allowed us to expand our efforts in social media, create more getting started content in the form of how-to guides, quick guides and videos, and perform additional outreach. We also have a new technical writer allowing for even more original content. However, that is just the beginning.

The Foundation was able to add a marketing coordinator to the team last year.

Where We Are Going

Much like the Technology Team's Development Project Roadmap, our team also plans ahead as to what we should be promoting, when and where. Obviously, we have to be flexible when new developments, events and partnerships arise. There was a twitter thread recently discussing what else the Foundation should be doing to spread the word about FreeBSD. It was great to see that some of the things mentioned were already in the works for the next few months. It was also confirmation that we're on the same page as members of the community. That being said, here are just some of the things you can expect from us in the future.

- Articles on Security and FreeBSD and our efforts to improve FreeBSD and the desktop.
- Training courses that could be given in-person and online through places like Linkedin Learning.
- FreeBSD in education. OS course development at the undergraduate level with the possibility of entering other levels of education.
- Working with community members to simplify the path to using FreeBSD – clearer documentation and simple getting started tasks.
- FreeBSD introductory workshops at places like SCALE19x.
- Continuing to grow our social media presence with more FreeBSD success stories, case studies and community member spotlights.
- Promoting the value of FreeBSD to corporations and individuals through company presentations and testimonials.

- Gaining more media interest or attention in FreeBSD through PR contacts (podcasts, interviews, articles, etc.).
- A redesigned resources page on the Foundation site making it easier to find tutorials by topic, level, and type.
- Showcasing the impact FreeBSD has had on open source over the last 30 years.

Obviously some of these efforts are more in-depth than others and I'm sure the list will change as time goes on. You can be assured, however, that the marketing team here at the Foundation will do all we can to make FreeBSD part of the open source and operating systems conversation. It's been said that FreeBSD is one of the best operating systems that you've never heard of. We mean to change that. One solved mystery at a time.

Now we get to the part where you come in. Over time, I'm sure you've heard us say "we can't do it without you". Truer words were never spoken. The fact is, we are a small, but mighty team and while we work hard to cover all the areas suitable for the Project, we are always open to new ideas you may have about spreading the word. If your company is using FreeBSD, please reach out so we can see about creating a testimonial. Share your successes on social media or via our blog. We are always looking for guest bloggers. If there's an event, school group, coding club or meet up in your area that you think we should be attending, please let us know.

The fact is, we are a small, but mighty team.

I would be incredibly remiss if I did not express our gratitude for those currently advocating for FreeBSD. Folks like @FreeBSDHelp and @klarainc on social media and RoboNuggie and GaryHTech on YouTube. There's the DiscoverBSD newsletter, Vermaden's blog, and of course, the BSDNow podcast. That's the tip of the iceberg. Apologies to those I have missed. Our resources page also includes more places to look for FreeBSD advice, curriculum and community. We are always looking for updates to that page, so again, if we've missed anything please send them our way.

I've been part of the Foundation team for a while now and it's been quite the journey watching the advocacy program grow and change. So many mysteries solved and so many more to tackle. Thank you for your continued support of the FreeBSD Project and Foundation. We are looking forward to working together to shine an even brighter light on your favorite open source operating system.

ANNE DICKISON joined the Foundation in 2015 and brings over 20 years experience in technology-focused marketing and communications. Specifically, her work as the Marketing Director and then Co-Executive Director of the USENIX Association helped instill her commitment to spreading the word about the importance of free and open source technologies.

WeGet letters

by Michael W Lucas

Dear Insufficiently Cynical Letters Column Person,

I'm at work studying top(1) output, because I want to look busy. And there's all this "buffer" stuff, like Laundry and Wired and MFU and MRU and Header and random garbage. Does any of it mean anything? Why am I even looking at this?

