November/December 2022

FreeBSD® JOURNAL

Observability and Metrics

Writing Custom Commands in FreeBSD's DDB Kernel Debugger

DTrace: New Additions to an Old Tracing System

Certificate-based Monitoring with Icinga

activitymonitor.sh

Pragmatic IPv6 (Part 4)

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

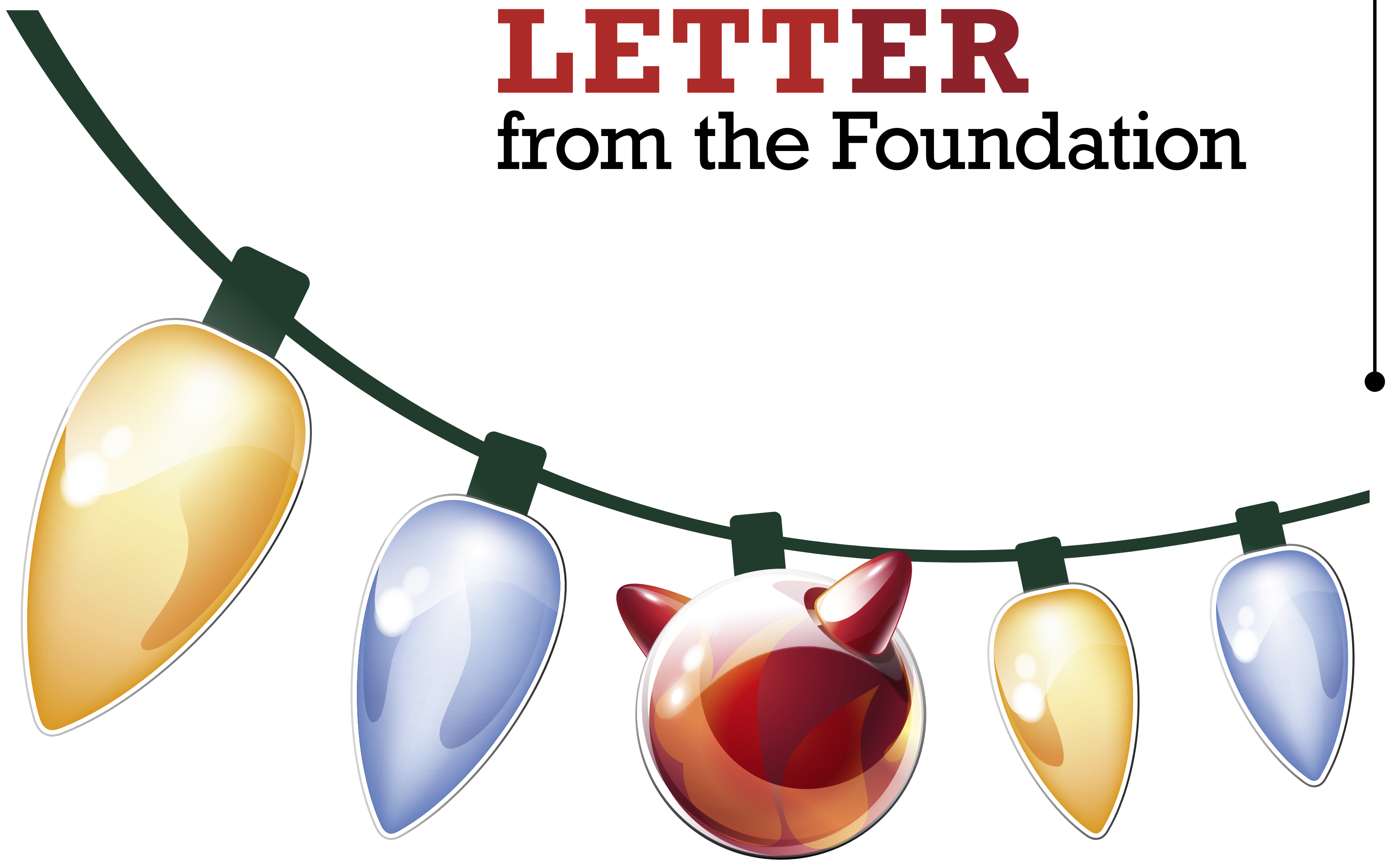
Editor-at-Large • James Maurer
jmaurer@freebsdjournal.com

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



Dear Readers,

As we embark on the 10th year of publishing *FreeBSD Journal*, I could not be prouder of the investment we've made to produce this high-quality publication for you. When reflecting on the past year's issues, the first thing that stands out to me is the quality of the Journal's content which is written by noteworthy experts in their fields and on a volunteer basis. We are also thankful for the outstanding, volunteer editorial and advisory boards that plan future topics, wrangle authors, and make sure we provide the most helpful and timely content to readers around the globe.

The *FreeBSD Journal* delivers informative and interesting content for folks in our community and beyond with a fresh issue every two months. It reaches anyone interested in Unix, operating systems, system administration, research, product development and more. The FreeBSD Foundation funds this professionally produced publication and makes it widely available because we view it as an investment for anyone who uses what they learn from it to further advance the FreeBSD operating system and the community.

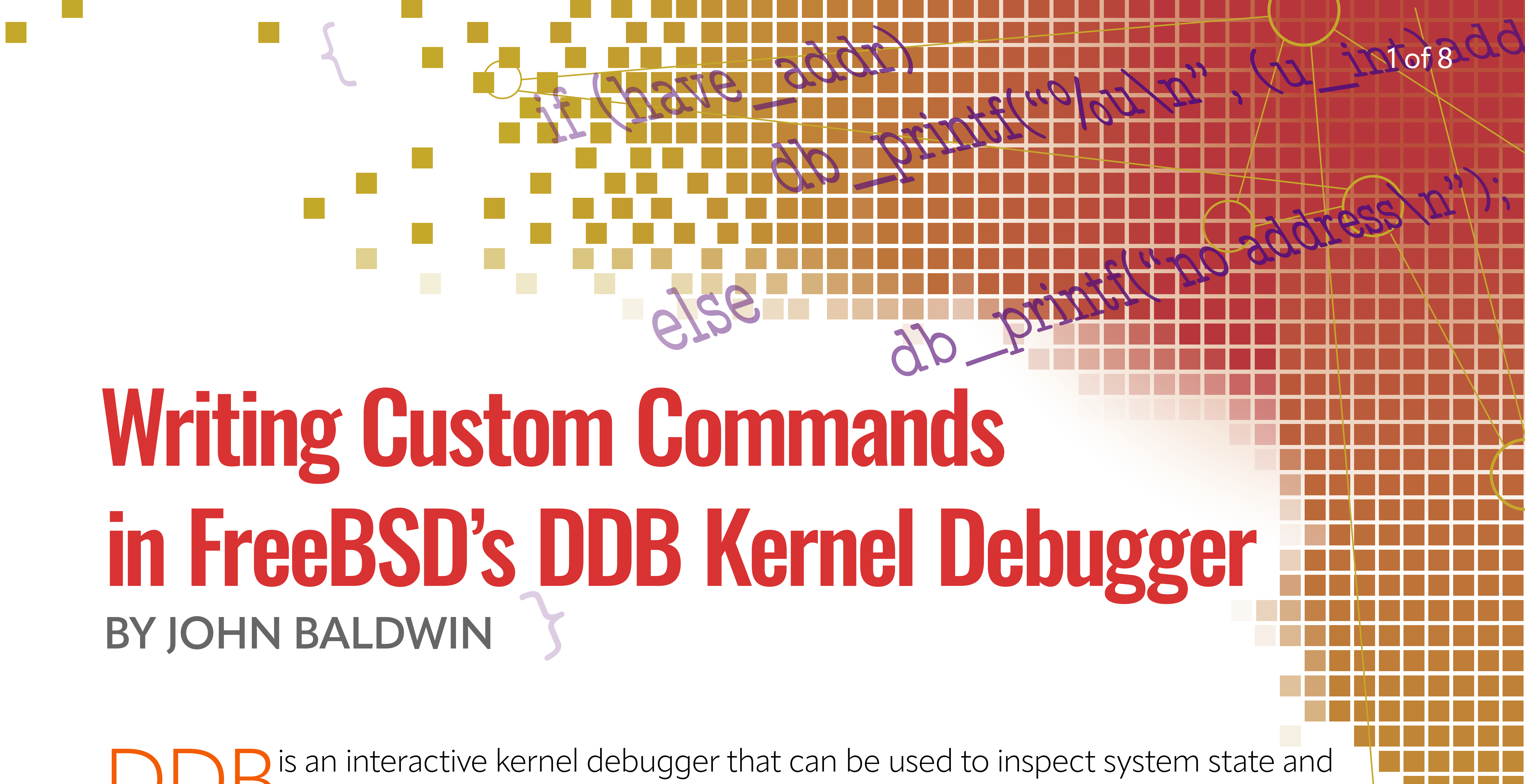
Finally, I want to thank you for enthusiastically reading each issue! We hope you've enjoyed *FreeBSD Journal's* articles and columns this past year and we look forward to continuing it for years to come.

We wish you a wonderful year-end, however you celebrate this holiday season.

Deb Goodkin
Executive Director
FreeBSD Foundation

FreeBSD® JOURNAL
November/December 2022

Observability and Metrics



Writing Custom Commands in FreeBSD's DDB Kernel Debugger

BY JOHN BALDWIN

DDB is an interactive kernel debugger that can be used to inspect system state and control the running kernel. DDB was first developed as part of the Mach operating system. It was later ported to 386BSD from which it was inherited by various operating systems including FreeBSD, NetBSD, and OpenBSD. This article focuses on the implementation of DDB in FreeBSD.

DDB runs on the system console, and system execution is suspended while the debugger is active. This permits inspection of the system in a consistent state. DDB can be entered manually, but DDB is typically used after a kernel panic. FreeBSD's kernel can be configured to enter DDB after a kernel panic permitting a user or system administrator to examine the system state before rebooting. Debugging kernels built from FreeBSD's main branch do this by default.

DDB provides many features common to debuggers. It supports run control such as single stepping and breakpoints, and also supports hardware watchpoints on platforms with suitable hardware support. DDB includes several commands to display information about a system including stack traces and memory dumps.

Unlike many other debuggers, DDB does not understand type information and is not able to pretty-print structures or evaluate members of structures or unions in expressions. However, DDB can be extended by defining new commands. New commands can even be implemented in kernel modules which can be loaded after boot.

DDB Execution Context

DDB executes in a special context which differs from the normal kernel execution context in several ways:

- When DDB is active, the system is paused and borrows the execution context of the currently executing thread on each CPU. The normal kernel scheduler does not function in this context and the borrowed threads on each CPU are not permitted to context switch. This means that code execution in this context must not sleep or block on locks.
- If a fault or trap occurs during execution of a DDB command, the current thread will use `longjmp()` to resume execution in DDB's main loop.
- DDB accesses console devices directly for console input and output.

Due to these unique behaviors, implementations of DDB commands should adhere to the following guidelines:

- Commands should avoid side effects. If a fault occurs during command execution, there is no way to undo any side effects. The safest approach is to avoid side effects when possible.
- Commands should not use locks. Since execution is paused on all CPUs, the state of most data structures in the system will not be changing, so locks are not needed to synchronize with other CPUs. In addition, acquiring a lock is a side effect that will not be unwound if a command faults while holding a lock. In exceptional cases, where a command wishes to modify system state in a safe way, a command may use try locks. One command which does this currently is the kill command which can send a signal to a process.
- Commands should avoid complicated APIs. Higher level APIs often modify system state or contain other side effects such as acquiring locks.
- Most commands in DDB inspect system state without modifying it and output a human readable description of some portion of system state. Many of these commands are pretty-printers which print information about a specific data structure or list of structures.
- Commands must use DDB's APIs for console input and output. Mostly this means using **`db_printf()`** for output instead of **`printf()`**.

DDB provides a simple API for console output. The **`db_printf()`** function is similar to the normal kernel **`printf()`** and supports all of the same format specifiers. This function writes directly to console devices bypassing the system log device.

In addition, **`db_printf()`** includes simple pager support. Each time a newline is output to the console, **`db_printf()`** checks if the output should be paused. If so, **`db_printf()`** outputs a prompt on the console permitting the user to control how many lines are displayed before the next pause. Once the user has responded to the prompt, **`db_printf()`** returns. If the user requests the current command to quit (stop generating output), **`db_printf()`** sets the global variable **`db_pager_quit`** to a non-zero value. If a command generates output in a loop (for example, using a loop to walk a linked-list of data structures), the command should check **`db_pager_quit`** in each loop iteration and break out of the loop early if it is set.

Command Functions

Most DDB commands follow a simple syntax described in [ddb(4)](ddb(4)):

```
command[/modifier] [address[,count]]
```

When a user enters a command at DDB's prompt, DDB parses this command line. The address and count fields are treated as expressions which can contain references to named symbols and many C arithmetic operators. The command and modifier fields are treated as simple strings. DDB uses the command field to locate a pointer to a C function. This C function is invoked to execute the command.

The functions implementing DDB commands use the following signature:

```
void fn(db_expr_t addr, bool have_addr, db_expr_t count, char *modif)
```

The **addr** argument contains the address the command should operate on. This can either be an explicitly-provided address or the address used with the previous command. The **have_addr** argument is true if the address was provided explicitly. The count argument contains the value of the **count** field. If the count field was not specified, **count** is set to -1. The **modif** argument is a pointer to a C string containing the modifier field. If a modifier was not specified, then **modif** will point to an empty string.

Command functions are associated with command names via internal tables maintained by DDB. DDB provides helper macros to abstract most of the details of registering new commands. Each macro accepts two arguments: the first argument is the name of the command, and the second argument is the name of the C function to associate with the named command. In addition to registering the linkage in the table, these macros also provide the C function declaration and should be immediately followed by the function body. Each macro is associated with a specific command table. The DB_COMMAND macro defines a new top-level command. The DB_SHOW_COMMAND macro defines a new command in the "show" table. The DB_SHOW_ALL_COMMAND macro defines a new command in the "show all" table. For example **DB_SHOW_COMMAND(bar, db_show_bar_func)** defines a new "show bar" command. It also defines a new C function, **db_show_bar_func**, which provides the implementation of this command. It is best practice, but not required, to name the C functions associated with a command using the pattern **db_<command>_cmd**.

Listing 1 is the source to a simple command named "double". This command multiplies the address provided by the user by 2 and outputs the result. Listing 2 shows some use cases of this command. The output from the third case may be surprising, as 32 times 2 is certainly not 100. The reason for this behavior is that DDB parses integer values with a default base of 16 (controlled by DDB's internal **$radix** variable). In base 16, 32 evaluates to the decimal value of 50.

```
DB_COMMAND(double, db_double_cmd)
{
      if (have_addr)
            db_printf("%u\n", (u_int)addr * 2);
      else
            db_printf("no address\n");
}
```

Listing 1: Source for the "**double**" command

```
db> double
no address
db> double 4
8
db> double 32
100
```

Listing 2: Sample output for the "**double**" command

Commands with Custom Syntax

DDB commands do not have to use the simple syntax given above. Command functions can choose to support other syntaxes. Commands request this by passing an additional flag when registering commands. A separate set of macros accept command flags as a third argument: DB_COMMAND_FLAGS, DB_SHOW_COMMAND_FLAGS, and DB_SHOW_ALL_COMMAND_FLAGS.

Two flags are available to control command line parsing. CS_MORE indicates that a command mostly follows the simple syntax, but that the command supports more than one address. When this flag is specified, the main loop of DDB will still parse the command line as normal, but it will not discard any remaining tokens from its lexer before invoking the command function. This allows the command function to parse additional options on the command line. The second flag, CS_OWN, indicates that the command function performs all of the parsing itself. When this flag is specified, the main loop of DDB stops parsing the command line after reading the name of the command. The command function uses DDB's lexer to parse the rest of the command line. Regardless of which flag is specified, the command function must call **db_skip_to_eol()** to discard remaining tokens from the current command line before returning.

DDB provides a few functions to parse command line arguments. **db_expression()** parses an arithmetic expression. This can consume multiple words of input and supports the full DDB expression syntax including symbol resolution and various C operators. If no more command line arguments were available, **db_expression()** returns 0. If an expression was successfully parsed, then **db_expression()** returns a non-zero value and stores the result of the expression in the value pointed to by its sole argument. If **db_expression()** encounters a syntax error while parsing an expression, it prints a message and aborts the current command via **longjmp()**. Command functions should avoid any side effects while calling **db_expression()** since they can't be unwound if the user provides invalid input.

Two other functions provide a lower level interface to DDB's lexer. **db_read_token()** parses the next token from the command line and returns a constant identifying the type of token parsed. The constants are named **t<TYPE>** and are defined in **<ddb/db_lex.h>**. Most of the constants are associated with C operators and other special tokens, but a few are useful for custom commands. **tEOL** is returned when the end of the command line is encountered. **tEOF** is returned for invalid input such as a number that contains invalid characters. **tIDENT** is returned when a word (identifier) is parsed. A copy of the word is saved in the global variable **db_tok_string**. **tNUMBER** is returned when a numeric value is parsed. The value is saved as an integer in the global variable **db_tok_number**. Note that DDB's lexer assumes that any word beginning with a decimal digit is a number, and that any word beginning with an alphabetic character, underscore, or backslash is an identifier. **db_unread_token()** inserts a single token to be returned by the next call to **db_read_token()**. The value passed to **db_unread_token()** is one of the **t<TYPE>** constants. Normally this function is used to put back the token just read from **db_read_token()** if the returned token was invalid or unexpected.

DDB provides two additional functions to handle parsing errors. **db_error()** prints out a caller-supplied message, flushes the lexer state, and invokes **longjmp()** to abort the current command and return to DDB's main loop. **db_flush_lex()** just flushes the lexer state discarding the current command line. **db_flush_lex()** can be used if a more detailed error message is desired or to unwind additional state if **longjmp()** is undesirable.

Listing 3 is the source to a command named "**sum**". This command computes a sum of all of the expressions given on the command line. It uses the CS_MORE flag and uses **db_expression()** in a loop to parse additional expressions from the command line. Listing 4 shows some sample output from this command. Note that in the third case, `db_expression()` parsed the expression "`9 * 3`" and returned the value 27 to the loop in `db_sum_cmd()`.

```
DB_COMMAND_FLAGS(sum, db_sum_cmd, CS_MORE)
{
        long total;
        db_expr_t value;

        if (!have_addr)
                db_error("no values to sum\n");

        total = addr;
        while (db_expression(&value))
                total += value;
        db_skip_to_eol();
        db_printf("Total is %lu\n", total);
}
```

Listing 3: Source for the "`sum`" command

```
db> sum 1
Total is 1
db> sum 1 2 3
Total is 6
db> sum 9 * 3 4
Total is 31
```

Listing 4: Sample output for the "`sum`" command

Listing 5 contains the source to a "`show softc`" command. This command accepts the name of a device as a single command line argument. If the device is found, the command prints out the value of the pointer to the device's `softc` structure. This structure contains the per-device information maintained by the device's driver. This command uses the `CS_OWN` flag to request full control of command line parsing. It uses `db_read_token()` to fetch the device name from the command line. If a valid device name is given, a `tIDENT` token will be returned with the device name saved in `db_tok_string`. Listing 6 shows some sample output for this command.

```
DB_SHOW_COMMAND_FLAGS(softc, db_show_softc_cmd, CS_OWN)
{
        device_t dev;
        int token;

        token = db_read_token();
        if (token != tIDENT)
                db_error("Missing or invalid device name");
        dev = device_lookup_by_name(db_tok_string);
        db_skip_to_eol();
        if (dev == NULL)
                db_error("device not found\n");
        db_printf("%p\n", device_get_softc(dev));
}
```

Listing 5: Source for the "`show softc`" command

```
db> show softc 4
Missing or invalid device name
db> show softc foo0
device not found
db> show softc pci0
0xfffff800039380f0
```

Listing 6: Sample output for the **"`show softc`"** command

Custom Command Tables

A DDB command table contains a list of commands. Additional tables can be defined by a special command in an existing table. This permits building a tree of command tables. Commands that define new tables do not specify a function to use as their command handler. Instead, tables must define and initialize a variable of type struct **`db_command_table`** which will contain a linked-list of commands belonging to the table. A pointer to this table is associated with the command entry in the parent table. This variable should be named using the pattern **`db_<name>_table`**. At the time of writing, there are not nicely abstracted macros similar to `DB_COMMAND` which permit defining new tables. Instead, new tables must be defined using an "internal" macro `_DB_SET`. Commands belonging to this table must either be defined by the "internal" macro `_DB_FUNC` or by defining a new helper macro similar to `DB_SHOW_COMMAND` which wraps `_DB_FUNC`.