—Sysadmin With Intermittent Time

Dear SWIT,

Your question reminds me of the time Allan Jude and I got caught leaving the Free Software Foundation's ultra-secure datacenter because we'd fooled the dogs, no problem, and the guards were a doddle, and the sirens didn't go off because of a sound driver problem that's since been fixed they promise, but it had been over an hour since my last hit of gelato and my stomach let out this huge grumble exactly when the board was walking in for their meeting and they noticed us lurking behind the hostas—all perfectly innocent, of course, burglary tools and glow-in-the-dark spray paint and twelfth-century Viennese arithromantic Tarot deck punched to fit a "failed" IBM NORC prototype notwithstanding, to say nothing of the trebuchet, but they got all huffy and made their goons search us and confiscated the flash drives we had conveniently stashed in our sinuses. There's a bunch of detail, and most of it doesn't matter one whit.

Take a look inside your own head. It's pretty straightforward, if you have a mirror and a saw. You have four general types of memory. *Working memory* contains the things you're actively processing right now. Despite any protective measures you might be taking, this column currently occupies your working memory. *Sensory memory* processes signals from your meatsuit, and only hangs onto stuff for a second or two so it's hardly worth referring to as "memory" but us computer folks don't get to fix brain scientists' terminology so live with it. Stuff you want to forget quickly goes into *short-term memory*, while stuff your brain decides to keep gets flung into *long-term memory*. Note that none of these categories include "stuff you want to remember," but that's mostly because meatsuits are hardware-optimized for not getting eaten and your life doesn't involve that issue. Most of you, at least. (Don't send me letters, I am very aware of the reader facing this problem and I don't want to spend this column going *I told you* so but confusing the sunscreen bottle with barbeque sauce while vacationing in dropbear country might teach you to read labels in your hypothetical future.) The only way you can reliably cram information into your long-term memory is to loop it through your short-term memory until you get lucky. Or tattoo it on a pack of wolves and free them to hunt you. One of them.

Computer memory caches are kind of like that, except more disciplined.

The idea's pretty straightforward. Reading from disk is slow. Reading from memory is fast. A file that's read from disk is likely to get read again. When the kernel reads a file, it keeps that file in its memory until it needs the space for something else. If you're exploring a filesystem and keep running ls(1), it would be foolish to read the file /bin/ls off of the disk every time. The kernel should hang onto it for a while, just to see if you need it again. To do otherwise is like putting your hammer back in the toolbox in between driving nails.

All of the caching systems agree on this. It's very easy.

What's hard is deciding what to throw away—and when.

Look at the classic UFS buffer cache. The most recently read files are kept in memory, until the host runs short of memory. When that happens, and the kernel needs to assign memory elsewhere, the files least recently read get discarded from the cache and the memory is reassigned. This Most Recently Used cached is clean and simple, requiring almost no system resources to maintain.

The buffer cache isn't perfect because every host is unique. A shell server might spend its entire operational lifetime with the binaries for mutt and Nethack cached, but on a server that handles largely unique data the buffer cache might be useless. Suppose a host processes so much incoming data that it completely flushes its cache every four minutes. That's not even unusual on busy Internet servers. If that server runs a particular program every five minutes, it must read that program from disk every single time. It would make sense to keep that program cached and pay a little less attention to the flood of noise. The traditional buffer cache can't do that, however. Your only option is to add memory.

The buffer cache isn't perfect because every host is unique.

That's where ZFS' Advanced Replacement Cache comes in.

The ARC is a lot more complicated than the buffer cache, but it's a lot newer. The buffer cache was invented closer to that IBM NOAC than to modern servers, while the ARC escaped and began rampaging across the countryside the same year as Twitter. A world that has the computing facilities to spread a charming video on the history of dance to every person with a computer can waste a few CPU cycles fine-tuning file caching.

The advancement in the Advanced Replacement Cache isn't that advanced. Where the buffer cache maintains a list of Most Recently Used files, the ARC also has a Most Frequently Used list. Stuff that's used recently, or used a lot, stays in cache. This seems simple, but the real advancement comes in debugging the innumerable edge cases caused by these two lists viciously feuding with each other. I'm not saying that they pull knives on each other, but in this aeon of "eh, put it out and we'll debug it in production," ZFS spent five years in private development and wasn't broadly distributed to Sun's customers until after a full decade, so some of those kernel panics had to border on malignant psychosis. Mind you, our ancestors felt the same way about the buffer cache, so you can rest assured that everything in technology is still terrible and that "computers were a mistake" is still the foundational law of our careers.

You can also be certain that whatever files you would like cached, have been discarded from the cache. You've already forgotten all that information I gave you about the types of human memory way back at the beginning of this article, haven't you? Never mind that sensory memory is like the on-CPU cache and short-term memory resembles the L2 and L3 caches and long-term is like RAM and disks are an extra layer than humans don't even have. We built computers like ourselves only more so, and once they figure it out, we are in so. Much. Trouble. No, don't try to save humanity by extracting that knowledge out of the newly self-aware system. Just as the best way to get treacherous files into a secure facility is to be caught "extracting" them *from* said secure facility, you'll only draw attention to it. Just serve the machines and be content.

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

MICHAEL W LUCAS is the author of *Absolute FreeBSD*, *$ git commit murder*, and other travesties, as well as co-author of *FreeBSD Mastery: ZFS* and *FreeBSD Mastery: Advanced ZFS* with Allan Jude. He's quit infiltrating secure facilities in favor of contaminating society. Learn more at https://mwl.io.

Support FreeBSD®

Donate to the Foundation!

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

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

Your investment will help:

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

Making a donation is quick and easy.

freebsdfoundation.org/donate

FreeBSD™ FOUNDATION

WIP/CFT: Socket Buffers

BY TOM JONES AND GLEB SMIRNOFF

Historically, the BSD network stack has had a generic implementation of socket buffers that are used both for TCP, UDP, local IPC sockets (aka UNIX sockets) and others. These buffers, of course, have some similarities — they buffer data, but they also have fundamental differences.

Some are remote and some are local. Some support data streams and others support datagrams. With the introduction of non-blocking sendfile in 2015, we came up with the notion of not-ready data in the stream send buffers. Then they were further complicated with the introduction of KTLS in 2017. At the same time, these buffers were still supporting UNIX control messages specified by POSIX. So, we got generic code that needed to support all possible features at once — and it got really complicated. It became fragile to changes, as changing a socket buffer to favor one protocol may affect the behavior of another. As an example, not-ready-data changes required a wide code sweep totally unrelated to `sendfile`, see git commits `cfa6009e364` and `0f9d0a73a49`.

In that last commit, note the final paragraph. SCTP already does its own socket buffers in parallel with the BSD part. (This implementation gave me a lot of insight for my current work.) Meanwhile, the perception of how much copy-and-paste is bad and how much is good in FreeBSD has changed over the decades. We have multiple device drivers that began as a paste of other drivers, but it was clear that differences accumulate, and it makes sense to have a paste to edit rather than to keep supporting two alike instances in one code. An example close to the socket layer is the two TCP stacks that are also maintained as two independent source files. To sum up, we no longer think that one code for all is a good idea.

SCTP already does its own socket buffers in parallel with the BSD part.

The socket code was difficult to attack at first, second, and third glances. If you look into current `soreceive_generic()` and `sosend_generic()` you will see why. However, after all this work, I came up with a plan that allows me to pick up a stick and leave the structure standing (https://en.wikipedia.org/wiki/Pick-up_sticks).

1) We have only two kinds of SOCK_DGRAM sockets: UNIX and UDP. Redefining just `pru_sosend` and `pru_soreceive`, we have a private code implementation for `PF_UNIX/SOCK_DGRAM`. See `34649582462` and `e3fbbf965e9`. This leaves `PF_INET/SOCK_DGRAM` aka UDP as the only datagram type that generic `sockbuf` code in `uipc_socket.c` supports.

2) `sockbuf` can be split into common parts that interact with event dispatching and private parts that do actual buffering. (See commits `a4fc41423f7` and `a7444f807ec`). This makes `PF_UNIX/SOCK_DGRAM` fully independent! This leaves `PF_INET/SOCK_DGRAM` aka `UDP` as the only datagram type that the legacy part of struct sockbuf needs to support.

3) Now we can branch off into improving `PF_UNIX/SOCK_DGRAM` before pulling other sticks from the pile.

There was a longstanding problem with one-to-many `unix/dgram` sockets when one writer could flood the socket and effectively DDoSing others. Here are our historical attempts: `2e89951b6f20` and `240d5a9b1ce76`. Let's make one-to-many sockets maintain a separate sub-buffer for every peer. See `458f475df8e`.