Listing 7 contains the source for a "demo" table along with two commands belonging to this table. The listing starts by defining a **`db_demo_`**table variable to contain the list of DDB commands belonging to the new table. The `_DB_SET` invocation adds the "demo" command to the top-level table similar to `DB_COMMAND`. Note that the third argument to `_DB_SET` (which normally contains a pointer to the function handler) is `NULL`, but that the last argument to `_DB_SET` contains a pointer to the new table. The rest of the listing defines two simple commands belonging to this new table. The second and third arguments to `_DB_FUNC` are similar to the two arguments given to `DB_COMMAND`. The fourth argument identifies the parent table the new command belongs to. The fifth argument contains flags such as `CS_MORE` or `CS_OWN`, and the final argument should be `NULL`. The first argument to both `_DB_SET` and `_DB_FUNC` should be the name of the parent table with a leading underscore and any spaces replaced by underscores. If the parent table is the main table, use "`_cmd`". Listing 8 shows sample output for these commands.

```
/* Holds list of "demo *" commands. */
static struct db_command_table db_demo_table = LIST_HEAD_INITIALIZER(db_demo_table);
```

```
/* Defines a "demo" top-level command. */
_DB_SET(_cmd, demo, NULL, db_cmd_table, 0, &db_demo_table);

_DB_FUNC(_demo, one, db_demo_one_cmd, db_demo_table, 0, NULL)
{
      db_printf("one\n");
}

_DB_FUNC(_demo, two, db_demo_two_cmd, db_demo_table, 0, NULL)
{
      db_printf("two\n");
}
```

Listing 7: Source for the "**`demo`**" table commands

```
db> demo
Subcommand required; available subcommands:
one                two
db> demo one
one
db> demo two
two
```

Listing 8: Sample output for the "**`demo`**" table commands

Pager-Aware Command

Our last sample command provides an example of honoring DDB's output pager. Most pager operations such as continuing for a page or for a single line are handled internally by the pager implementation in **`db_printf()`**. However, if the user requests that the pager stop, the global variable **`db_pager_quit`** is set to a non-zero value as noted earlier. Commands which generate output in a loop should check this variable and abort any loops if it is set. Listing 9 contains an abbreviated sample command which checks **`db_pager_quit`**. The command is an implementation of the Internet "**`chargen`**" service. It generates lines of output to the screen in a continuous loop until the user terminates the loop by requesting an exit via the pager. The main takeaway from this listing are the last two lines of the main loop which break out of the loop if **`db_pager_quit`** is set.

```
DB_COMMAND(chargen, db_chargen_cmd)
{
      char *rs;
      int len;

      for (rs = ring;;) {
            …
            db_printf("\n");
            if (db_pager_quit)
```

```
                Break;
        }
}
```

Listing 8: Abbreviated source for the "`chargen`" command

Conclusion

DDB provides a fairly simple framework for adding new commands. New commands can even be added post-boot by loading kernel modules containing new commands. There are many examples of custom commands in FreeBSD's source tree which can also be used as a reference when developing new commands. These can be found by searching for `DB.*_COMMAND` or `db_printf`. In addition, a kernel module containing all of the commands from this article can be found at https://github.com/bsdjhb/ddb_commands_demo.

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

DTrace: New Additions to an Old Tracing System

BY DOMAGOJ STOLFA

DTrace is a software tracing framework built into FreeBSD that allows users to inspect and modify the currently running system in real time. It is highly extensible and was originally built for Solaris, but has since been ported many times to other environments such as FreeBSD, macOS, Windows and Linux. This article will focus on DTrace usage in FreeBSD with examples and give a summary of recent developments in the DTrace space on FreeBSD.

DTrace in Short

Operating systems are very complicated pieces of software which have many components. A single tracing system attempting to support tracing of nearly the entire OS can be overwhelming given their complexity. In order to simplify this as well as to account for future extensions, DTrace introduces the notion of a *provider*. Providers live in the kernel as kernel modules by default in FreeBSD and are responsible for implementing the necessary functionality to instrument a particular component of the OS. They expose DTrace *probes* which are names for locations in the OS code that can be dynamically instrumented with scriptable routines written in the D programming language. Some example providers shipped with FreeBSD include the function boundary tracing provider (`fbt.ko`) — responsible for instrumentation of kernel function entry and exit points, the profile provider (`profile.ko`) which provides probes associated with a fixed time-based interrupt specified by the script-writer, the PID provider (`fasttrap.ko`) which implements `fbt` but for user processes and the libraries they link to and various others. While deep knowledge of DTrace is not required in order to understand this article, those wishing to know more about DTrace may want to check out the user guide[1], specification[2], FreeBSD Handbook Page[3], whitepaper[4], book[5] and various FreeBSD wiki pages that can be found such as the list of one-liners[6]. Furthermore, a number of previous FreeBSD Journal editions featured articles on DTrace[7,8,9].

Simple Examples

Probes are specified via a `provider:module:function:name` 4-tuple. Each of the entries can be globbed or left blank to mean "everything". We use an example toy snooper script as an introduction to D. The script tells us which programs users are running. Note that we specify the `-x quiet` option to avoid additional information that DTrace would otherwise output.

```
# dtrace -x quiet -n 'proc:::exec { printf("user = %u, gid = %u: %s\n", uid, gid,
stringof(args[0])); }'
user = 1001, gid = 1001: /usr/sbin/service
user = 1001, gid = 1001: /bin/kenv
user = 1001, gid = 1001: /sbin/sysctl
user = 1001, gid = 1001: /sbin/env
user = 1001, gid = 1001: /bin/env
user = 1001, gid = 1001: /usr/sbin/env
user = 1001, gid = 1001: /usr/bin/env
user = 1001, gid = 1001: /etc/rc.d/sendmail
user = 1001, gid = 1001: /bin/kenv
user = 1001, gid = 1001: /sbin/sysctl
user = 1001, gid = 1001: /bin/ls
```

As we can see, D is very similar to C in its syntax aside from a couple of special forms of syntax specific to it. Unlike C, it does not support loops so any form of looping must be done by manually unwinding the loop. In the above example we can access the user and group id through **uid** and **gid** built-in variables.

DTrace also supports aggregating the trace results together in various ways. For example, we can count up the system calls each program is doing:

```
# dtrace -n 'syscall:::entry { @syscall_agg[execname, pid] = count(); }'
dtrace: description 'syscall:::entry ' matched 1148 probes
 sh                                                      46569                 7
 sh                                                      46570                 7
 syslogd                                                   703                16
 sshd                                                      848                17
 devd                                                      501                20
 ntpd                                                      771                24
 sh                                                      46565                93
 dtrace                                                  46568               138
 ps                                                      46570               254
 sshd                                                    46564             27517
 ls                                                      46569             35755
```

Using **@** as a prefix to a variable makes it an aggregate variable. **@syscall_agg** is indexed by two keys, however one can keep adding keys. The aggregation output for **@syscall_agg** should be read as:

```
execname                                                pid                count
```

Our final example will be one with stack traces. DTrace allows the user to gather stack traces both in the kernel and userspace using **stack()** and **ustack()** routines respectively. Furthermore, DTrace can be extended with language-specific stack unwinders. One such example is the **jstack()** action, which provides the user a legible backtrace from a Java program. In our example, we focus on **stack()**:

```
# dtrace -x quiet -n 'io:::start { @[stack()] = count(); }'
              zfs.ko`zio_vdev_io_start+0x2f5
              zfs.ko`zio_nowait+0x15f
              zfs.ko`vdev_mirror_io_start+0xfd
              zfs.ko`zio_vdev_io_start+0x1eb
              zfs.ko`zio_nowait+0x15f
              zfs.ko`arc_read+0x14aa
              zfs.ko`dbuf_read+0xc84
              zfs.ko`dmu_tx_check_ioerr+0x84
              zfs.ko`dmu_tx_count_write+0x191
              zfs.ko`dmu_tx_hold_write_by_dnode+0x64
              zfs.ko`zfs_write+0x500
              zfs.ko`zfs_freebsd_write+0x39
              kernel`VOP_WRITE_APV+0x194
              kernel`vn_write+0x2ce
              kernel`vn_io_fault_doio+0x43
              kernel`vn_io_fault1+0x163
              kernel`vn_io_fault+0x1cc
              kernel`dofilewrite+0x81
              kernel`sys_writev+0x6e
              kernel`amd64_syscall+0x12e
                1

              zfs.ko`zio_vdev_io_start+0x2f5
              zfs.ko`zio_nowait+0x15f
              zfs.ko`zil_lwb_write_done+0x360
              zfs.ko`zio_done+0x10d6
              zfs.ko`zio_execute+0xdf
              kernel`taskqueue_run_locked+0xaa
              kernel`taskqueue_thread_loop+0xc2
              kernel`fork_exit+0x80
              kernel`0xffffffff810a35ae
                1
```

This D script counts up all of the kernel stack traces that lead to I/O on a block device. We omit the aggregation name as we only have one aggregation in this script and our key is `stack()` — a built-in DTrace action returning an array of program counters which are later resolved to symbols when printing results. DTrace can also gather stacks using the `profile` provider in order to gather on-CPU stack traces, making it possible to generate Flame Graphs[10].

New Developments

dwatch

A new tool called `dwatch` was developed by Devin Teske (dteske@freebsd.org) and upstreamed to FreeBSD 11.2. `dwatch` makes DTrace much easier to use for common use-cases than the **dtrace** command line tool. Going back to our toy snooping example, one can simply run:

```
# dwatch execve
```

to get nicely filtered output with more information than our simple snooper shown above.

```
# dwatch execve
INFO Watching 'syscall:freebsd:execve:entry' ...
2022 Nov 24 18:46:53 1001.1001 sh[46565]: sudo ps auxw
2022 Nov 24 18:46:53 0.0 sudo[46920]: ps auxw
2022 Nov 24 18:46:55 1001.1001 sh[46565]: ls
2022 Nov 24 18:47:01 1001.1001 sh[46565]: ls -lapbtr
2022 Nov 24 18:47:09 1001.1001 sh[46924]: kenv -q rc.debug
2022 Nov 24 18:47:09 1001.1001 sh[46924]: /sbin/sysctl -n -q kern.boottrace.enabled
2022 Nov 24 18:47:09 1001.1001 sh[46565]: env -i -L -/daemon HOME=/ PATH=/sbin:/
bin:/usr/sbin:/usr/bin /etc/rc.d/sendmail onestop
2022 Nov 24 18:47:09 1001.1001 env[46565]: /bin/sh /etc/rc.d/sendmail onestop
2022 Nov 24 18:47:09 1001.1001 sh[46924]: kenv -q rc.debug
2022 Nov 24 18:47:09 1001.1001 sh[46924]: /sbin/sysctl -n -q kern.boottrace.enabled
```

Furthermore, **dwatch** supports filtering based on jails, groups, processes and many other features that make it worthwhile to learn for even the most seasoned DTrace users. All along the dwatch tower[11] is an excellent talk that introduces **dwatch** and goes over its features in detail. Similarly, the **dwatch(1)** man page in FreeBSD has a lot of good examples for those interested to try out.

CTFv3

Compact C Type Format (CTF) is a format used to encode C type information in FreeBSD ELF binaries. It allows DTrace to know C type layouts for target binaries (processes, the kernel) so that scripts written by users can refer to those types and explore them. In the past DTrace only supported a total of **2^15** C types in a single binary encoded as CTF due to the way that CTFv2 was implemented. This limitation was a source of many bug reports in FreeBSD relating to DTrace. In March of this year, Mark Johnston (markj@freebsd.org) committed changes which switches DTrace to use CTFv3 instead which raises not only the number of C types that can be manipulated by DTrace, but also various other limits in CTF.

dwatch supports filtering based on jails, groups, processes and many other features that make it worthwhile to learn.

kinst – A New DTrace Provider for Instruction-level Tracing

A 2022 Google Summer of Code project successfully completed by Christos Margiolis (christos@freebsd.org) and mentored by Mark Johnston (markj@freebsd.org) implemented and upstreamed instruction-level tracing to FreeBSD. The provider that implements this functionality is called **kinst**. It reuses parts of the **fbt** mechanism and extends it to instrument arbitrary points of a kernel function, rather than just the entry and exit points.

Kernel developers reading this might already see the potential of **kinst** when it comes to analyzing call stacks from certain branches in a function. As a result finding bugs and performance issues in FreeBSD could be made easier and faster. For a demonstration, we consider scenarios resembling the following C-style pseudo-code:

```
if (__predict_false(rarely_true)) {
    return (slow_operation());
} else {
    return (get_from_cache());
}
```

In this example, we focus on a particular function in the FreeBSD kernel that has behavior similar to this. The simplified and stripped down version of it is:

```
void
_thread_lock(struct thread *td)
{
        ...
        if (__predict_false(LOCKSTAT_PROFILE_ENABLED(spin__acquire)))
                goto slowpath_noirq;
        spinlock_enter();
        ...
        if (__predict_false(m == &blocked_lock))
                goto slowpath_unlocked;
        if (__predict_false(!_mtx_obtain_lock(m, tid)))
                goto slowpath_unlocked;
        ...
        _mtx_release_lock_quick(m);
slowpath_unlocked:
        spinlock_exit();
slowpath_noirq:
        thread_lock_flags_(td, 0, 0, 0);
}
```

It's immediately noticeable that there are two slow paths: **slowpath_unlocked** and **slowpath_noirq**. In the two slow paths, either **spinlock_exit()** or **thread_lock_flags_()** is called, whereas **_mtx_release_lock_quick()** is just an atomic compare-and-swap instruction on amd64. In order to use **kinst** to identify the call stacks which end up in the slow paths, we first need to disassemble the function in some way. One possible way of doing so is using **kgdb** in FreeBSD (**pkg install gdb**):

```
# kgdb
(kgdb) disas /r _thread_lock
Dump of assembler code for function _thread_lock:
```

```
...
0xffffffff80bc7dcc <+124>:   5d       pop    %rbp
  0xffffffff80bc7dcd <+125>:   e9 4e 72 09 00   jmp    0xffffffff80c5f020
<witness_lock>
  0xffffffff80bc7dd2 <+130>:   48 c7 43 18 00 00 00 00 movq   $0x0,0x18(%rbx)
  0xffffffff80bc7dda <+138>:   e8 e1 43 4e 00   call   0xffffffff810ac1c0
<spinlock_exit>
  0xffffffff80bc7ddf <+143>:   8b 75 d4         mov    -0x2c(%rbp),%esi
...
  0xffffffff80bc7df2 <+162>:   41 5d    pop    %r13
  0xffffffff80bc7df4 <+164>:   41 5e    pop    %r14
  0xffffffff80bc7df6 <+166>:   41 5f    pop    %r15
  0xffffffff80bc7df8 <+168>:   5d       pop    %rbp
  0xffffffff80bc7df9 <+169>:   e9 82 00 00 00   jmp    0xffffffff80bc7e80
<thread_lock_flags_>
```

In this case, we can take the instructions at offset +138 and +169, which are the function calls to **`spinlock_exit()`** and **`thread_lock_flags_()`**. Using those offsets, we can now write our DTrace script:

```
# dtrace -n 'kinst::_thread_lock:138,kinst::_thread_lock:169 { @[stack(),
probename] = count(); }'
...
             0xcf566bb0
             kernel`ipi_bitmap_handler+0x87
             kernel`0xffffffff810a48b3
             kernel`vm_fault_trap+0x71
             kernel`trap_pfault+0x22d
             kernel`trap+0x48c
             kernel`0xffffffff810a2548
 138                                                                  8
```

Those familiar with DTrace might notice that this could have easily been implemented using speculative tracing instead of needing to use **`kinst`**. However, one can easily imagine scenarios where the "slow path" or its equivalent is not a simple function call or where the same function call might be present in all of the branches.

`kinst` also has other implications on the DTrace ecosystem on FreeBSD. Historically, there has been a problem with instrumentation of inlined functions in the kernel using **`fbt`**. The mechanisms used to implement **`kinst`** could help extend **`fbt`** in order to support reliable tracing of inlined functions.

Ongoing work

DTrace and eBPF – a Comparison

Mateusz Piotrowski (0mp@FreeBSD.org) has been working on the performance analysis of DTrace on FreeBSD and how it compares to eBPF on Linux. Some of the results were presented[12] this year at EuroBSDcon 2022. This work could lead to interesting results which could serve as a basis for further optimization of DTrace. This would make enabling instrumentation on performance-critical systems less disruptive.

HyperTrace

HyperTrace is a framework built on top of DTrace which allows the user to apply DTrace-like tracing techniques using the D programming language to tracing virtual machines. It grew out of the CADETS project at the University of Cambridge in the UK. As a simple example, we look at our original snooper script and extend it to use HyperTrace:

```
# dtrace -x quiet -En 'FreeBSD-14*:proc:::exec { printf("%s: user = %u, gid = %u:
%s\n", vmname, uid, gid, stringof(args[0])); }'
scylla1-webserver-0: user = 0, gid = 0: /usr/sbin/dtrace
scylla1-webserver-0: user = 0, gid = 0: /sbin/ls
scylla1-webserver-0: user = 0, gid = 0: /bin/ls
scylla1-client-0: user = 0, gid = 0: /usr/sbin/sshd
scylla1-client-0: user = 0, gid = 0: /bin/csh
scylla1-client-0: user = 0, gid = 0: /usr/bin/resizewin
scylla1-client-0: user = 0, gid = 0: /usr/sbin/iperf
scylla1-client-0: user = 0, gid = 0: /usr/bin/iperf
scylla1-client-0: user = 0, gid = 0: /usr/local/bin/iperf
host: user = 0, gid = 0: /bin/sh
host: user = 0, gid = 0: /usr/libexec/atrun
scylla1-client-0: user = 0, gid = 0: /bin/sh
scylla1-client-0: user = 0, gid = 0: /usr/libexec/atrun
scylla1-client-1: user = 0, gid = 0: /bin/sh
scylla1-client-1: user = 0, gid = 0: /usr/libexec/atrun
scylla1-client-2: user = 0, gid = 0: /bin/sh
scylla1-client-2: user = 0, gid = 0: /usr/libexec/atrun
scylla1-client-3: user = 0, gid = 0: /bin/sh
scylla1-client-3: user = 0, gid = 0: /usr/libexec/atrun
scylla1-webserver-0: user = 0, gid = 0: /bin/sh
scylla1-webserver-0: user = 0, gid = 0: /usr/libexec/atrun
```

We modified the script in order two new things: the prefix of where each of the processes was executed using the built-in variable **`vmname`** and a 5th tuple entry in the probe specification: **`FreeBSD-14*`**. This allows the user to specify which target machines (VMs) to instrument and can be controlled through command-line flags to support name resolution via things like the OS version or the machine's hostname.