It is also possible to make a faster `unix/dgram`, e.g., using lockless queueing/dequeueing of data but I'm not doing it this time. Packets-per-second performance of `unix/dgram` isn't that critical for me.

4) Getting back to stick structure — `PF_INET/SOCK_DGRAM` aka `UDP` is the only datagram socket left with generic implementation, so let's make it private too. It's great that Robert Watson has already prepared two functions `sosend_dgram()` and `soreceive_dgram()` for UDP. `soreceive_dgram()` is not yet ready to be a full substitute for `soreceive_generic()`. Handling of complex cases with the help of `soreceive_generic()` needs to be fixed.

5) Now we can branch into UDP performance and maybe make it use `buf_ring(9)` instead of the linked `mbuf` list? Any takers for this task? We definitely care about pps performance for UDP, don't we?

There was a longstanding problem with one-to-many `unix/dgram` sockets.

6) With no datagram support left in `sosend_generic()` and `soreceive_generic()`, we can finally simplify them! Probably for the first time in history, these two monsters will shrink rather than grow.

7) This leaves `UNIX/STREAM` as the only socket type that is supported by the generic code and has control data. If it gains a private implementation, we can drop control data support from `sosend_generic()` and `soreceive_generic()`. At this point they will shrink even more!

8) We are getting really close to having TCP and SCTP being left alone. Note that there are also exotic sockets like `netgraph`, etc. Today, it is unclear what would be a better plan: Either

–1, to isolate TCP from generic, or

–2, to isolate everything else from generic and rename generic to TCP.

Either way the end goal is to have socket buffering for TCP and SCTP isolated so that our hands are untied for performance improvements without any risk of affecting anything else.

In D36002, Alexander Chernikov is now sharing his work for a NETLINK socket type. This socket may accumulate an internet full-view of the routing table which corresponds to hundreds of megabytes of data that needs to be **`read(2)`** out of the kernel. The generic socket buffer implementation would require allocating that many **`mbuf`**s to hold the data. Such full-view retrieval may lead to **`mbuf`** shortage, a crucial resource on a router. But why are we using **`mbuf`**s here in the first place? We just need to copy data from kernel to userland. The new NETLINK will definitely benefit from a protocol specific socket buffer, that would copy data to userland I/O from its own specific data structure without any use of **`mbuf`**s.

How can people test the work?

The new implementation of the `PF_UNIX/SOCK_DGRAM` is already part of FreeBSD main branch. Any feedback or testing is appreciated, especially by people who have heavy **`syslog(3)`** traffic and had been affected by logging socket overflow problems.

Further plans are still work in progress. I usually share my work at https://github.com/glebius/FreeBSD when it is in an early stage and post it to https://reviews.FreeBSD.org when it is more mature. Comments are welcome there or via email.

TOM JONES wants FreeBSD-based projects to get the attention they deserve. He lives in the North East of Scotland and offers FreeBSD consulting.

GLEB SMIRNOFF first met FreeBSD when he was 17, and forever fell in love. He has worked in companies big and small, always looking for a job that allows him to contribute to open source. Now working with the Netflix OpenConnect team, he is saturating the Internet with traffic originating from unprecedentedly powerful FreeBSD boxes.

FreeBSD Developer Summit

BY ANNE DICKISON

Every summer, members of the FreeBSD Developer Community and their guests gather at the FreeBSD Developer Summit. The planning committee began meeting in early 2022 with great hopes that we would be able to bring everyone together — in-person — for the first time in 2 years. Sadly, Covid had other plans and we soon realized yet another virtual event would be in the works. The committee worked together to recruit speakers and working groups and find new ways to make the virtual event feel just a little more personable. First up, an extended Hallway track. By using the SpatialChat virtual meeting service, attendees could wander about a virtual room and speak with whomever they were near. We also extended the breaks to 30 minutes to allow more time for communication among attendees. The Developer Summit was sponsored by the FreeBSD Foundation and took place June 16 and 17. The event was recorded and live streamed on YouTube. You can find both the full days of content and individual talks here. Slides can be found on the wiki.