Similar changes can be made to our block I/O example:

```
# dtrace -x quiet -En 'scylla1-*:io:::start { @[vmname, immstack()] = count(); }'
...
 scylla1-webserver-0
             devstat_start_transacti+0x90
             g_disk_start+0x316
             g_io_request+0x2d7
             g_part_start+0x289
```

```
              g_io_request+0x2d7
              g_io_request+0x2d7
              ufs_strategy+0x83
              VOP_STRATEGY_APV+0xd2
              bufstrategy+0x3e
              bufwriteL+0x80
              vfs_bio_awrite+0x24f
              flushbufqueues+0x52a
              buf_daemon+0x1f1
              fork_exit+0x80
              fork_trampoline+0xe
              aio_process_rw+0x10c
              aio_daemon+0x285
              fork_exit+0x80
              fork_trampoline+0xe
              fork_trampoline+0xe
            10598
scylla1-client-0
              g_disk_start+0x316
              g_io_request+0x2d7
              g_part_start+0x289
              g_io_request+0x2d7
              g_io_request+0x2d7
              ufs_strategy+0x83
              VOP_STRATEGY_APV+0x9e
              bufstrategy+0x3e
              bufwriteL+0x3e
              vfs_bio_awrite+0x24f
              flushbufqueues+0x52a
              buf_daemon+0x1f1
              fork_exit+0x80
              fork_trampoline+0xe
              fork_trampoline+0xe
              aio_process_rw+0x10c
              aio_daemon+0x285
              fork_exit+0x80
              fork_trampoline+0xe
              fork_trampoline+0xe
            10605
```

Here a new DTrace action **`immstack()`** is used which works similar to **`stack()`** but symbol resolution happens in the kernel rather than during time of printing output.

HyperTrace works by aiming to execute the entire D script on the host kernel rather than running DTrace inside the guest, while each of the guests is responsible for instrumenting itself and issuing a synchronous hypercall (akin to a system call in an OS) to the host when the probe is executed on the guest. This kind of design enables keeping global state across all of the guests and host in one place — increasing the overall expressiveness of D when it comes to tracing VMs. The work is still in progress and can be viewed on GitHub[13].

Further Reading

1. https://illumos.org/books/dtrace/preface.html#preface
2. https://github.com/opendtrace
3. https://docs.freebsd.org/en/books/handbook/dtrace/
4. https://www.cs.princeton.edu/courses/archive/fall05/cos518/papers/dtrace.pdf
5. https://www.brendangregg.com/dtracebook/
6. https://wiki.freebsd.org/DTrace/One-Liners
7. https://freebsdfoundation.org/wp-content/uploads/2014/05/DTrace.pdf
8. https://issue.freebsdfoundation.org/publication/?m=29305&i=417423&p=14&ver=html5
9. http://www.onlinedigeditions.com/publication/?m=29305&i=536657&p=4&ver=html5
10. https://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html
11. https://papers.freebsd.org/2018/bsdcan/teske-all_along_the_dwatch_tower/
12. https://github.com/freebsd/freebsd-papers/pull/112
13. https://github.com/cadets/freebsd

DOMAGOJ STOLFA is a Research Assistant at the University of Cambridge working on dynamic tracing of virtualized systems. He has been working with bhyve and DTrace on FreeBSD and contributing patches since 2016. Domagoj is also a teaching assistant on the Advanced Operating Systems courses at the University of Cambridge, teaching operating systems concepts with FreeBSD using DTrace and PMCs.

Certificate-based Monitoring with Icinga

BY BENEDICT REUSCHLING

Icinga is the successor to the popular Nagios host and service-monitoring software. With the aim of being a drop-in replacement with backwards compatibility to Nagios, Icinga also provides a couple of new features. This article describes how to set up a central Icinga host to monitor endpoint systems using certificates in a top-down configuration sync fashion.

Top-down describes the way that the checks (disk full, load-avg too high, etc.) execute on the remote machines. In top-down monitoring, the central Icinga host called parent is responsible for synchronizing the configuration files to the monitored nodes called child nodes. No manual restart after a configuration change is required on these child nodes, as syncing, validation, and restarts happen automatically. Checks execute directly on the child node scheduler in regular intervals. Hosts are organized in a global zone and each host (parent and children) is defined as an endpoint in it. Each child will have to verify that it is part of the monitoring zone by creating a certificate signing request in a ticket which the parent will then verify and sign. This creates trust that the machines communicate in an encrypted way and that the monitoring data received on the parent is not manipulated in transit.

This article describes how to set up a central Icinga host to monitor endpoint systems using certificates.

The setup described here uses FreeBSD as parent to monitor another FreeBSD host as child. Of course, other operating systems such as Linux or Windows are possible in the same way, but not covered as part of this already long description. Although the setup is described to run on the host itself to reduce complexity, the author has it happily running in a jail—so far without issues.

Preparation

We assume that all our hosts are installed, can find each other on the same network, and have a basic SSH connection going on between them. Our central monitoring server will be called `monitor.example.org` and the client we're going to set up later is called `client.example.org` (you see my creative naming choices right there). The prompts used in front of the commands will indicate on which host this command is to be executed. Let's start with preparing our central monitoring host (the parent).

Ensure that clocks on the machines are synchronized. This is important for proper generation of certificates later. This is typically done using `monitor# ntpdate -b pool.ntp.org`.

On this central FreeBSD host, we want to use the latest version of Icinga and other packages, instead of the quarterly ones. Edit **/etc/pkg/FreeBSD.conf** and change the word quarterly to latest in the line starting with url:. Save and exit afterwards. To update the package repository with the newer packages, run:

```
monitor# pkg update
```

PostgreSQL Setup

Before installing the required software packages (including PostgreSQL as the backend database and nginx for the webserver to host the Icingaweb2 monitoring interface), we're creating a ZFS dataset for the postgres database first to get populated when the packages extract. If you don't run ZFS, this also works fine with regular directories.

The following commands create a new dataset on our example pool called **mypool** a **/var/db/postgres/data**. Non-existent datasets on that path are also created using the **-p** parameter. Next, access time is deactivated as we don't need it here and it saves some I/O by not updating timestamps of files at every write. With newer ZFS 2.0, we also use the zstd compression on the dataset. As Postgres writes data in chunks of 8k, we set the ZFS **recordsize** to match for best performance. With **logbias** set to throughput, we instruct ZFS to optimize synchronous writes from the database for efficient resource use. The mountpoint is set to overlap the existing **/var/db/postgres** path. When the package gets installed, it is put on that dataset instead of the regular **/var/db** directory. Note: we don't focus on further tuning of the PostgreSQL database here. Go to https://pgtune.leopard.in.ua/ and enter the values of your PostgreSQL host to get configuration recommendations to put into the **postgresql.conf** file.

```
monitor# zfs create -p mypool/var/db/postgres/data
monitor# zfs set atime=off mypool/var/db
monitor# zfs set compression=zstd mypool/var/db
monitor# zfs set recordsize=8k mypool/var/db/postgres
monitor# zfs set logbias=throughput mypool/var/db/postgres
monitor# zfs set mountpoint=/var/db/postgres mypool/var/db/postgres
```

Now it is time to install the required packages. Easily done using pkg install, including automatic dependency resolution:

```
monitor# pkg install icinga2 icingaweb2-php74 postgresql13-server nginx \
ImageMagick7-nox11 php74-pecl-imagick-im7
```

Note: at the time of this writing, these package versions were the latest available. Check to make sure that is still the case by going to www.freshports.org to see if there are newer package versions listed. PHP, in particular, may have gotten a version bump in the meantime.

Some of these services need entries in **/etc/rc.conf** to start when the system boots. These include the following:

```
monitor# sysrc sshd_enable=yes
monitor# sysrc icinga2_enable=yes
monitor# sysrc postgresql_enable=yes
```

Note that the services have not started yet, as some configuration is required before that can happen. Along with the postgres package came the system user and group of the same name, which is why setting permissions on `/var/db/postgres` is possible now.

```
monitor# chown -R postgres:postgres /var/db/postgres
```

The PostgreSQL database cluster is initialized next by running `initdb` as the postgres user with UTF-8 as encoding. Note that the postgres user needs to execute these commands, although there is now a way to do this via the service command (sometimes I'm old fashioned).

```
monitor# su postgres
postgres@monitor$ initdb -D /var/db/postgres/data -E UTF8"
```

After a successful initialization of the database cluster, the server is started using `pg_ctl`:

```
postgres@monitor$ pg_ctl start -D /var/db/postgres/data
```

Next up is the creation of the Icinga role and database, which will later load some initial tables and sequences that form the monitoring backend.

```
postgres@monitor$ createuser -drs icinga
postgres@monitor$ createdb -O icinga -E UTF8 icinga
```

Entries in `pg_hba.conf` (in the data directory) like the following allow the just created Icinga user access to the database via localhost (no need to expose it to the network for Icinga to work properly):

```
        local icinga icinga md5
        host icinga icinga 127.0.0.1/32 md5
```

Load database schema definition for Icingaweb2 as well as those for the IDO (Icinga data objects) into the database now:

```
postgres@monitor$ psql -U icinga \
  -d icinga < /usr/local/share/icinga2-ido-pgsql/schema/pgsql.sql
```

Log out of the postgres user and continue the rest of the setup. On the Icinga side, features control the functionality of the monitoring system. This includes which database backend is used. To enable PostgreSQL as the backend for the IDO, run the following command:

```
monitor# icinga2 feature enable ido-pgsql
```

In some cases, not all files and directories are owned by the Icinga system user. Running `chown` over the main `icinga2` directory ensures the permissions are properly set.

```
monitor# chown -R icinga:icinga /usr/local/etc/icinga2
```

This concludes the database part of the setup. We'll continue with the webserver setup. Although nginx is used here, other webservers like Apache2 are also perfectly fine to use. The Icinga documentation shows the necessary steps for that, too.

Nginx Setup

Icinga's web interface (aptly named Icingaweb2 since it is version 2) is a PHP application to manage hosts and services from the comfort of your browser. Events concerning any failed checks are also shown there and you can acknowledge problems or define downtimes at a central location. To configure the PHP fastCGI process manager (`php-fpm`) to handle requests coming from the webserver, enable the following options located in `/usr/local/etc/php-fpm.d/www.conf`:

```
monitor# cd /usr/local/etc/php-fpm.d
monitor# sed -i "" 's/^;listen = 127.0.0.1:9000/listen = /var/run/php5-fpm.sock/'
www.conf
monitor# sed -i "" 's/^;listen.owner/listen.owner/' www.conf
monitor# sed -i "" 's/^;listen.group/listen.group/' www.conf
monitor# sed -i "" 's/^;listen.mode/listen.mode/' www.conf
```

This basically uncomments lines that are in the file already to activate them and replaces the listen directive to use the local php5 socket instead of opening a port on the host for it. The major webserver configuration is done in the nginx.conf file located in `/usr/local/etc/nginx`, where we reference the fastCGI socket, among other things. The following configuration block is inserted after the commented `#access_log` line.

```
location ~ ^/icingaweb2/index\.php(.*)$ {
      fastcgi_pass unix:/var/run/php5-fpm.sock;
      fastcgi_index index.php;
      include fastcgi_params;
      fastcgi_param SCRIPT_FILENAME /usr/local/www/icingaweb2/public/index.php;
      fastcgi_param ICINGAWEB_CONFIGDIR /usr/local/etc/icingaweb2;
      fastcgi_param REMOTE_USER $remote_user;
}

location ~ ^/icingaweb2(.+)? {
      alias /usr/local/www/icingaweb2/public;
      index index.php;
      try_files $1 $uri $uri/ /icingaweb2/index.php$is_args$args;
}
```

Note that this does not contain an SSL section to keep this tutorial simple. It is definitely recommended (and pretty much standard practice nowadays) to generate a certificate for the webserver and configure port 443 for the secure channel configuration. The web is full of tutorials for it and services like *Let's Encrypt* make the process easy and convenient.

When the packages were installed, a file containing PHP settings meant for production use was created in **/usr/local/etc/php.ini-production**. These settings are fine for our purposes, and we activate them by copying them to be our new **php.ini** file:

```
monitor# cp /usr/local/etc/php.ini-production /usr/local/etc/php.ini
```

With this file as a base PHP configuration, we only have to replace the timezone information in it. I use this sed one-liner to place my installation within central Europe. Use whatever fits your location best:

```
monitor# cd /usr/local/etc
monitor# sed -i "" s,;date.timezone =,date.timezone = Europe/Berlin, php.ini
```

We replaced the regular sed divider / with a comma here to not confuse it with the separator between region and city. See, I'll smuggle some sed tricks into my tutorials for you as well. Thank me later... Oh by the way, this is all that is needed for the basic serving of icingaweb2 to end-users. We will now focus our attention on the Icinga configuration.

Icinga Monitoring Setup

Icinga uses various ways to monitor systems and is quite flexible about different monitoring environments and needs. For example, a host may not be available all the time (roaming user) or not have a direct connection to the central monitoring host. In the latter cases, satellite systems can relay monitoring data and check results from a different network or subnet to the central instance. In the setup that we use here, the central instance (called master) controls the execution of checks on the monitored systems. Trust is established by certificates exchanged between the master and clients. A zone defines either a geographic location (Europe, Africa, etc.) where monitoring happens or some other kind of logical grouping that makes sense in our monitoring context. For example, a whole plant, office, server room, rack, etc. could each form their own zone. Surely, a DNS zone is also possible, whatever is most useful to monitor as a whole or based on some common criteria.

First, we generate the master certificate, which we will use to monitor the central instance itself and as a base of trust for adding other monitored clients further down this tutorial. The setup is typically interactive, but here we pass all the necessary parameters on the command line:

```
monitor# icinga2 node setup --master --zone "my-zone" \
  --cn monitor.example.com \
  --listen monitor.example.com,5665 --disable-confd"
```

Here, the certificate for our central host **monitor.example.com** is generated and we instruct Icinga to not populate the **conf.d** subdirectory in **/usr/local/etc/icinga2**. We are going to create those files ourselves anyway.

Both the zone and cn are up to you to name based on your local requirements. Port 5665 should be open on your firewall to allow contacting the clients and sending the check result back. Next, we change into the directory **/usr/local/etc/icinga2** and create a couple of files and directories:

```
monitor# cd /usr/local/etc/icinga2
```

```
monitor# mkdir conf.d
```

We define an API user that has the permissions to generate a ticket through which the monitored clients request becoming part of the monitoring zone. The master will then allow or deny the ticket request and sign the client certificate with its own to establish a secure, trusted connection between the two.

The `api-users.file` contains the following:

```
object ApiUser "client-pki-ticket" {
     password = "randomstringthatmustbechanged"
     permissions = [ "actions/generate-ticket" ]
}
```

Definitely change the password line to a random string consisting of random numbers and characters, the longer the better. Next, a zones subdirectory in the `/usr/local/etc/icinga2` directory is created that holds all the information to distribute to all members of this zone. Typical examples are check information and monitoring intervals that the clients will receive from the central monitoring instance. That way, the clients need no extra local configuration, and the system administrator needs only to change the central zone configuration, which will then propagate securely to all the hosts. Since our zone is called my-zone (creativity is clearly my thing), we create a subdirectory that holds only the relevant information for clients in that zone. Other zones can be completely different, yet the monitoring configuration is located at a central place instead of each host that makes up the zone.

```
monitor# mkdir -p /usr/local/etc/icinga2/zones.d/my-zone
```

Our zones contain various information: the hosts to monitor, what to monitor (i.e., which checks to execute on each host), the monitoring intervals (how often), etc. We're starting with defining the central monitoring master in a file called `/usr/local/etc/icinga2/zones.d/my-zone/hosts.conf`. For our master, it looks like this:

```
object Host "monitor.example.com" {
     import "generic-host" // import generic settings for all hosts
     address = "monitor.example.org"
     vars.os = "FreeBSD"

     //follow convention that host name == endpoint name
     vars.agent_endpoint = name
}
```

Each host is defined as a host object and an address where it is reachable on the network. We also define a variable by which we can filter out specific hosts in Icingaweb2 for grouping purposes or define checks only for certain hosts matching these criteria. This is shown later.

The import `"generic-host"`-line is where we reference a template. Templates help us apply common settings to all hosts without having to redefine them for each host added to the file. For example, each host should have the same check interval (how often it is checked) and other similar settings. It makes this file smaller by avoiding repetitions and different hosts may use other template settings or override them with their own that are only valid for this special system.

The **templates.conf** file is located within the **zones.d/my-zone** directory and looks like this:

```
template Host "generic-host" {
      max_check_attempts = 5
      check_interval = 2m
      retry_interval = 30s
      enable_flapping = true
      check_command = "hostalive" //check is executed on the master
}

template Service "generic-service" {
      max_check_attempts = 5 // re-check 5 times before HARD state
      check_interval = 2m
      retry_interval = 1m
      enable_flapping = true
}
```

Two templates are defined here for generic hosts and services. In total, hosts and services are checked five times before an alert is generated. This is to avoid occasional packet loss or slow reacting equipment or processes, but that are generally working. The **check_interval** defines how many times the checks are executed, while the **retry_interval** defines when to check again after one check did not return in an OK-state. Definitely play around with these intervals to fit your monitoring needs. Remember that the more often you monitor, the more traffic is generated, and the data returned by the checks needs to be stored in the database, gradually making it bigger the longer you monitor.

A flapping state can happen when a host or service seems to be available, then next time it is unavailable, then available again and so on (changing rapidly between states, without seeming to become stable). Icinga is capable of detecting those states by comparing the last known state with the current one over a period of time. These flapping states are not enabled by default but are valuable information for someone debugging a problem that only happens during certain load times or busy activity. An erratic host behaving that way shows up in Icinga and should be investigated further for the root cause. The problem may also originate in the network itself, so rule out any other influences that might be responsible. A **check_command** defines which check from the ones Icinga provides needs to run by default. The **hostalive** command is basically a ping in disguise, checking to see if a host is reachable. The reason that this is only defined in the generic-host template is because services usually define a different **check_command** that fits the service and can't be easily generalized with a template.

Now that we have templates for common functionality in place, it is time to define which checks our monitoring should run and on which hosts. They are defined in the **zones.d/my-zone/services.conf** file. Here is my definition to check the disk space:

```
apply Service "disk" {
      import "generic-service"
      check_command = "disk"

// Specify remote agent as command execution endpoint, fetch host custom variable
      command_endpoint = host.vars.agent_endpoint

// Only assign where a host is marked as agent endpoint
      assign where host.vars.agent_endpoint
}
```

Applying a service means assigning it to a particular target, either a host or the result of an expression. In this example, we define that all hosts that are defined as endpoints should have the disk check running. The execution of the check is happening on the host itself, called an active check. A passive check would be running on the central monitoring instance, trying to reach the remote system, run the check, and fetch the result. Both active and passive checks can be defined for a host or target. Both have their pros and cons, but in a simple monitoring setup such as this, it is a good start to use the services that come with Icinga.

Icinga provides these common services as checks: disk, load, users, swap, procs, ping, and ssh. These provide a good initial basis for monitoring to see if swap space is low, the disk is filling up, the load is extremely high, or that there are suddenly 200 users logged in (which may or may not be normal).