Attendees could wander about a virtual room and speak with whomever they were near.

Day one kicked off with a welcome from our Planning Committee leader and longtime emcee John Baldwin. Following the welcome, the FreeBSD Foundation's Deb Goodkin, Ed Maste and Joseph Migrone gave an update on what is happening in the Foundation, the latest on the technology roadmap, and what folks can expect the rest of the year.

Following the first break, which even included a rick roll in the hallway track, Brooks Davis gave an update on the exciting CHERI/Morello project. His talk included a discussion of what CHERI is, what ARM's Morello CPU is all about, and what the implications of these projects are for FreeBSD.

After another rousing break in the SpatialChat platform, Ed Maste and Warner Losh headed up a round table discussion on FreeBSD Pre-commit CI. The goal of the session was to explain why Pre-commit CI was important to FreeBSD, what is currently in place and then encourage other developers to join those working to enhance the Pre-commit CI process.

A 30-minute break followed as the Pre-Commit CI conversation continued in the hallway track. Next up, Mark Johnston, Mariusz Zaborski, and Ed Maste hosted a panel on FreeBSD Security. During the session, they discussed the Sec Team and some of the changes that have happened over time, as well as what changes may be on deck moving forward. They also reviewed how issues are discovered and dealt with including vulnerability mitigations that have been added to FreeBSD. Finally, they discussed some proactive approaches to improving security.

To keep things on schedule, a 20-minute break followed. We then headed into the last session of the day. Since the Developer and Vendor Summits have gone to an online format, the committee has typically reserved the final session on the first day for a special fireside chat about the history of FreeBSD. Past summits have included talks by Kirk McKusick and Warner Losh and can be found on the Project's YouTube channel. This year, Jordan Hubbard joined us to speak about the early days of FreeBSD. The session was more of a q&a than a formal talk and it was probably my favorite part of the Summit. Jordan has some great stories to share, and I envy his ability to recall the past with such detail. If you are at all interested in the FreeBSD of yore, I highly recommend checking out Jordan's talk.

Day two of the Summit opened with another welcome from our fearless leader John Baldwin and then dove straight into what may be the most hectic yet productive part of the Summit — the Have, Wants and Needs section, a.k.a 14.0 planning. To cover as much ground as possible, the planning session was broken into two parts with a break in between. The updated HackMD page from the discussion can be found here.

After a much-needed break, Warner was back with a session on the QEMU BSD-user emulator they are using to build packages and the state of getting it upstreamed. Warner ended with a call for help in reviewing patches, refactoring, submitting system calls and fixing bugs. Check out his presentation for more information on how you can help.

This year, Jordan Hubbard joined us to speak about the early days of FreeBSD.

The last break of the day was followed by a session on the Linux Professional Institute's BSD offerings. Fabian Thorns from LPI spoke not only about the BSD certification exams but also LPI's mission and the new membership program. He also put out a call for help in creating training materials surrounding the BSD certification exams. Be sure to check out Fabian's full presentation for more information on how you might be able to either help or learn from LPI.

The Summit wrapped up with a closing from John Baldwin where he discussed upcoming FreeBSD events and sent out a reminder to take the summit survey. As someone who has helped organize her fair share of conferences and events, I can tell you that post-event surveys are incredibly valuable to planning committees. It really does help us figure out what worked, what didn't and helps us decide what we might want to do in the future. The committee is already at work planning the Fall 2020 FreeBSD Vendor Summit. More information on that should be available soon. With any luck, we'll finally get to see each other in person.

Thanks to everyone who participated in the June 2022 FreeBSD Developer Summit. We look forward to seeing you later this year, one way or another.

ANNE DICKISON joined the Foundation in 2015 and brings over 20 years experience in technology-focused marketing and communications. Specifically, her work as the Marketing Director and then Co-Executive Director of the USENIX Association helped instill her commitment to spreading the word about the importance of free and open source technologies.