A special check is to test whether our remote endpoint system is still available within the defined zone. For that, we can extend the **`services.conf`** file we just created to contain the following agent health check:

```
apply Service "agent-health" {
      check_command = "cluster-zone"

      display_name = "cluster-health-" + host.name

// Follow convention: agent zone name is FQDN same as host object name.
      vars.cluster_zone = host.name

      assign where host.vars.agent_endpoint
}
```

In addition to running the cluster-zone check command (which we don't have to know too much about to use it), we also see how a different check description is displayed in the Icingaweb2 interface by defining **`display_name`**. With this, we can see at a glance the name of the monitored system, prefixed by the string "**`cluster-health`**".

The internal Icinga database (IDO) may also fail, so it is good to monitor it as well (remember to watch the watchers). Even though newer Icinga versions are doing away with the IDO altogether, replacing it with a database on its own, small installations are perfectly fine to still use the IDO. The checks for our IDO based on PostgreSQL are defined like this (again in **`services.conf`**):

```
object Service "ido" {
            check_command = "ido"
            vars.ido_type = "IdoPgsqlConnection"
            vars.ido_name = "ido-pgsql"
            host_name = NodeName
}
```

We don't even need to apply this to any host, as this check only runs where the Icinga IDO database is installed (the central monitoring instance). The assignment `host_name = NodeName` takes care of that, since `NodeName` is defined as the name of the host by default doing the checks and collecting the results. The plugin periodically checks the IDO database and emits (upon successful execution) information about the IDO:

```
Connected to the database server (Schema version: '1.14.3'). Queries per second:
4.633 Pending queries: 21.000. Last failover: 2022-03-23 16:05:05 +0100.
```

Moving on to a different file `zones.d/my-zone/dependencies.conf` is where we define (you guessed it) dependencies for a service. This allows us to say certain services depend on the functionality of other services (and their check results) and form a logical unit. A typical example would be a web application consisting of a database and a webserver. If the database fails, the application running on the webserver does not work properly, so it makes sense to define a dependency between the two. Thus, if the database checks fail, Icinga will also mark the webserver (or the application if that is also monitored somehow) as failed. This helps in determining the impact an outage has. If a service comes back online, other dependent services also need to be checked (or restarted) to ensure continued functionality. Otherwise, the checks may report all green again, but the application may have suffered from the loss of the database and may need manual intervention to fix.

Here, we show the dependency of the agent-health check for services only:

```
apply Dependency "agent-health-check" to Service {
      parent_service_name = "agent-health"

      states = [ OK ] // Fail if parent service state switches to NOT-OK
      disable_notifications = true

// Automatically assign all agent endpoint checks as child services on the
// matched host
      assign where host.vars.agent_endpoint

// Avoid self reference from child to parent
      ignore where service.name == "agent-health"
}
```

We see how flexible Icinga is with its domain specific language using common pieces like **"apply"** together with placeholders (like **Host**, **Service** or **Dependency**) to define what and how the monitoring should take place. The **`agent-health`** checks trigger if a state other than OK (like or "FAILED" or "UNREACHABLE") is detected. To not define this for every single host we have, and not forget it for any new hosts added later, we use the assign keyword again to apply this to all hosts defined as an endpoint.

Groups of hosts or services help to keep an overview of systems with common tasks or criteria, like webservers, database servers, front-end hosts, firewalls, etc. This is what **`groups.conf`** defines, but is optional when the infrastructure to monitor is small or too diverse for any commonalities:

```
object HostGroup "FreeBSD-servers" {
     display_name = "FreeBSD Servers"
     assign where host.vars.os == "FreeBSD"
}
```

Remember the object Host **`"monitor.example.com"`** definition in **`hosts.conf`** above? We defined a local variable **`vars.os`**. We can now filter on the value of this variable using the **`"assign where"`** statement. Tools that automatically add entries for new hosts in the infrastructure to **`hosts.conf`** may also hold the information about what operating system is used (among others), hence Icinga groups these systems in the Icingaweb2 display. ServiceGroups are defined similarly. That way, a report may contain the number of systems that are periodically checked for certain services. Webservers may run different checks than database servers, but as service groups, it is easy to either apply them to new hosts as a whole or define a mixture of both to form a whole new monitoring target.

The last file that I want to show is the **`users.conf`** file that holds all the information about users that Icinga understands and notifies when some checks fail. A basic definition may look like this:

```
object UserGroup "icingaadmins" {
     display_name = "Icinga Admin Group"
}

object User "icingaadmin" {
     display_name = "Icinga 2 Admin"
     groups = [ "icingaadmins" ]
     email = "icinga@localhost"
}

object User "Helpdesk" {
     email = "ticket@example.org"
     display_name = "The Friendly Helpdesk Folks"
     groups = [ "icingaadmins" ]
}
```

Users may be part of other groups as in this example where the Helpdesk user is part of the Icinga Admin Group. Individual users may be assigned to only a certain host or a set of services (experts in their field), but not to the overall infrastructure that is monitored.

Notification rules define who is contacted when and by which method (email by default, but pagers, SMS, and even various instant messengers are possible). Escalations to a different group after a certain amount of time can happen when a problem has not been dealt with (or at least acknowledged), to define certain service-level agreements or for paying (our impatient) customers.

Other files make up the Icinga monitoring and all are well defined in the documentation. For now, let's start all the services to get our basic monitoring infrastructure going. Especially after all the extra files are added, Icinga needs to know about them, so we restart that particular service:

```
monitor# service postgresql restart
monitor# service php-fpm start
monitor# service nginx start
monitor# service icinga2 restart
```

The Icingaweb2 service is configured via the web browser for which a token is needed because we don't want a random stranger driving by our freshly installed monitoring by accident to misconfigure it. The token is generated and emitted with the following command:

```
monitor# icingacli setup token create --config=/usr/local/etc/icingaweb2
monitor# chown -R www:www /usr/local/etc/icingaweb2
```

The token is now readable from the browser and when pasted into the web form, the remaining setup steps for Icingaweb2 can happen. Fill in the details like the database users we created and other information like the admin user and its password. At the end, the Icingaweb2 login will be presented, and you can access all your monitored hosts and services from this central place.

Adding New Host Endpoints

After the initial excitement about Icinga's functionality you may be wondering how to add more objects to monitor. We will demonstrate this with a new host and show all the steps necessary to include it into our monitoring.

On a freshly installed host (we use FreeBSD here) called `client.example.org`, install the icinga2 package.

```
client# pkg install icinga2
```

Since this is a certificate-based authentication between this host and the central Icinga monitoring instance, we need to ensure that the directory holding the certificates exists and has the right ownership:

```
client# mkdir /var/lib/icinga2/certs
client# chown icinga:icinga !$
client# chown -R icinga:icinga /usr/local/etc/icinga2
```

Next, we enable the icinga2 service to start at system bootup:

```
client# sysrc icinga2_enable=yes
```

A client certificate is generated next using the "icinga2 pki" subcommand. While this command is interactive, we can also provide all necessary parameters directly on the command line to ease automation later when adding hundreds of hosts. Note that this has to run on the central monitoring instance.

```
monitor# icinga2 pki new-cert --cn client.example.org \
  --key /var/lib/icinga2/certs/client.example.org.key \
  --csr /var/lib/icinga2/certs/client.example.org.csr"
```

The file ending in `.csr` is the certificate signing request, which is now used in combination with the previously generated master key to create a new signed client certificate (`example.org.crt`).

```
monitor# icinga2 pki sign-csr \
  --csr /var/lib/icinga2/certs/client.example.org.csr \
  --cert /var/lib/icinga2/certs/client.example.org.crt"
```

When we ran "`icinga2 node setup --master`" at the beginning of this article to generate the master certificate to sign the others, a file called `monitor.example.org.crt` was created in `/var/lib/icinga2/certs/`. Transferring this to the client in a secure way is necessary to validate the server certificate. There are various ways to do this, depending on how much you trust the client and any users connected to it, as well as the network (or medium) between the two.

```
monitor# scp /var/lib/icinga2/certs/monitor.example.org.crt \
  client.example.org:/var/lib/icinga2/certs/
```

Next, import the certificate into the client and tell Icinga to trust it.

```
client# icinga2 pki save-cert --trustedcert \
  /var/lib/icinga2/certs/monitor.example.org.crt \
  --host client.example.org"
```

A new ticket is created for the client on the monitoring server to establish a trust relationship. Essentially, the client asks to be part of the monitoring infrastructure. These requests may be generated automatically and signed at a later time (after a review by a human or third entity).

```
monitor# icinga2 pki ticket --cn client.example.org
```

Note the resulting ticket output on the commandline (in our case 4f76d2ec-da535753e9180838ebffbcbca242fe61), we'll need it in this next step on the client. It will take the generated ticket from the central monitoring instance and generate configuration files just like we did manually when we set up our monitor. The zone relationship is established, making the monitor a parent of the client, establishing trust between them. Additionally, we tell the client to accept commands and configuration changes sent to it by the monitor. This is optional and clients may also choose to make their own configuration choices, independent of the host. Having the configuration on the central server and controlling the configuration of each client there eases the burden of keeping them in sync on every monitored host. When changing a setting like a new monitoring interval, it only needs to be set once and the clients will apply the changes coming from the monitor locally.

```
client# icinga2 node setup --ticket 4f76d2ecda535753e9180838ebffbcbca242fe61 \
  --cn client.example.org --endpoint monitor --zone client.example.org \
  --parent_zone my-zone --parent_host monitor.example.org \
  --trustedcert /var/lib/icinga2/certs/monitor.example.org.crt \
  --accept-commands --accept-config --disable-confd"
```

Before we start our Icinga instance, we need to verify that all files were written correctly and conform to Icinga's logic. To do that, we tell the Icinga daemon to perform a configuration check with the following command:

```
client# icinga2 daemon -C
```

If there are any errors, Icinga tries to help pinpoint the file and line in question. Typical errors may be providing the wrong names for the parent or zone. Once the validation is complete, a new entry for this new client needs to be added on the monitoring server to include it in future check executions. On `monitor.example.org`, edit

```
/usr/local/etc/icinga2/zones.d/my-zone/hosts.conf
```

and add the following lines of configuration, ensuring that this is placed before the central hosts object definition:

```
object Host "client.example.org" {
      import "generic-host" // import generic settings for all hosts
      display_name = "My Client Host"
      address = "client.example.org"

      vars.os = "FreeBSD"
//follow convention that host name == endpoint name
      vars.agent_endpoint = name
}
```

Host objects are fairly simple to define and don't contain any new fields that we have not yet seen from our previous edits when we added the central server itself. As before, we also need to define that this host is an endpoint (no further monitoring clients are below it and that it is not a parent of another host) as well as the zone it belongs to. Typically, these entries are placed before the line containing `'object Zone "director-global" {'` and look like this:

```
object Endpoint "client.example.org" {
// client connects itself
      host = "client.example.org"
      log_duration = 0
}

object Zone "client.example.org" {
      endpoints = [ "client.example.org" ]
      parent = "my-zone" // Establish zone hierarchy
}
```

In the zone object, we only need to define the name of our endpoint, referencing the host definition we did earlier in the file. The parent zone is the one that was generated when we created the monitor certificate. There should already be entries for it in the file by the Icinga configuration. The log duration entry as endpoint attribute instructs the endpoint how long to store a replay log of all check results on the client if the connection to the parent is lost. Once the connection is reestablished, the client will replay the log and all the data will be sent to the parent. Since the parent schedules all the checks to be run on the monitored systems, setting this to zero is fine.

We're done wading through configuration files on both hosts. The only thing left is to start the icinga2 service on the client and on the server to read the configuration changes we made.

```
client# service icinga2 start
monitor# service icinga2 restart
```

The new client should now appear in the Icingaweb2 overview as pending. When the next scheduled check interval happens, the client is contacted in a secure way (since they exchanged certificates, remember?) checks are executed, and results delivered to the central host.

Congratulations, you can now enjoy monitoring your infrastructure for common services and add new hosts to it. Make sure to check out the Icinga configuration for various monitoring-related information and further ways of configuring your Icinga installation to fit your needs.

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.

Tom Once Again, Does Stupid Things with a Computer: activitymonitor.sh

BY TOM JONES

At EuroBSDCon in Vienna this year, I spoke about my work examining the performance of QUIC on FreeBSD and Linux

One core part of my measurement approach was CPU saturation during a network transfer that is using all the available CPU cycles for a sender. I used CPU saturation two ways, the first was to detect if the CPU was the bottleneck in the system. If it wasn't the bottleneck, then I needed to be sure that the network was the bottleneck, and not some other component I wasn't trying to measure. And second, once I had controlled for CPU saturation and made sure another bottleneck wasn't at play, I could use the actual CPU usage for a test to estimate how fast the system could send if it could saturate the network card with UDP. This was really helpful, as with all my measurements, my network interfaces can only manage ~6Gbit/s with UDP traffic in any form on any of the tested operating systems. But this doesn't saturate the CPU, instead it runs at about 70% utilization. With good CPU measurements, I could invent a metric to optimistically predict what the processor could do if the network interface wasn't getting in the way.

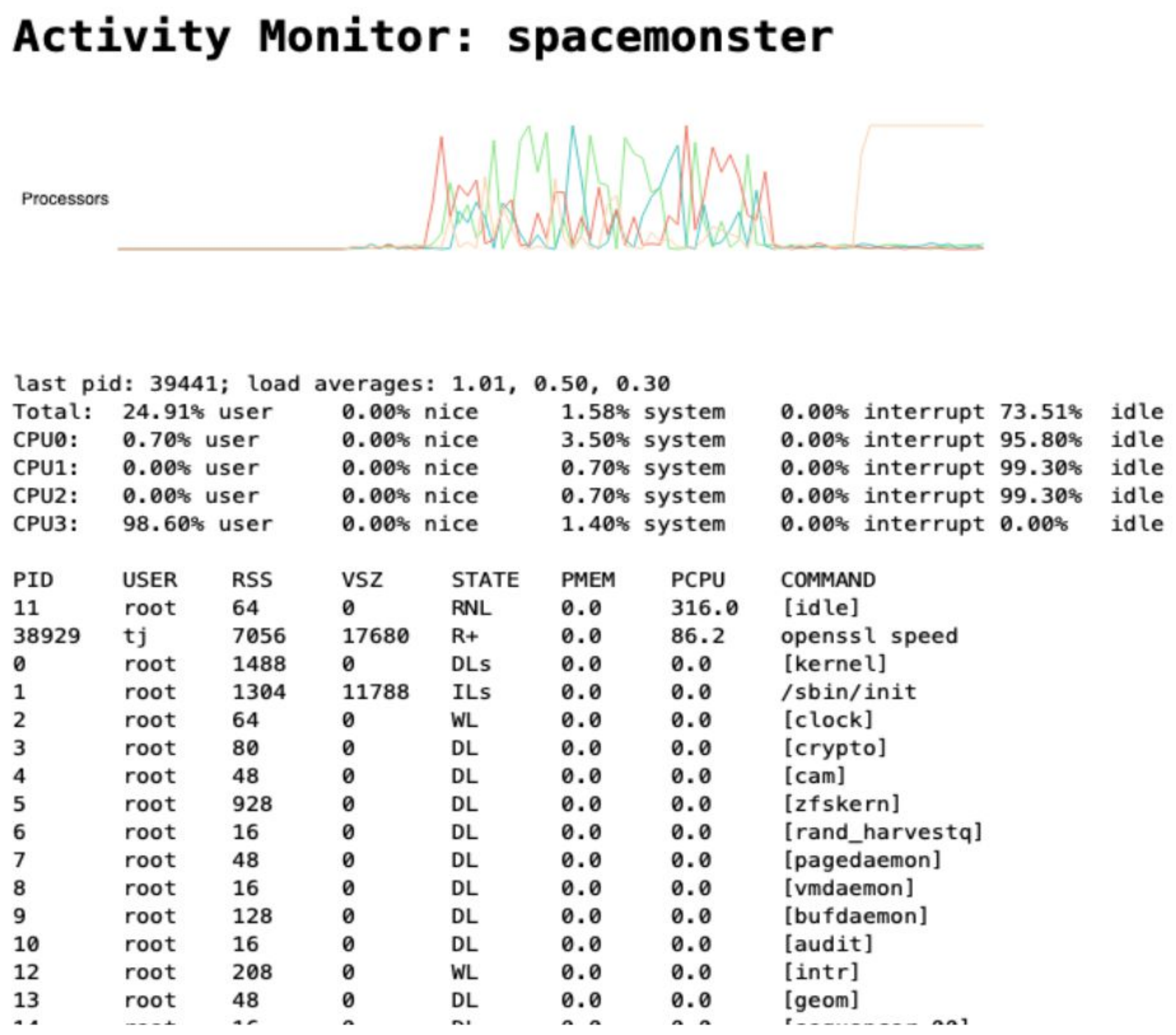
It seemed a good idea to find a well-reasoned approach to looking at CPU performance during a network test.

My EuroBSDCon presentation was based on yet unpublished academic work, and it seemed a good idea to find a well-reasoned approach to looking at CPU performance during a network test. Some of my tests were inspired by work that Fastly did to compare the computational efficiency of QUIC compared to TCP. While Fastly seems to have established a great testbed, the only mechanism I could figure out from their writing was to "eyeball" top.

This wasn't really good enough for publishing results or for accurate evaluation of thousands of test results. If looking at top wasn't good enough, maybe I could, instead, figure out how top does its own looking and reimplement that?

How Does top Estimate CPU Utilization?

top is probably the first tool we go to when we wonder what is happening on a system. Even with its universal usefulness, top is a very simple program that actually draws things nicely on the screen and it has some abstractions on the machine to make the code a little more portable. The CPU utilization interface is provided by FreeBSD in the form of two sysctl nodes:



```
Activity Monitor: spacemonster

last pid: 39441; load averages: 1.01, 0.50, 0.30
Total:  24.91% user    0.00% nice    1.58% system   0.00% interrupt 73.51%  idle
CPU0:   0.70% user     0.00% nice    3.50% system   0.00% interrupt 95.80%  idle
CPU1:   0.00% user     0.00% nice    0.70% system   0.00% interrupt 99.30%  idle
CPU2:   0.00% user     0.00% nice    0.70% system   0.00% interrupt 99.30%  idle
CPU3:   98.60% user    0.00% nice    1.40% system   0.00% interrupt 0.00%   idle

PID     USER    RSS     VSZ     STATE   PMEM    PCPU    COMMAND
11      root    64      0       RNL     0.0     316.0   [idle]
38929   tj      7056    17680   R+      0.0     86.2    openssl speed
0       root    1488    0       DLs     0.0     0.0     [kernel]
1       root    1304    11788   ILs     0.0     0.0     /sbin/init
2       root    64      0       WL      0.0     0.0     [clock]
3       root    80      0       DL      0.0     0.0     [crypto]
4       root    48      0       DL      0.0     0.0     [cam]
5       root    928     0       DL      0.0     0.0     [zfskern]
6       root    16      0       DL      0.0     0.0     [rand_harvestq]
7       root    48      0       DL      0.0     0.0     [pagedaemon]
8       root    16      0       DL      0.0     0.0     [vmdaemon]
9       root    128     0       DL      0.0     0.0     [bufdaemon]
10      root    16      0       DL      0.0     0.0     [audit]
12      root    208     0       WL      0.0     0.0     [intr]
13      root    48      0       DL      0.0     0.0     [geom]
```

```
$ sysctl hw.ncpu
4
$ sysctl kern.cp_time
kern.cp_time: 3832370 11408 3650627 44926 2061043745
$ sysctl kern.cp_times
kern.cp_times: 1080959 3035 954019 4354 515101995 1062132 2823 815176 1143
515263088 960320 3419 980090 37760 515162773 728956 2131 901334 1668 515510273
```

The first node **`kern.cp_times`**, returns 5 values for the processor which report the total time since boot for time spent in user, nice, system, interrupt and idle. **`kern.cp_times`** reports the same 5 values of the each of the processors in the system. By using **`kern.cp_times`** with **`hw.ncpu`**, we can break down this list. By working with both sysctls we get the total system time usage since boot and the per-processor time usage since boot.

Usage since boot can be helpful in understanding what the machine has been up to and it is useful to see how the system usage is changing over short periods of time. top on startup displays the total system time breakdown, but once it refreshes (by default after 1 second), it then shows how these fields have changed over that second.

Plotting

With the ability to very easily reproduce what top does, I wondered if I could grab the system CPU utilization periodically and plot it out. I figured I could include these plots next to throughput plots to show that between tests, the host was almost entirely idle, and it saturated the expected single core when the test was running.

Over the years, I have had different go-to tools when it comes to plotting, but with this work, I got tired of custom things and wanted simplicity. I could have plotted out the utilizations with the easiest method—a spreadsheet, but I thought something that gave me a little more control would be nice.

In other recent work, I have used web-based tooling for plots. The c3js library gives nice interactive charts but struggles when there are a lot of data points (more than the low 10s of thousands). Given that I was also going to look at network usage, the amount of data spread out over a minute of recordings was going to be a lot.

When thinking about tooling, I recalled a recent article I had written for Klara Systems on inetd. When writing that article, I created my own simple inetd service that implemented the datetime service with a shell script.

Could I deliver my CPU usage information live from the host using inetd?

activitymonitor.sh

This leads us to activitymonitor.sh, a hacky creation that abuses all good norms to give simple plots in a web browser.

activitymonitor.sh is a single shell script that consists of three parts:

- A script run by inetd.
- A basic html page.
- Some javascript to update a live page.

inetd is an Internet service runner. The full history and features of inetd fall a little outside the scope of this short article, but inetd was used for on-demand launch applications when hosts were too small to have waiting services hanging around in memory. inetd handles listening for traffic. When connections are received or datagrams arrive (for UDP based protocols), inetd launches either a built-in handler or a specified program. The program is given reads from the Internet connection on standard in. Any writes to standard out are sent out over the network. This is a really simple interface, but powerful enough to implement plain text server protocols.

A Small Script

The small script is quite simple. It has two main components and then a blob of data appended to it which is the web page and javascript.

First, the shell script deals with being a guest of inetd and parsing the http headers. The input to the script will be the HTTP headers the client sends when making its request.

```
# Read client headers, we only really care if one is data.json.
h=""
while read -t 1 h
do
    log $h

    if echo $h | grep -q "data.json";
    then
        page="data.json"
        contenttype="application/json"
    else
    fi
done
```

```
echo "HTTP/1.0 200 OK"
echo "Content-Type: $contenttype"
echo
```

The script uses the **read** built-in command with a timeout, meaning the script will consume all the input on the socket until there is a 1-second gap between incoming lines before proceeding. This **read** timeout is the rate-limiting mechanism. The headers are checked to see if the data url is being requested, if not then it delivers the base html page.

The data url path of the script is where we gather up interesting data about the host. activitymonitor.sh implements most of the default interface of top. To do so, it gathers up the required information using FreeBSD base commands and then encodes them into a JSON blob delivered to the requester.

```
if ["$contenttype" == "text/html" ]
then
    indexstart=$((cat -n $0 | grep -e 'INDEX START'\
       | awk '{print $1}' | tail -n 1+1))
    sed -n"$indexstart"',$p' $0
elif ["$contenttype" == "application/json" ]
then
     psout=$(ps -ax -o \
        "user,pid,%cpu,cpu %mem,vsz,rss,state,command"\
        --libxo json)
    vmstatout=$(vmstat -libxo json)
    netstatout=$(netstat -bi -libxo json)

    # kern.cp_time(s) gives us 5 numbers for the system:
    # user nice system interrupt idle
    # kern.cp_times gives us hw.ncpu entries for those 5 values
    totalcputime=$(sysctl -n kern.cp_time)
    percputime=$(sysctl -n kern.cp_times)
    ncpu=$(sysctl -n hw.ncpu)
    loadavg=$(sysctl -n vm.loadavg)
    lastpid=$(sysctl -n kern.lastpid)
    hostname=$(sysctl -n kern.hostname)

    system=$(printf ‘{"hostname":"%s",
        "cp_time":"%s", "cp_times":"%s", "ncpu":"%s",
        "loadavg":"%s", "lastpid":"%s"}’ "$hostname"
        "$totalcputime" "$percputime" "$ncpu"
        "$loadavg" "$lastpid")
    log $system

    physmem=$(sysctl -n hw.physmem)
    pagesize=$(sysctl -n hw.pagesize)
    pagecount=$(sysctl -n vm.stats.vm.v_page_count)
    wirecount=$(sysctl -n vm.stats.vm.v_wire_count)
    activecout=$(sysctl -n vm.stats.vm.v_active_count)
    inactivecount=$(sysctl -n vm.stats.vm.v_inactive_count)
    cachecount=$(sysctl -n vm.stats.vm.v_cache_count)
    freecount=$(sysctl -n vm.stats.vm.v_free_count)

    memory=$(printf '{"physmem":"%s", "pagesize":"%s",
        "pagecount":"%s", "wirecount":"%s",
        "activecout":"%s", "inactivecount":"%s",
        "cachecount":"%s", "freecount":"%s" }'
        "$physmem" "$pagesize" "$pagecount"
        "$wirecount" "$activecout" "$inactivecount"
        "$cachecount" "$freecount")

    log $totalcputime
    # deliver the data json
    printf '{"system":%s, "memory":%s, "ps":%s,
        "vmstat":%s, "netstat":%s}'"$system"
        "$memory" "$psout" "$vmstatout" "$netstatout"

fi
exit    # don't continue into the web page
```

The first set of information the script collects comes from FreeBSD tools that have libxo support. Libxo is a very powerful FreeBSD feature—base tools with support can give output in JSON natively. We grab the output of `ps`, `vmstat` and `netstat`. This lets us display processes, vm system information, and network statistics such as interface rates.

The second set in the script is concerned with getting information directly from the sysctl interface. Right now, we get the kern.cp_time(s), number of cpus, hostname, load average and base statistics about memory. Each of these has to be bundled into a JSON object by hand by using `printf`.

All of this information is then built into a JSON object which the script prints out after a simple response header. inetd then feeds back into the connecting socket and returns to the client as the body of the http response.

A Basic Web Page

The activitymonitor.sh script embeds a tiny webpage within itself which it delivers if the data url isn't requested. The page has a header, some canvases to give the javascript somewhere to draw plots, and a pre block for the top-style process list. It also embeds the javascript that causes all the magic to happen.

The html page (and javascript) is appended to the end of the shell script and marked with a `'INDEX START'` tag. activitymonitor.sh searches itself for this tag and takes everything after the tag as content to deliver, using sed to cut it up.

Some Javascript

The javascript does all the heavy lifting to parse out information from the data and to give us a user interface. It pulls out and processes the `kern.cp_times` values and calculates deltas we need to draw the plots.

The main functional thing it does other than drawing is to request data from the data side of activitymonitor.sh. Once the basic web page has loaded, the script will kick off a task to fetch the `'/data.json'` url.

When data successfully arrives it pulls in the fields it needs to from the JSON result and merges in new data to calculate what should be displayed.

Finally, at the end it calls `getdata` again to start off this task. Because of the `read` timeout in activitymonitor.sh, this will happen with a 1-second gap between results.

This is a Bad Idea

activitymonitor.sh was a fun little project that got away from me and almost became a usable tool. The CPU plots helped me understand that the FreeBSD scheduler moves processes between CPUs very eagerly and helped me devise a measurement strategy that would account for this.

While a fun project, it is not something that should be used by anyone in the real world. Instead, it is an example of the power of composability of the tools in the FreeBSD base system. Other than sysctl, all the tools we use natively output JSON and can be fed into powerful user interface languages.

This native support for JSON makes it easy to consume output from standard tools and lets automation occur with the data a human would consume trivially. This gives us power to build systems that are machine readable with the same data we read on the screen. It is a small enhancement beyond the traditional UNIX interfaces, but an incredibly powerful one.

activitymonitor.sh is available here

inetd configuration such as the following is required:

```
http-alt stream tcp     nowait  tj      /home/tj/code/activitymonitor.sh
activitymonitor.sh
```

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

Pragmatic IPv6 (Part 4)

BY HIROKI SATO

As mentioned in earlier columns, IPv6 looks similar to IPv4 except for the address format. However, these two protocols work independently, and of course, there are features specific to each protocol. This column will take a look at IPv6's unique characteristics based on multiple addresses and its practical use, and NDP, Neighbor Discovery Protocol, which is responsible for the L2-to-L3 address resolution and discovery of hosts and routers on the same network.

Many IPv6 Addresses on Your Box

When you use IPv6 on your FreeBSD box, you will usually get multiple addresses like this:

```
% ifconfig vlan100
vlan100: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500
        options=80003<RXCSUM,TXCSUM,LINKSTATE>
        ether a4:ba:db:e0:ae:33
        inet6 2001:db8:fb5d::1prefixlen64
        inet6 fe80::a6ba:dbff:fee0:ae33%vlan100 prefixlen 64 scopeid 0x6
        inet6 fe80::ffff:2:7b%vlan100 prefixlen 64 scopeid 0x6
        inet6 fe80::ffff:2:35%vlan100 prefixlen 64 scopeid 0x6
        inet 192.168.100.1 netmask 0xffffff00 broadcast 192.168.100.255
        groups: vlan
        vlan: 100 vlanproto: 802.1q vlanpcp: 0 parent interface: lagg0
        media: Ethernet autoselect
        status: active
        nd6 options=21<PERFORMNUD,AUTO_LINKLOCAL>
```

This is a live example from one of the author's boxes. The `/etc/rc.conf` contains the following:

```
ifconfig_vlan100_ipv6="inet6 2001:db8:fb5d::1/64"
ifconfig_vlan100_alias0="inet6 fe80::ffff:2:7b/64"
ifconfig_vlan100_alias1="inet6 fe80::ffff:2:35/64"
```

You can see four IPv6 addresses in the output of `ifconfig(8)`. The `2001:db8:fb5d::1` is a GUA[1], and the other three are LLAs[2]. In `/etc/rc.conf` file, only two LLAs are explicitly specified. Why do we have four?

Automatically-configured LLA

Remember that the following will happen when an `ifconfig_IF_ipv6` is specified:

• The `IFDISABLED` flag is removed, and

• an LLA based on the L2 address of the interface will be automatically configured.

More precisely, all IPv6-capable interfaces have `AUTO_LINKLOCAL` flag by default in the kernel level, and it will configure an LLA when the interface is becoming "up". The `rc.d(8)` scripts will add `IFDISABLED` flag when no `ifconfig_IF_ipv6` is specified to prevent the interface from configuring an LLA. This is a seatbelt for people who want IPv4 only. As long as you have no `ifconfig_IF_ipv6` line, the interface will get no IPv6 address. The automatically configured LLA is an L3[3] address, so one on the same network can try to access your box over IPv6 TCP or UDP. For this reason, the LLA is not configured unconditionally.

Note that an LLA is mandatory if you want to use an IPv6 GUA. Unlike IPv4, you always have to configure an LLA. This is the reason why there is the `AUTO_LINKLOCAL` flag by default and the kernel configures one. While you can remove the LLA manually, various odd behaviors will occur.

Modified EUI-64 Format Interface Identifiers

Let's see the automatically-configured LLA again. The prefix is always `fe80::/64`. The IID is filled by using the L2 address. If you are using Ethernet, it is the IEEE 802 48-bit MAC, also known as Ethernet MAC address. The Ethernet MAC address is 48-bits long. You can find "`ether`" keyword in the output of the `ifconfig(8)` command:

```
ether a4:ba:db:e0:ae:33
inet6 fe80::a6ba:dbff:fee0:ae33%vlan100 prefixlen 64 scopeid 0x6
```

The IID looks similar to the MAC address but not the same. This is called "modified EUI-64 format interface identifier" and is generated from the 48-bit MAC address. The generation algorithm[4] is simple. Let's compare the IID to the MAC address octet-by-octet[5]:

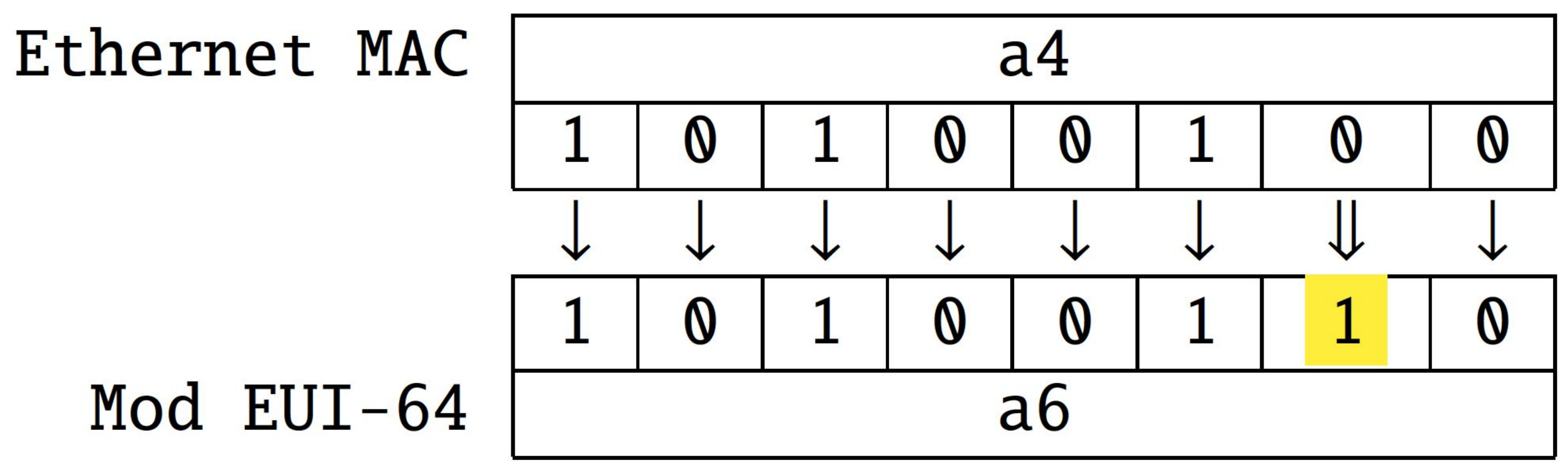
Ethernet MAC	a4	ba	db			e0	ae	33
	⇓	↓	↓			↓	↓	↓
Mod EUI-64	a6	ba	db	ff	fe	e0	ae	33

The IID is 64-bit long, so two octets must be filled. The "`0xff`" and "`0xfe`" in the center of the IID are always added. In other words, if the IID has `0xfffe` at the center, it is generated by an EUI-48 MAC address. There is one more difference—the first octet has slightly been changed. The first and second bits (from the LSB[6]) of the first octet in the MAC address have the following meanings:

first bit: "individual"(0) or "group"(1),

second bit: "universal"(0) or "local"(1).

The "individual" means unicast (i.e., 1-to-1 communication), and the "group" means multicast or broadcast (1-to-n communication). When using a real hardware NIC, not a virtual one, it has a unique MAC address assigned by the vendor. In this case, the address is universally unique, and the second bit of the octet is 0. However, in the modified EUI-64 format, the second bit is specified as an inverted value of the MAC address. So, in most cases, the second bit of the first octet is 1. The first octet, "0xa4" in this example, is changed in the following way. The bit array of the hexadecimal value "0xa4" is "10100100". The second bit from the rightmost bit in the array will be inverted, and you will get "0xa6" in the IID:



Currently, on FreeBSD, the automatic LLA and GUAs set by SLAAC use this algorithm. There are two topics you have to be aware of. One is the reason why the second bit is flipped, and another is a problem with the generated IIDs.

Problems of Modified EUI-64 IID

The reason for inverting the second bit is to make it easy to configure an address manually. The MAC address on a real hardware NIC has a "universal" bit, so the first octet of the generated IID will never be "0x00." Using this fact, you can configure an IID that does not conflict with automatically-configured ones. For example, "0:0:0:1" or "::1" is an address you can choose because the first octet is 0x00. If this inversion were not defined, you would have to use something like "0200:0:0:1."

Although the modified EUI-64 IID is popular in IPv6 implementations, privacy is an issue. As you can imagine, the MAC address in the generated address can be used to track your network activity. While the IPv6 address space is enormous for address scanning, the EUI-64 IID address space is smaller than that. RFC 7721, "Security and Privacy Considerations for IPv6 Address Generation Mechanisms," has extensively discussed the security and privacy aspects of the algorithm.

There are two algorithms to mitigate them. RFC 8981, "Temporary Address Extensions for Stateless Address Autoconfiguration in IPv6," defines "temporary address." The temporary address is an automatically-configured IPv6 address by SLAAC with a random IID and is valid for a short period of time. This is intended for the source address when initiating an outgoing session. It is difficult for an outside entity to predict the IID that is employed for temporary addresses. FreeBSD partially supports this extension, and you can enable it by setting the following sysctl variable:

```
# sysctl net.inet6.ip6.use_tempaddr=1
```

After enabling this, two addresses will be configured by SLAAC:

```
# ifconfig vlan84
vlan84 : flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500
         options=80003<RXCSUM,TXCSUM,LINKSTATE>
         ether a4:ba:db:e0:ae:33
         inet6 2001:db8:fb5d:8001::42 prefixlen 64
```

```
        inet6 fe80::a6ba:dbff:fee0:ae33%vlan84 prefixlen 64 scopeid 0x7
        inet6 fe80::ffff:2:7b%vlan84 prefixlen 64 scopeid 0x7
        inet6 fe80::ffff:2:35%vlan84 prefixlen 64 scopeid 0x7
        inet6 2001:db8:fb5d:8001:a6ba:dbff:fee0:ae33 prefixlen 64 autoconf
        inet6 2001:db8:fb5d:8001:7c36:33b7:b967:382f prefixlen 64 autoconf
        temporary groups: vlan
        vlan: 84 vlanproto: 802.1q vlanpcp: 0 parent interface: lagg0 media:
        Ethernet autoselect
        status: active
        nd6 options=23<PERFORMNUD,ACCEPT_RTADV,AUTO_LINKLOCAL>
```

Note that the **`vlan84`** is a different interface from **`vlan100`** in the previous example. You can see two addresses with the "autoconf" keyword. SLAAC and a modified EUI-64 IID generate the first one, and the second one has a random IID and is labeled with "temporary." The temporary address will be automatically changed once in 24 hours by default. Note that if you already have a SLAAC address and then enables the **`use_tempaddr`** variable, you need to remove the SLAAC address first.

This extension is useful to some extent, but the current FreeBSD implementation has the following problem:

- You cannot control the generation of temporary addresses per-interface basis. When enabled, all of the interfaces accepting Router Advertisement will have a temporary address.
- The address generation algorithm is based on an old specification in RFC 4941, not in RFC 8981.

There are also several pitfalls when you try to use it. This topic will be covered in later columns. At this moment, you should learn that the modified EUI-64 IID is popular, and FreeBSD uses it when autoconfiguration is performed. The auto-configured address is a normal address that can be used for TCP or UDP communication. Thus, you might want to be aware that someone on the same network segment can try to access your box using the address.

Another algorithm is a stable IPv6 interface identifier proposed in RFC 7217, "A Method for Generating Semantically Opaque Interface Identifiers with IPv6 Stateless Address Autoconfiguration (SLAAC)." This is a drop-in replacement of the modified EUI-64 IID and a solution to security and privacy issues due to the MAC address used. FreeBSD has not supported this yet, but the author is working on the implementation. This will also be covered in the later columns.