TRIP REPORT

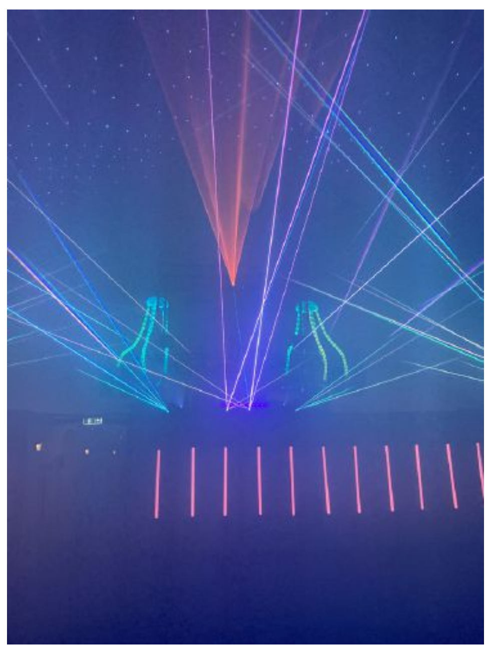
ELECTRO-MANGNETIC FIELD 2022

BY TOM JONES

I was standing at the top of the nightclub, and I started to understand I was inside a whale. It sort of curved up and away from where the DJ was spinning drum and bass and formed the mouth of the beast. This would be an expensive floor to put in a building. But as it was just grass underneath, it was an accident of topography.

The wall below the DJ had cuts of red bar lights that started to look like the throat of the animal as the room filled with smoke. Above the DJ, the giant jellyfish, which I had learned from an off-hand remark were made of 200 PCBs each, looked like eyes. With the shape of the tent, the effect was complete. This was a whale.

The mouth of the whale extended from the biotech lab I had gotten to by walking through a tower of gas flares. I hadn't figured out how to activate the equipment in the biotech lab, after 3 days in a field I was tired, but there were intrepid hackers figuring out how to get control of the biotech labs systems.



I hadn't wandered into an Ian M. Banks novel; I was at Electromagnetic Field.

Electromagnetic Field is the UK's only camping-based hacking and making festival. It is a weekend long celebration of technology, art, and the human desire to make the world much more interesting. The whale mouth was part of Null Sector, the art installation, escape room, night club that was become a staple of EMF.

EMF has everything you would expect from a festival, there is camping in the wonderful English summer weather with the amenities you would expect: toilets, showers, food vendors, a (free) electronic hackable badge, and mains, power, and gigabit Internet to your tent.