Non-Unicast Addresses

When an IPv6 address is configured, your FreeBSD box actually has more addresses. Try **`ifmcstat`** command like this:

```
% ifmcstat -i vlan84 -f inet6
vlan84 :
        inet6 fe80::a6ba:dbff:fee0:ae33%vlan84 scopeid 0x7
        mldv2 flags=2 <USEALLOW > rv 2 qi 125 qri 10 uri 3
                group ff02::1:ff67:382f%vlan84 scopeid 0x7 mode exclude
                        mcast-macaddr33:33:ff:67:38:2f
                group ff02::202%vlan84 scopeid 0x7 mode exclude
                        mcast-macaddr 33:33:00:00:02:02
                group ff02::1:ff02:35%vlan84 scopeid 0x7 mode exclude
                        mcast-macaddr 33:33:ff:02:00:35
                group ff02::1:ff02:7b%vlan84 scopeid 0x7 mode exclude
                        mcast-macaddr 33:33:ff:02:00:7b
                group ff02::1:ffe0:ae33%vlan84 scopeid 0x7 mode exclude
                        mcast-macaddr 33:33:ff:e0:ae:33
                group ff01::1%vlan84 scopeid 0x7 mode exclude
                        mcast-macaddr 33:33:00:00:00:01
                group ff02::2:a17e:3d85%vlan84 scopeid 0x7 mode exclude
                        mcast-macaddr 33:33:a1:7e:3d:85
                group ff02::2:ffa1:7e3d%vlan84 scopeid 0x7 mode exclude
                        mcast-macaddr 33:33:ff:a1:7e:3d
                group ff02::1%vlan84 scopeid 0x7 mode exclude
                        mcast-macaddr 33:33:00:00:00:01
                group ff02::1:ff00:42%vlan84 scopeid 0x7 mode exclude
                        mcast-macaddr 33:33:ff:00:00:42
```

Addresses after the keyword "group" are ones assigned to the interface **`vlan84`**. You can even try to send a ping to the addresses and get a response:

```
% ping6 ff02::1:ff67:382f%vlan84
PING6 (56=40+8+8 bytes) fe80::a6ba:dbff:fee0:ae33%vlan84 --> ff02::1:
ff67:382f%vlan84
16 bytes from fe80::a6ba:dbff:fee0:ae33%vlan84, icmp_seq=0 hlim=64
time=0.073 ms
16 bytes from fe80::a6ba:dbff:fee0:ae33%vlan84, icmp_seq=1 hlim=64
time=0.044 ms
16 bytes from fe80::a6ba:dbff:fee0:ae33%vlan84, icmp_seq=2 hlim=64
time=0.054 ms
^C
--- ff02::1:ff67:382f%vlan84 ping6 statistics ---
3 packets transmitted, 3 packets received, 0.0% packet loss
round-trip min/avg/max/std-dev=0.044/0.057/0.073/0.012ms
```

However, they were not in the output of the **`ifconfig`** command. What are they?

Well-Known Addresses and Their Application

You can see that all of the addresses have the same prefix "**`ff00::/12`**". Do you remember that in the last column, **`ping6(8)`** utility was used with the address "**`ff02::1`**" to check if the IPv6 communication works or not? "**`ff02::1%vlan84`**" is listed in the sixth entry.

An address with the prefix "**`f000::/4`**" is an IPv6 multicast address. It is used for 1-to-n communication. When you send a ping to this address, you may receive one or more responses. The prefix determines whether the address is multicast or not and the scope to which the address belongs. And the purposes of each address are also defined. All of the well-known multicast addresses and their application can be found in "IPv6 Multicast Address Space Registry"[7].

Let's see the address listed in the above example. An address with "`ff01::/16`" is an interface-local multicast address, and one with "`ff02::1/16`" is a link-local multicast address. You always need the `%zoneid` part.

"`ff02::1`" is the all-nodes address whose scope is link-local. This means that all IPv6-capable nodes on the same network have this multicast address. If you send a ping to "`ff02::1%vlan84`", you will get a lot of responses from the network on `vlan84`. A multicast address does not belong to a single node. Thus, we usually say that a host "joins" the address. All IPv6-capable hosts automatically join the all-nodes multicast address. There is no need to configure it. This is why you can always use "`ff02::1`" as a tool to check if there is an IPv6 node on the interface. "`ff02::2`" is the link-local all-routers address. This is not included in the output of `ifmcstat` command because this machine is not configured as an IPv6 router. If you send a ping to "`ff02::2%vlan84`," you can check if there is a router on `vlan84`.

"`ff01::1`" is the interface-local all-nodes address. The "interface-local" means an isolated group where only the interface belongs. You will receive a response from the same interface by sending a ping to this address.

"`ff02::202`" is the multicast address which the `rpcbind(8)` deamon uses.

Of course, these addresses are not only for the `ping6(8)` utility. They are used when ICMPv6 or some other 1-to-n communication is required. If all IPv6 nodes need to receive it, "`ff02::1`" is used. If all IPv6 routers need to do it, "`ff02::2`" is used. Therefore, most local ICMPv6 control messages will be delivered using a combination of an LLA on the host and these well-known multicast addresses.

Let's see more concrete examples. What is remaining? The following addresses are still unclear:

```
ff02::1:ff67:382f%vlan84 scopeid 0x7 mode exclude
ff02::1:ffe0:ae33%vlan84 scopeid 0x7 mode exclude
ff02::1:ff00:42%vlan84 scopeid 0x7 mode exclude
ff02::1:ff02:35%vlan84 scopeid 0x7 mode exclude
ff02::1:ff02:7b%vlan84 scopeid 0x7 mode exclude
ff02::2:a17e:3d85%vlan84 scopeid 0x7 mode exclude
ff02::2:ffa1:7e3d%vlan84 scopeid 0x7 mode exclude
```

Solicited-Node Multicast Address

The following addresses are called "Solicited-Node Multicast Address":

```
ff02::1:ff67:382f%vlan84 scopeid 0x7 mode exclude
ff02::1:ffe0:ae33%vlan84 scopeid 0x7 mode exclude
ff02::1:ff00:42%vlan84 scopeid 0x7 mode exclude
ff02::1:ff02:35%vlan84 scopeid 0x7 mode exclude
ff02::1:ff02:7b%vlan84 scopeid 0x7 mode exclude
```

A Solicited-Node Multicast Address is one with the prefix `ff02:0:0:0:0:1:ff00::/104`. This means it ranges from `ff02::1:ff00:0 to ff02::1:ffff:ffff`. These five addresses start with this prefix.

What is the purpose and how is the IID configured? The objective is NDP, Neighbor Discovery Protocol[9].

Neighbor Discovery Protocol

NDP is one of the core protocols of the IPv6 protocol suite and is responsible for the following functionalities seen in IPv4:

- ARP (L2-L3 address translation)
- ICMP Router Discovery[9]

and the following IPv6-specific features:

- DAD (Duplicate Address Detection)
- SLAAC (StateLess Address AutoConfiguration)

Before diving into the details, let's see the IID of the address format of a Solicited-Node Multicast Address. This address is generated from a unicast address. To understand the correspondence, compare the output of **`ifconfig`** and **`ifmcstat`** command:

```
inet6 2001:db8:fb5d:8001:7c36:33b7:b967:382f prefixlen 64 autoconf temporary
ff02::1:ff67:382f%vlan84 scopeid 0x7 mode exclude

inet6 2001:db8:fb5d:8001:a6ba:dbff:fee0:ae33 prefixlen 64 autoconf
ff02::1:ffe0:ae33%vlan84 scopeid 0x7 mode exclude

inet6 2001:db8:fb5d:8001::42 prefixlen 64
ff02::1:ff00:42%vlan84 scopeid 0x7 mode exclude

inet6 fe80::ffff:2:35%vlan84 prefixlen 64 scopeid 0x7
ff02::1:ff02:35%vlan84 scopeid 0x7 mode exclude

inet6 fe80::ffff:2:7b%vlan84 prefixlen 64 scopeid 0x7
ff02::1:ff02:7b%vlan84 scopeid 0x7 mode exclude
```

In short, the last three octets of a unicast address are used in the multicast address. For example, `ff02::1:ff67:382f%vlan84` has `0x67`, `0x38`, and `0x2f`. These three octets are at the last of the unicast address `2001:db8:fb5d:8001:7c36:33b7:b967:382f`. So, your box will have as many link-local multicast addresses as there are unicast addresses. While no multicast address appears in the output of **`ifconfig`** command, they are always automatically configured.

Address Resolution

Let's move on to how multicast addresses are used in NDP. One of the most crucial functionalities of NDP is L2-L3 address resolution. For IPv4, ARP[10] is responsible for this. The big difference is that ARP is a protocol of L2 network such as Ethernet, not IPv4. To communicate in IPv4, the source and destination L2 address are required. This address mapping information cannot be obtained using IPv4 due to a chicken-and-egg problem. In IPv6, using an LLA and a Solicited-Node Multicast Address, a host can initiate an IPv6 communication without knowing the destination address. The all-node address is always available.

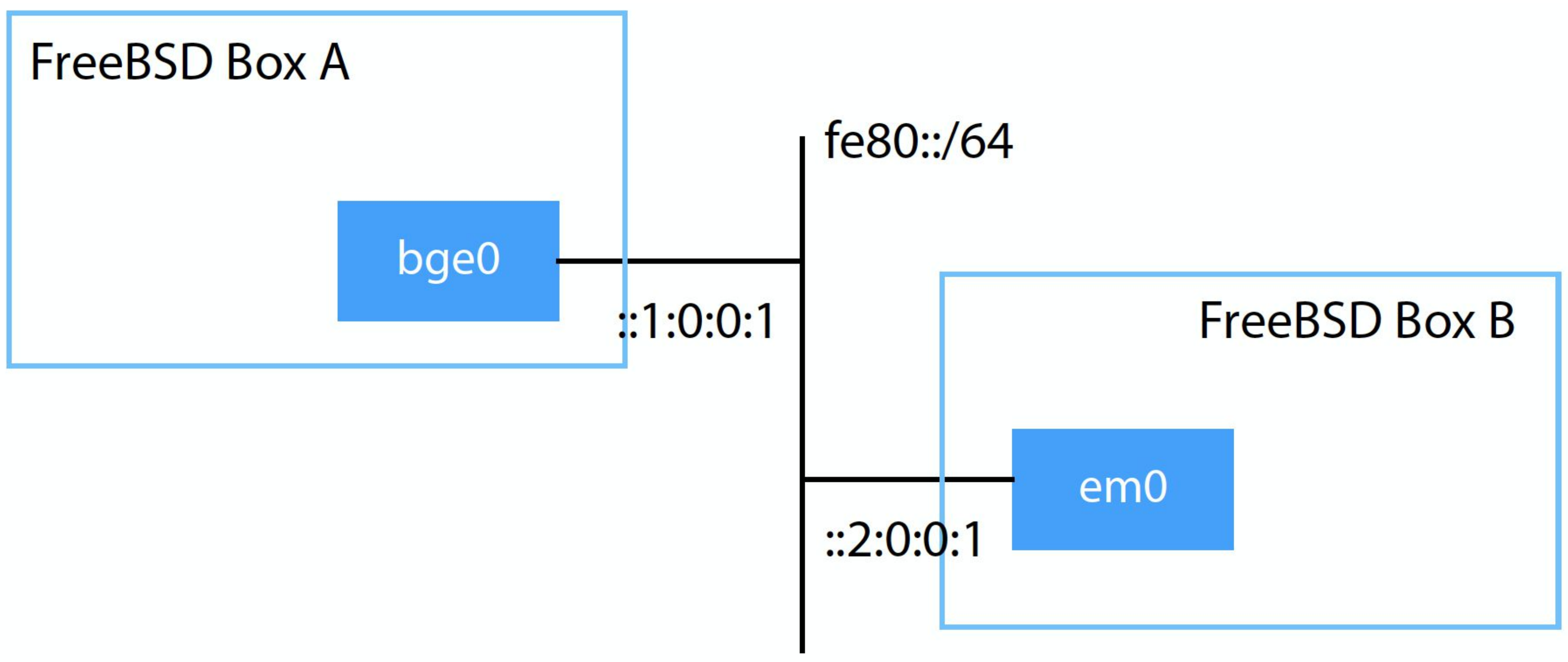


Figure 1: An example network with Box A and Box B

More specifically, the address resolution in NDP works in the following way. Figure 1 shows an example network. There are two machines, Box A and Box B.

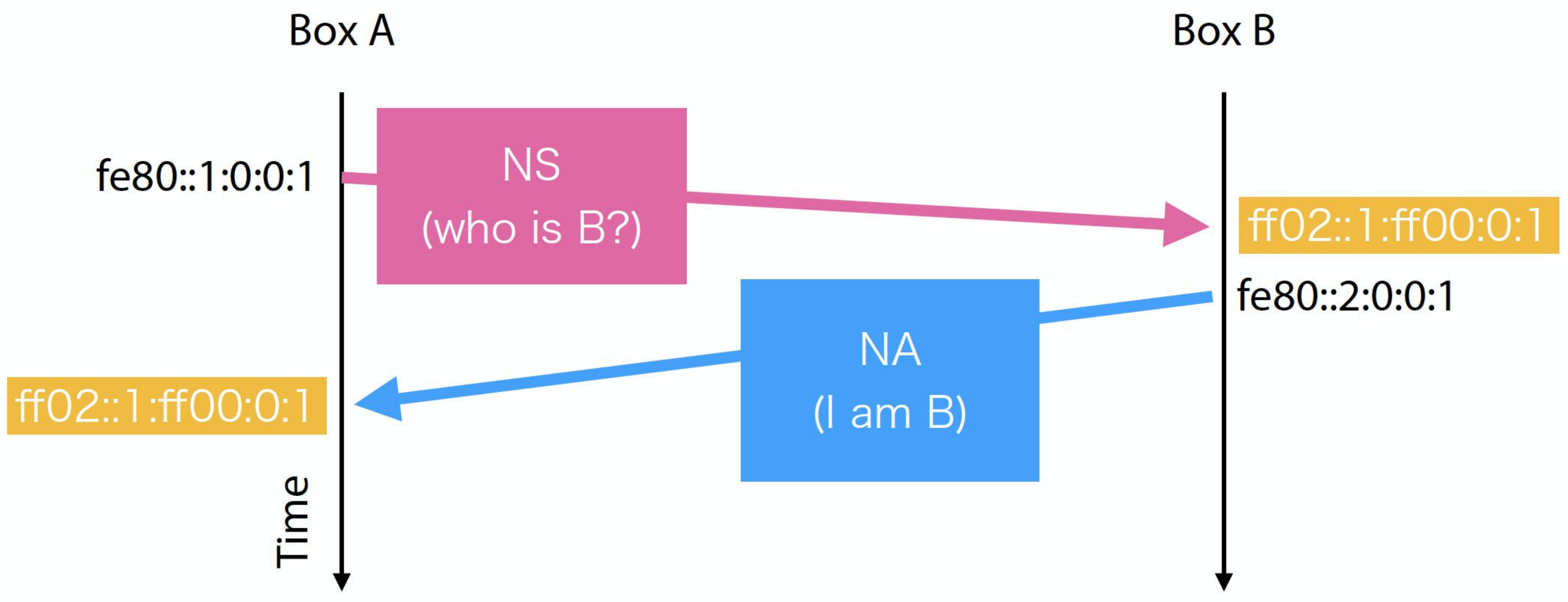


Figure 2: Timing diagram of Neighbor Solicitation and Neighbor Advertisement

When initiating a communication from A, it needs the L2 address of B. As shown in Figure 2, Box A sends a "Neighbor Solicitation" (NS) message of ICMPv6. It contains a pair of the LLA and the MAC address. The destination address of NS is Solicited-Node Multicast Address. The last three octets in the unicast addresses of A and B is `0:0:1`, so the address will be `ff02::1:ff00:0:1`. Box B will send back a "Neighbor Advertisement" (NA) message that contains another pair of the LLA and the MAC address on the Box B side. A knows B's LLA and MAC address in this exchange. Note that the Solicited-Node Multicast Address for A is the same as one for B by chance in this example. It depends on the IID of A and B.

Router Discovery and Autoconfiguration

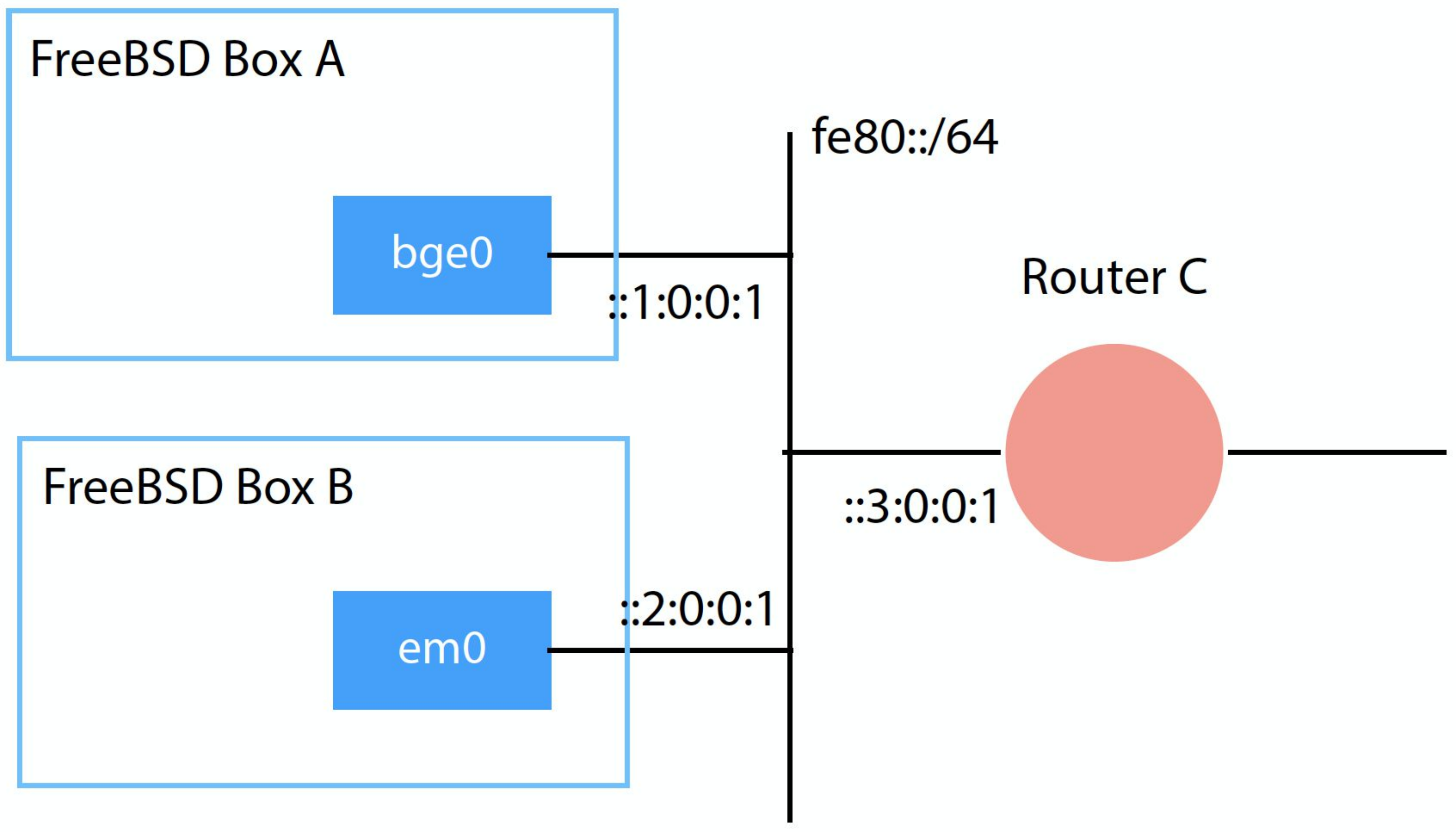


Figure 3: An example network with Box A, Box B, and Router C

Figure 3 shows another example network when there is a router. NS and NA messages are also exchanged for L2-L3 address resolution on this network. In addition, a host can discover the router's existence.

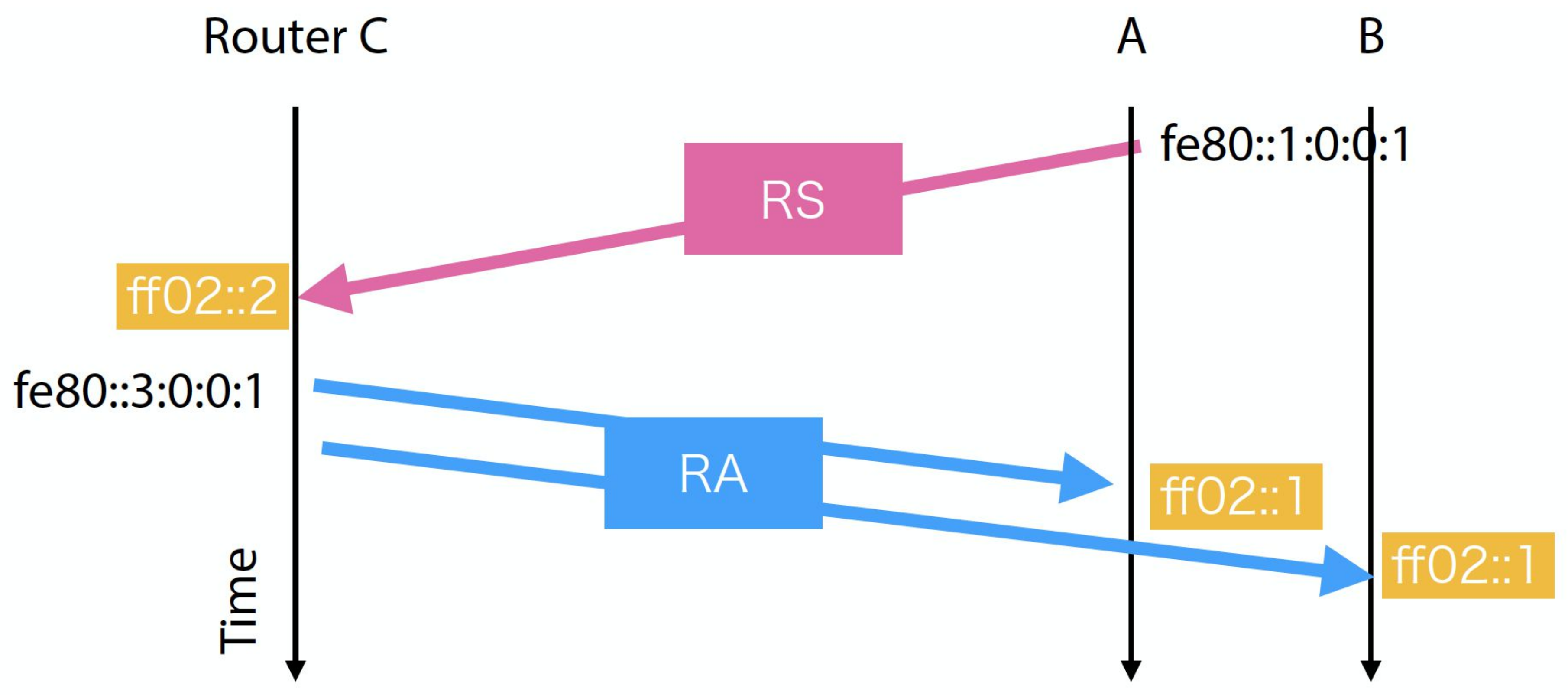


Figure 4: Timing diagram of Router Solicitation and Router Advertisement

A host can send a "Router Solicitation" (RS) message of ICMPv6 as shown in Figure 4. The destination address of RS is the all-routers multicast address. The connected routers will receive the RS and send back a "Router Advertisement" (RA) message. The destination address of RA is the all-nodes multicast address so that all hosts will receive it. An RA message contains network configuration parameters, such as MTU, subnet prefix, default router address, etc. The host nodes can configure themselves using the information. The default router address and the subnet prefix are sufficient to make the host ready to communicate with the IPv6 Internet.

As explained in the past columns, RA is sent by the `rtadvd(8)` daemon. RS can be sent by the `rtsol(8)` utility. The kernel handles NS and NA, so usually you do not need to be aware of them. Note that the kernel processes RAs only when the interface has the ACCEPT_RTADV flag. A router sends RAs periodically even if no RS is received, so you do not always need to run the rtsol(8) utility.

In this way, IPv6 uses various addresses for each specific purpose. Unlike IPv4, not all addresses are listed in the output of the `ifconfig` command. Multicast addresses can be shown using the `ifmcstat` command instead. In addition, the author wants to emphasize that an LLA on an interface is essential for NDP. Without an LLA, IPv6 does not work well. Actually, `IFDISABLED` flag, which is used to disable IPv6 communication on the interface, means disabling all of NDP traffic in the kernel. It blocks the NDP traffic only, but effectively disables IPv6.

Summary

This column has walked through how multiple addresses work in IPv6. You do not need to understand these kinds of details for communications using TCP or UDP. However, understanding of the configured addresses is quite important from the system administrator's perspective. You may want to configure additional access control lists or packet-filtering rules in some cases because an LLA on the box is another address where someone can access. On the other hand, blocking communications via LLAs may break NDP.

In the next column, several useful configuration tricks will be covered. They are based on the knowledge we know so far, and include missing parts such as DNS on IPv6 and DHCPv6.

Footnotes

[1] Global-scope Unicast Address. A routable address that consists of the prefix assigned from your ISP and the IID, interface identifier, which you have assigned or has been automatically assigned by SLAAC[2] or DHCPv6.

[2] Link-Local-scope Address. The prefix is always `fe80::/64`. This is unique only on the link and not routable.

[3] L3 stands for Layer 3 in the OSI reference model. This is a classic abstraction of communication protocols defined in `ISO/IEC 7498`. The TCP/IP protocol suite used for Internet does not fully follow this abstraction model. However, Layer 1, 2, and 3 are still helpful to understand the layer structure of Ethernet, IP, and UDP/TCP.

[4] This algorithm is explained in RFC 4291.

[5] An octet means 8-bit long data.

[6] Least Significant Bit. The lowest-order bit of a binary value.

[7] https://www.iana.org/assignments/ipv6-multicast-addresses/ipv6-multicast-addresses.xhtml

[8] This is defined in RFC 4861, "Neighbor Discovery for IP version 6 (IPv6)".

[9] RFC 1256, "ICMP Router Discovery Messages."

[10] RFC 826, "An Ethernet Address Resolution Protocol"

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 since 2007 has hosted AsiaBSDCon, an international conference on BSD-derived operating systems in Asia.

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

WeGet letters

by Michael W Lucas

Dear Last Worst Hope,

It's all too much. I've cleaned up the servers and dealt with the outages and stabilized the environment, but the boss keeps piling on more work and more work and there's no way to complete it all. I can't quit, but how to I manage all this?

—Overwhelmed

Dear Overwhelmed,

Helping you begins with rewriting your letter to put the blame where it belongs.

I have performed my duties with the bare minimum of competence, but I don't understand that the reward for work well-done is more work. My efforts to educate my manager about the amount of work required for further tasks have failed, either because my communication skills are inadequate or my manager honestly does not see how the amount of effort is relevant because he is a sociopath rocketing to the C-level. Probably the latter. I arranged my life with insufficient prescience or flexibility, and now I'm trapped. What should I do?

There. That's better.

The problem with computing professionals is that they think of themselves as "problem solvers." After all, they make these super complicated machines do complicated things, like add really big numbers together. Each processor core contains a billion trillion gazillion transistors, and you command them all! When this horribly complex machine implodes, you fix it! You don't work with software—you are a professional problem solver.

Problem solver. That's what the sociopath rocketing to the C-level wants you to think.

Computers are simple. The complexity of the most advance computer is wholly inadequate to grasp the intricacies of the simplest virus, and those pale next to the horrors of corporate politics. Computing professionals are skilled at solving problems in a tightly constrained environment where the Four Sacred Resources—Processor, I/O, Storage, and Memory—reign, but those skills cannot model the innumerable resources of reality.

To succeed in the outside world, you must accept the limitations of your skills and reject the constraint of being a *problem solver.* As long as you dream of yourself that way, you'll lose against the illogic of meatspace. You have metrics. You have measurements. You have all the data that says you're working hard. Abandon the labels others have slapped on you. Liberate yourself. Abandon solving problems in favor of *strategic failure*.

Strategic failure isn't about bringing the whole system down. Any sysadmin can do that. It's not even about timing, although timing is important. It's about choosing failures that will embarrass the right people at the right time, and being able to declare with a straight face, "I only had time to maintain one system, and I chose the mission-critical one."

Yes, your manager will be angry. So what? If you can't quit, they can't replace you. Everyone in a position like yours, in any organization, possesses a unique combination of skills ranging from the bizarre to the obscene, a brew which is entirely impossible to replicate in any other single person. They only way to develop those skills is to *be* you, and nobody will sign up for that. Don't be arrogant—after all, would anyone competent outside the tiny, specialized cell of computing let themselves be maneuvered into this situation? Showing anger and frustration will only get you sent to Human Resources for counseling about your attitude. A shrug and an indifferent "I allocated my resources in accordance with guidance from management" will serve you well. Use those words, *in accordance with guidance from management*, like roasted garlic. A bit sprinkled here and there will give your businesslike attitude credibility.

Yes, your manager will be angry. So what?

After two or three incidents of properly chosen, high visibility, irritating but non-devastating strategic failures, you'll wind up in meetings with your manager and assorted outsiders who want to know why you suck so much. Your manager would prefer to throw you out the nearest airlock, so ignore them. Concentrate on the others. Be calm. Present everyone but your manager with the documentation on how you work. Almost always, merely having the documentation will suffice. Outsiders won't ask too many questions, out of a well-reasoned fear that they might learn something about computers and thus be forcibly transferred to your department.

People will present solutions. You should also offer solutions. One of them should be your preference, the others, acceptable. If the company wants you to, say, stop aggregating syslog and netflow data and shut down those systems, that's fine. You can always answer trouble tickets with, "Management has declared that I cannot help you with this problem."

Eventually, they'll settle on hiring someone. As previously established, whoever they hire will not be qualified to help you or to manage your systems. Remember Sysadmin Rule #27: "Competent coworkers are not hired. They are forged. By you." If you offer to mentor a junior sysadmin and save the firm tens of thousands of dollars, you improve the odds of getting help. If you make that offer in front of other people, you improve your image in the company. Be sure to say that you have specific questions you want to ask applicants.

Talking to job seekers? Ugh. Yes, I know, it's painful, but you're going to have to talk to the survivor daily so you best discard everyone who'll be painful to work with. No, don't ask about binary trees or bubble sorts or any of that other garbage. You want to discard applicants as quickly as possible, so set up a puzzle. Now that technology has advanced and CRT monitors are no longer standard, I can finally share my Secret Helpdesk Hiring Puzzle.

I would bring the applicant to an isolated room lit with the worst fluorescent tube in the building. If I had a chance beforehand, I would establish mood by starting my CD of "Great Horror Movie Screams" at a nearly subliminal volume and lighting a sample of incense from the Despair Collection. The room contained a desk, a computer, and a CRT monitor with a twisted and distorted image. I would say "If you worked for me, how would you fix this? Talk me through it."

Simple, right?

Swapping out the monitor didn't work. Neither did swapping out the video card, or the whole computer. At that point, most applicants said they would ship the whole computer back to the manufacturer.

The people who realized that the problem involved the computer's location and shifted it two feet from the magnet I'd taped to the bottom of the desk? They got the job.

You need a puzzle like this, with modern technology and just a whiff of malice. Something even your boss can understand. The convenient thing about the *problem solver* label is that people outside the computing department also believe it. "I set up this typical problem I have to solve, and only these applicants could solve it," is instantly credible.

Yes, you'll spend time training your new flunky—but the reward for work well-done is more work. Plus, you'll be training them to handle the work you don't want to do, and you already know they have problem solving skills sufficient for the tightly constrained environment of computers. This will give you time to solve your real problem and rearrange your life to be more flexible.

Be careful practicing strategic failure, however. Do it too much, and you'll find yourself rocketed to the C-level.

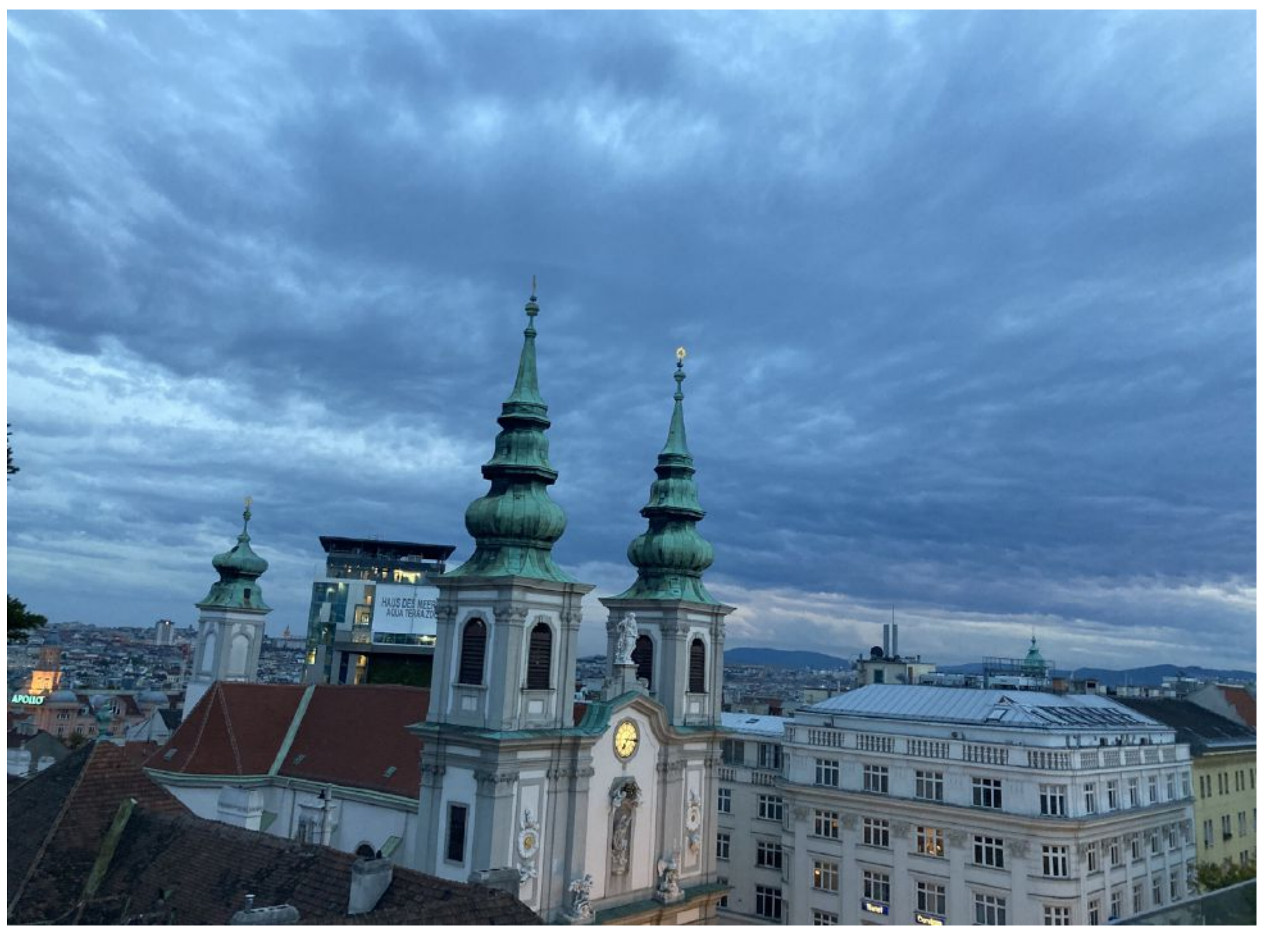
Have a question for Michael?
Send it to letters@freebsdjournal.org

MICHAEL W LUCAS is the author of *Absolute FreeBSD*, *$ git sync murder*, and fifty-odd other books. *Letters to ed(1)* collects his FreeBSD Journal columns. Learn more at https://mwl.io.

EuroBSDCon 2022

BY KYLE EVANS

In September, I visited the beautiful city of Vienna for EuroBSDCon 2022; many thanks to my employer, Klara, for covering enough of my travel expenses to make this trip possible. This was my second BSD conference (and certainly not my last), but it was definitely the more exciting of the two for a number of reasons. This particular trip was my first flight across an ocean, a full seven time zones east of home, and I brought my wife and toddler along with me. My last conference was BSDCan 2018, so I was quite excited to meet—in person—a lot of the folks I've worked with online over the years.



I visited the beautiful city of Vienna for EuroBSDCon 2022.

We arrived later in the day, one day before the dev summit began. Our trip was generally uneventful until we got stuck on the tarmac at our last leg, AMS, for an extra two or three hours. dch@ was kind enough to offer transport from the airport and gave us a brief tour of the city before dropping us off at Hotel Erzherzog Rainer. We were generally exhausted by the time we got there, so I was quite relieved that I forgot to RSVP for the casual core dinner that was scheduled for only an hour or two after we had arrived.

The first day was the FreeBSD developer summit and the associated group dinner. I met up with some other folks from Klara and hung out near the back of the room where the summit was held. At the back, I ran into Eirik Øverby from Modirum, who had some more Apple hardware for me to take home and add to the literal tower of Apple Silicon hardware for porting, and I also met one of the Apple engineers, Cosimo Cecchi, who came early and attended the devsummit. We listened to talks from the FreeBSD Foundation as well as presentations from various developers on the state of their work (Workflow issues, ALTQ, Netlink, CI). Lunch and coffee breaks scattered throughout the day offered a good hallway track for the early days.

There was a designated chunk of time for hacking groups, but—in all of the pre-travel chaos—I had apparently left my laptop charger at home, so I took advantage of the time and went for a walk with my family to pick up a USB-C charger before the devsummit dinner. The hosts of the dinner were kind enough to allow my wife and daughter to attend for which I was very grateful since I was effectively abandoning them half the time we were there. The other attendees were incredibly awesome with our young one, despite her being a bit cranky at times.

The second day of the developer summit was much like the first, with more talks and working groups along with more scheduled time for unstructured hacking. jhb@ spent ten minutes and solved an issue we had—for a lot longer than ten minutes—with PCI on Apple Silicon, which was simultaneously exciting and depressing. After the devsummit, my family met me outside of the TU building and we walked around a bit to explore the area.

Day one of EuroBSDCon started off with a very interesting keynote from Frank Karlitschek. Next, I attended Taylor R Campbell's talk on "How I learned to stop worrying and yank the USB", in which he discussed many of the interesting ways he broke and fixed USB hotplug in NetBSD, and how he fixed those issues in a pretty clean way.