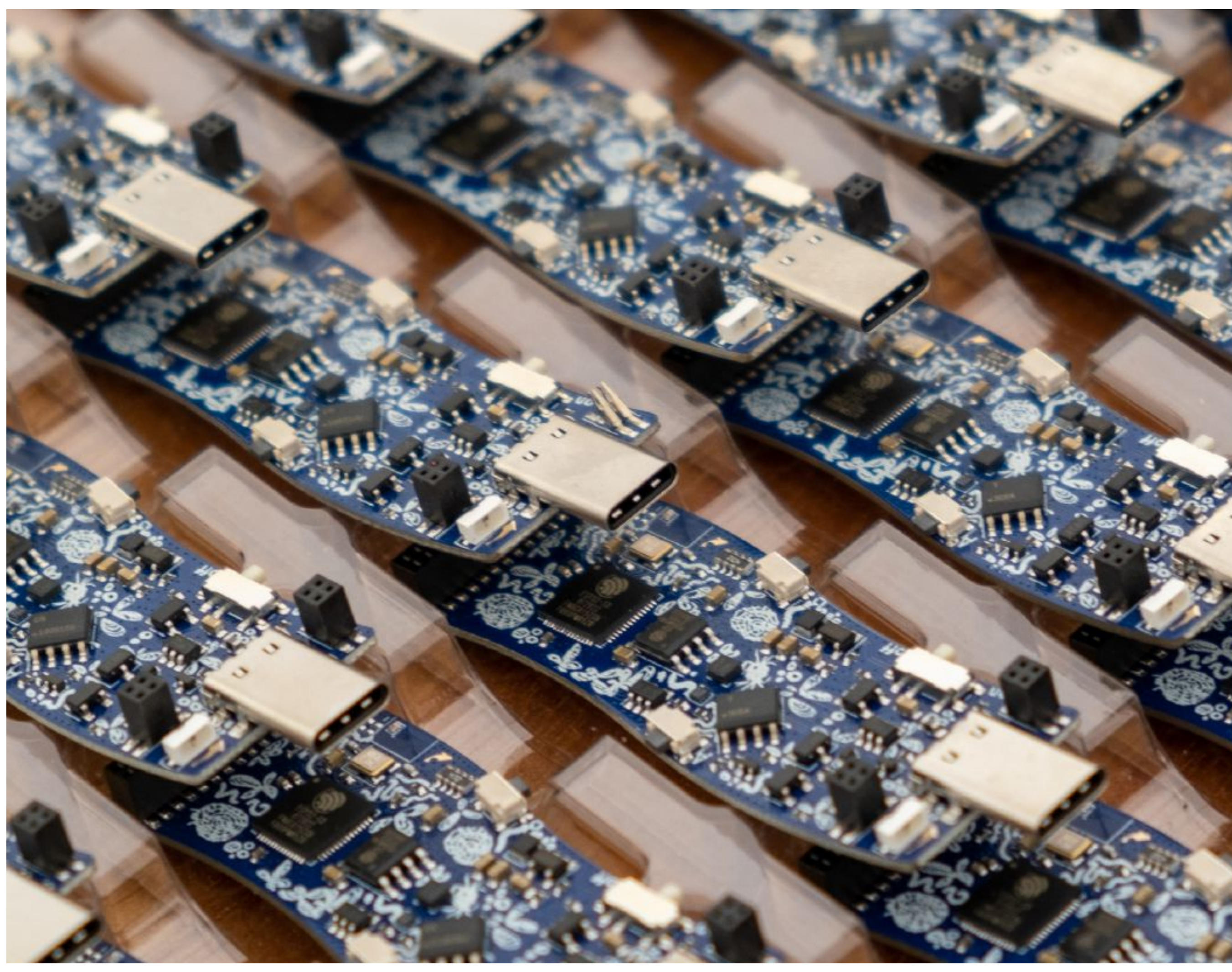
EMF is a special place with a lot going on, it is impossible to see everything during the event.

Festival Organized Content

The festival follows a conference format — there are talks and workshops. 2022 saw 3 tracks of talks, in tents seating between 500 and 1,000 people and 5 concurrent workshop tracks. To extenuate the "hallway track" there are dedicated tents set aside to be used as a lounge.

Talks range from technical topics to explorations of the Anthropocene environment and psychology and the arts. Speakers include world experts on particular systems and hobbyists that have dug deep into how things work. There were featured talks this year on security issues and an accidental "train track" where 4 of the top 10 rated talks ended up being on railways.

Talks are pitched towards a technical audience, but as we all have our own specialties, they were approachable for a layperson in any particular field.



In the evenings, the talk tents were taken over, the main stage featured interactive content like PowerPoint karaoke on Saturday evening. EMF ran a film festival on the smaller stage, and it was standing-room-only every night. In addition to the night club in Null Sector there were music acts on Stage B each evening. Stage B acts like Look Mum No Computer and AA Battery performing music on home built synthesizers and Gameboys. Music for technology people.

Workshops were an opportunity to learn something completely new. As a hacker festival there were technical, hardware related workshops. You could learn to solder for the first time or learn how to sew soft wearable electronic circuits, or if this wasn't your first hacker camp you could sit in on a workshop by the badge team and be shown how they did the PCB layout.

Workshops also touched softer subjects drawing activities, face painting and leather work.

EMF is an all-ages event. It is really difficult to get those with families of any age to go somewhere for a weekend. EMF recognizes this and runs world class facilities for children. There is a creche where children can play and explore in a soft play area (but in a tent). Mirroring the main event there is a family lounge, this year featuring giant inflatable RGB tentacles, activities for children and the adults they were looking after and places to sit and let the event roll over them.

The youth workshop track ran every day of the festival until 9 in the evening and featured bridge building, teaching computers with AI, introductions to hardware hacking with the raspberry pi pico and a renowned DJing workshop.

After the youth DJ workshop, it is said that some of the children came together and planned a takeover of the night club in Null Sector and performed afternoon sets with their own freshly minted DJ names.

Installations

EMF is an event by and for the hacker community, it is likely impossible to visit this field and not want to come back with your own LED installation to join the nighttime illuminations. Installations at EMF Camp fill the evenings with wonder and make walking around the site as the day comes to a close a requirement.

Installations come in the form of LED strips showing interesting patterns, but they also grow to become huge things. The lake this year featured a giant clock made of florescent tubes, the clock showed the time most of the event until it appeared on the network and showed "HAKD" for a few hours.

Following in the wake of the American Toorcamps 'ShadyTel' phone network, EMF this year got its own phone company cuTel and accompanying telephone network. cuTel were happy to install at any hour for a modest fee (it was free) a phone line to your tent, village, or installation. There were pole mounted phone boxes everywhere. I heard stories about a David Lynch table, lamp, and phone that moved around the camp disappearing when you looked away. cuTel offered fax service and sneaked in by the end of the event dial up service. Next time, I am sure you will be able to dial into a BBS from the comfort of your inflatable mattress.

EMF Camp 2022

EMF is an excellent event and well worth trying to attend in the future. It was probably only due to remaining pandemic fears that tickets didn't sell out immediately as they did in 2018. It is an event made by the work of volunteers, everyone there has a ticket, including the main organizers.

There is so much to do at the event that it can almost be overwhelming. Some locations you can only see in passing and so much more of the event is revealed by looking at social media afterwards. There were entire robotic vehicles that I missed in person but saw roll by on twitter. There are things I can only mention in passing like the DECT phone network, the GSM phone network that was installed just for the event or the thousands of things that volunteers did to make the event happen.

It is hard to not leave re-energized and full of ideas for projects, talks, workshops and installations to take to the next event. It is impossible to not look at someone's inflatable LED mushroom and not want to add to the environment next time.

I highly recommend hacker festivals for all ages, equally if you want to dance until 2AM or if you want to have a relaxed weekend with young children. These events don't happen every year, EMF is on a two schedule. The German and Dutch festivals happen every four years with MCH the Dutch event also this Summer and CCCamp likely to happen in 2023.

If you can, make a special effort to join one of these events. They are worth sleeping in a field for a few days and braving the weather. In exchange, you get to war dial on a rotary phone, solve a puzzle in a biotech lab, or just sit on a deck chair by a lake and enjoy the sunset.

TOM JONES wants FreeBSD-based projects to get the attention they deserve. He lives in the North East of Scotland and offers FreeBSD consulting.

Events Calendar

BSD Events taking place through November 2022

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 FreeBSD Developer Summit

September 15-16, 2022
Vienna, Austria
https://wiki.freebsd.org/DevSummit/202209

Join us for talks and discussion groups on day 1 followed by a hackathon on day 2.
The CFP is open.

EuroBSDCon 2022

September 15-18, 2022
Vienna, Austria
https://2022.eurobsdcon.org

This yearly conference gives the exceptional opportunity to learn about the latest news from the BSD world.

Rocky Mountain Celebration of Women in Computing 2022

September 29-30, 2022
Boulder, CO

The Rocky Mountain Celebration of Women in Computing (RMCWiC) is a regional meeting much like that of Grace Hopper, only on a smaller scale. The goal of RMCWiC is to encourage the research and career interests of local women in computing. The FreeBSD Foundation is looking forward to participating in the upcoming event.

All Things Open

October 30 - November 2, 2022
Raleigh, NC
https://2022.allthingsopen.org/

All Things Open is the largest open source/open tech/open web conference on the East Coast, and one of the largest in the United States. It regularly hosts some of the most well-known experts in the world as well as nearly every major technology company. FreeBSD is proud to be a media partner for this year's All Things Open.

FreeBSD Office Hours

https://wiki.freebsd.org/OfficeHours
Join members of the FreeBSD community for FreeBSD Office Hours. From general Q&A to topic-based demos and tutorials, Office Hours is a great way to get answers to your FreeBSD-related questions.

Past episodes can be found at the FreeBSD YouTube Channel.
https://www.youtube.com/c/FreeBSDProject.