I needed to catch up a branch or two in one of my local trees, so I wandered over to Brooks' session on how to add a system call in FreeBSD, since I had a decent amount of knowledge on the topic already. Despite this, it was still chock full of interesting tidbits about other ABIs and compatibility concerns.

For the final two talks I attended that day, I checked out Mateusz's presentation on measuring performance overhead of tracing and Allan's talk on scaling ZFS. I hadn't spent much time tracing in the years I've worked on operating systems, but I was still curious as to how dtrace and ebpf compared for the task, overhead-wise, in real-world scenarios. I wanted to attend Ken's talk on OpenBSD filesystem blocks, but I got caught up socializing in the hallway track instead.

For the final two talks I attended that day, I checked out Mateusz's presentation on measuring performance overhead of tracing and Allan's talk on scaling ZFS.

My wife and daughter met me outside again, and this time we hunted down some döner kebab I had been anxious to try. Disaster struck that night as our young one finally realized she was jet lagged and barely slept. On the final day, I rolled onto the locked campus at around 07:00 after about an hour of sleep (but not wanting to wake anyone else), and within 30 minutes or so a staff member inside the building (security, I think?) noticed that I was standing outside, patiently awaiting the conference start, and allowed me to enter.

I realized after staring at my laptop for a while that I wasn't likely going to comprehend much during the talks, so I admitted defeat and hung out in the lobby for the day, intermittently hacking on various things.

Despite the appearance that I was losing value by not attending talks on the final day, I feel that I actually gained a lot more from that decision. I ended up meeting a lot of not-yet-familiar faces I otherwise would not have if I had attended talks. I brought along the MacBook that Eirik was lending me for the porting cause along, and a couple of us in the hallway battled with the laptop to get the Norwegian keyboard remapped in software to a layout that I was more familiar with. macOS' keyboard mapping does 98% of the job, but it doesn't remap what's easily one of my top-five keys used: tilde/backtick. If you stumble across this for similar reasons, the answer is to use `hidutil` to finish the job and get your tilde back.

As the conference wrapped up and we bid one another farewell, I managed to get a list of family-friendly things to do with our remaining three days in Vienna from krion@, whom I had met in-person back in Allan's talk about scaling ZFS. The list was, indeed, full of great suggestions, though sadly we didn't get to all of them as the weather didn't quite cooperate.

My general advice for conferences a longer distance away from "home" is what Allan Jude had tried to tell me: book your flight a bit earlier to give yourself a day or two before the conference to try and get your sleep schedule somewhat normalized. It's hard not to recommend also booking a couple of days after a conference for tourist activities in case you end up learning of more exciting sights to see from conference attendees.

KYLE EVANS is a FreeBSD developer currently employed by Klara, Inc. He has been a part of the FreeBSD project since 2017 working on a wide variety of projects in base.

Prometheus Installation & Setup

BY BENEDICT REUSCHLING

For a long time, monitoring systems and services running on them has been done as part of a sysadmin's job. This serves many purposes:

- is the system generally reachable (Availability),
- answers the question of what did the system do last week on Sunday at 4 a.m.? (Metrics)
- alerts IT personnel (often at ungodly hours) about unusually high processes and other events out of the ordinary (Alerting)

Sysadmins typically collect these metrics at a central location for further study and visualization, which helps more than logging into individual systems and running `tail -F /var/log/messages` or other logfiles. Finding when a problem started by seeing a spike in CPU usage or a dramatic decline in available disk space at the beginning of the month is clearly visible from a graph. When alerts are configured, notifications are sent about certain events (is the system reachable at all?) or if certain thresholds are reached (only 10% free disk space left). All of these have traditionally been done by software such as Munin, CheckMK, Nagios or Zabbix among others.

Prometheus is a fairly young monitoring project in the open source space. It did become a well-established solution—mainly in the Kubernetes and Cloud space—but is also usable in other environments. The setup was surprisingly easy for me, after having used a combination of telegraf, InfluxDB, and Grafana for a long time. Grafana is also used here as well for the visualization of Dashboards. InfluxDB, as the name suggests, serves as the central storage place for the collected metrics from which Grafana pulls the data. Sending the data was done by Telegraf running on each machine, sending its metrics to InfluxDB at regular intervals (i.e., every 10 seconds).

Prometheus is a fairly young monitoring project in the open source space.

Prometheus' architecture is similar: a central Prometheus server for transforming, storing, and streamlining the received data, with so-called `node_exporters` collecting the metrics on client machines. Again, Grafana uses the Prometheus data and can display them with some ready-made dashboards to impress your colleagues and be useful at the same time.

Other components include an alert manager to send various configurable notification types (email, SMS, pager, chat messages) when certain events occur. To query the data, Prometheus offers its own query language called PromQL that Grafana understands and uses for the dashboard content. Users can also write their own ad-hoc queries using PromQL, allowing for quick searches without having to build a dashboard first.

The logic by which metrics are extracted, formatted, and sent is coded into the exporters. There are several different exporters available for specific software like databases. Applications like RabbitMQ, GitLab, and Grafana itself allow the export of their own application states into a Prometheus-compatible format for monitoring. Written in Go, Prometheus is highly scalable and does not need too many resources when running.

In this article, we'll setup a FreeBSD based Prometheus server and have clients (Linux and FreeBSD) send system metrics to it via the **`node_exporter`**. We'll also use Grafana to visualize the data in an existing dashboard that we are importing for this purpose.

Users can also write their own ad-hoc queries using PromQL.

Prometheus Setup

First, we setup the Prometheus instance on a FreeBSD system. Freshly installed and connected to the network, we begin by creating the dataset where Prometheus stores its data in **`/var/db/prometheus`**. A directory is created by the port automatically, so this step is not strictly necessary. However, running it on ZFS as a separate dataset with properties like compression is good practice.

```
# zfs create -o compression=zstd sys/var/db/prometheus
# pkg install prometheus node_exporter
```

To extract system level metrics from the host, we install the **`node_exporter`** on our Prometheus host and all other machines we want to monitor. The Prometheus port installs a default configuration file called **`prometheus.yml`** in the **`/usr/local/etc`** path. We're going to modify it to fit our needs. The syntax for the configuration file is done in YAML, so be extra careful to avoid tabs and use the proper indentation with spaces.

```
prometheus.yml:
# my global config
global:
  scrape_interval: 15s # Set the scrape interval to every 15 seconds. Default is ev-
ery 1 minute.
  evaluation_interval: 15s # Evaluate rules every 15 seconds. The default is every 1
minute.
```

```
  # scrape_timeout is set to the global default (10s).

# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries scraped from
this config.
  - job_name: "prometheus"
    static_configs:
     - targets:
        - mistwood:9090

  - job_name: bdc
    static_configs:
     - targets:
        - mistwood:9100

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.
```

Before diving into the configuration bits, we enable the `prometheus service` and the `node_exporter` on FreeBSD to start upon boot.

```
# service prometheus enable
# service node_exporter enable
```

Prometheus provides a web-based interface for querying and displaying the metrics exported by the systems (called targets in Prometheus lingo). I provide an extra argument to the start of the Prometheus service to define at which port the web interface should be reachable on my host called mistwood.

```
# sysrc prometheus_args="--web.listen-address=mistwood:9090"
```

Once we have that, we can start the `prometheus service` and the `node_exporter` like this:

```
# service prometheus start
# service node_exporter start
```

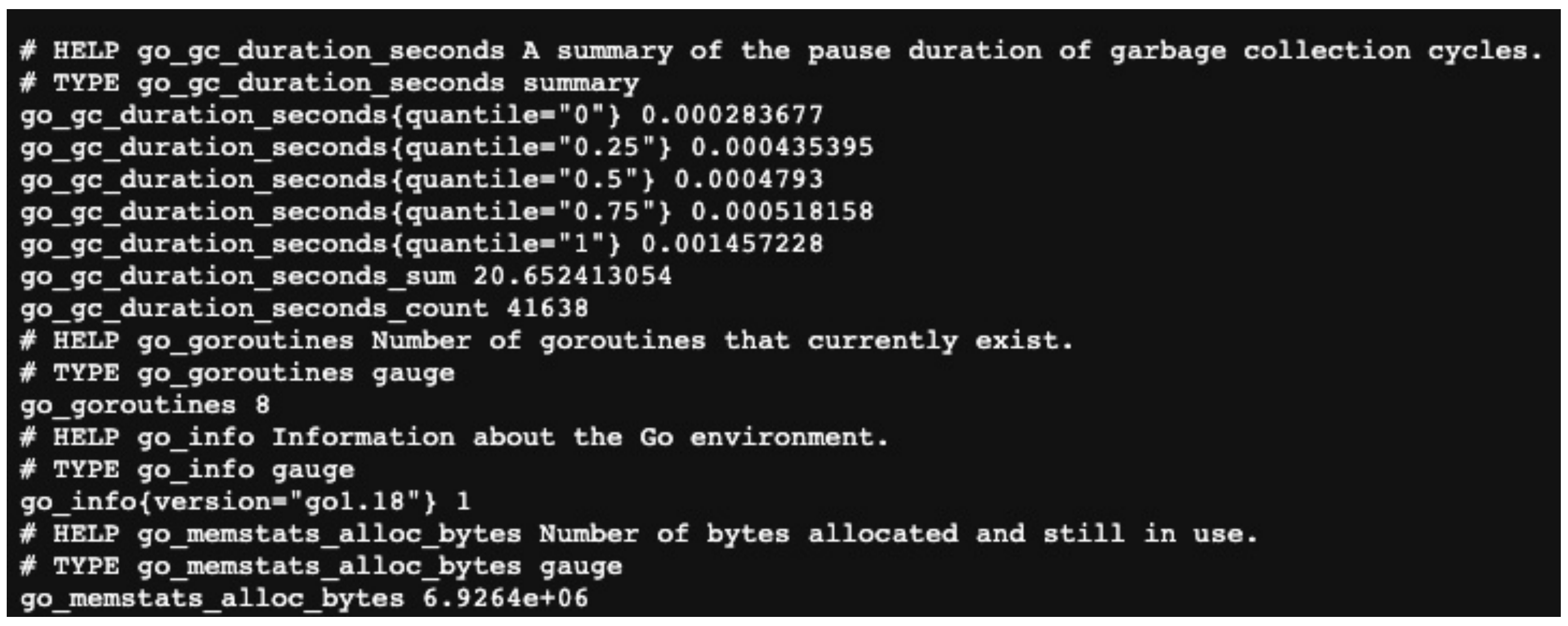
After a few seconds, the output of

```
# sockstat -l
```

should have the following lines in it, confirming both services started successfully:

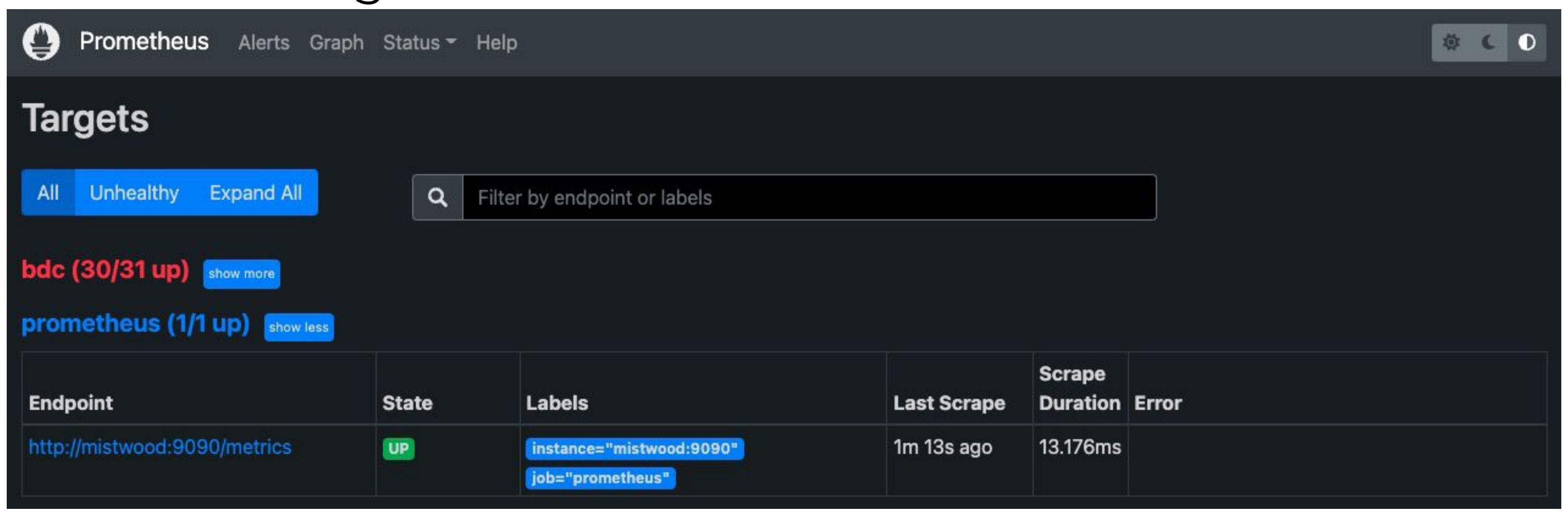
User Address	Command	PID	FD	PROTO	Local Address	Foreign
prometheus	prometheus	70027	8	tcp4	mistwood:9090	*:*
nobody	node_exporter	2950	3	tcp46	*:9100	*:*

First, let's see if the `node_exporter` is extracting some metrics. We can do that by pointing our browser to the URL of the host running the `node_exporter` service (mistwood in my case), adding the port 9100 and /metrics to the end to form this URL: http://mistwood:9100/metrics



```
# HELP go_gc_duration_seconds A summary of the pause duration of garbage collection cycles.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 0.000283677
go_gc_duration_seconds{quantile="0.25"} 0.000435395
go_gc_duration_seconds{quantile="0.5"} 0.0004793
go_gc_duration_seconds{quantile="0.75"} 0.000518158
go_gc_duration_seconds{quantile="1"} 0.001457228
go_gc_duration_seconds_sum 20.652413054
go_gc_duration_seconds_count 41638
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 8
# HELP go_info Information about the Go environment.
# TYPE go_info gauge
go_info{version="go1.18"} 1
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
go_memstats_alloc_bytes 6.9264e+06
```

You can see a list of exported metrics in a namespace separated by underscores. Refresh this page every 10 seconds and you'll see updated data collected by the `node_exporter`. Since we're already in the browser, we can also check the status of Prometheus. The URL is very similar but using the port 9090 (no `/metrics` at the end) to get to the Prometheus web interface. Go to the Status pulldown and select Targets to see all configured hosts from the `prometheus.yml` above. In this example setup, I have extracted pieces of configuration for our big data cluster (bdc). Of course, you can pick your own labels instead of "bdc" and the "mistwood" host is also exchangeable.



bdc (30/31 up)

prometheus (1/1 up)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://mistwood:9090/metrics	UP	instance="mistwood:9090" job="prometheus"	1m 13s ago	13.176ms	

Prometheus checks if the hosts are reachable. The `node_exporter` that we added to the `prometheus.yml` file with port 9100 is also listed here and can be reached from here by clicking on the URL as well. Prometheus also checks the availability of the host, indicated by the UP in the State column of the endpoint.

Tags can be assigned to a certain host group to logically group them together. All host metrics collected receive this tag and can be filtered later using the PromQL language or directly within Grafana. My job name here is called bdc and all the machines that belong to that group are listed under targets. (I abbreviated it here to have only one FreeBSD and one Linux host in it.)

Before we dive into visualizations with Grafana, we can also create simple graphs from the Prometheus web interface by going to the Graph tab. At the top, there is an input field. On the left the blue Execute button, there is smaller one called the metrics explorer. Click on it and a search field opens, containing all the names of the metrics collected so far. Pick the one that you want to see. After selecting the metric, click the Graph tab next to Table to see a visualization of the metric over all your hosts. Click and select a portion to zoom into that timeframe or use the controls above to zoom out. This is already good to get a quick overview, but may not be as visually appealing as a full-blown dashboard. We add Grafana to the mix as it has more capabilities and different ways to display the data in various forms.

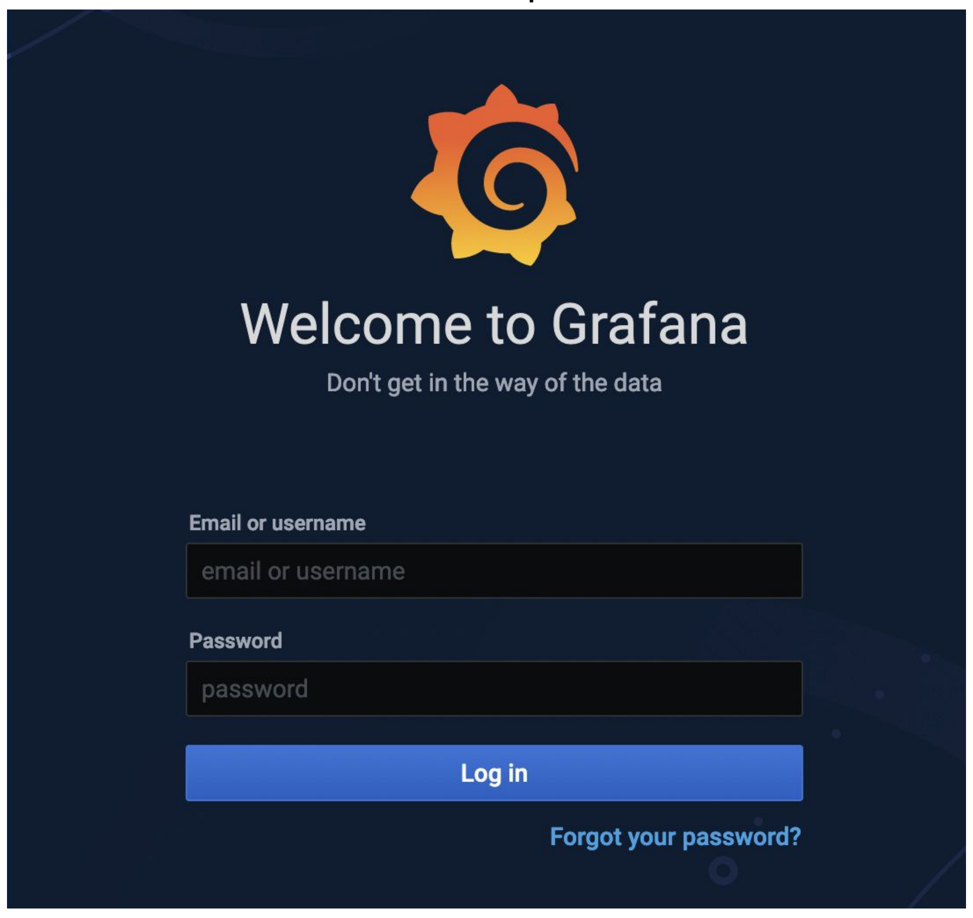
At the time of writing this article, Grafana 8 is the current version. Older versions work just as well, so there is no need to always chase the latest version to get pretty pictures.

```
# pkg install grafana8
```

Like before, we activate the `grafana service` to run upon boot and start it right away with the following two lines:

```
# service grafana enable
# service grafana start
```

Grafana does have a configuration file under `/usr/local/etc`, but we do not need to modify it here. Make sure to visit and read the documentation on the Grafana homepage to change the file for your environment. Wait a little for Grafana to start (check the `sockstat` output for a Grafana line listening on port 3000 by default). Browse to the login page for Grafana on the host that you installed it on with port 3000.

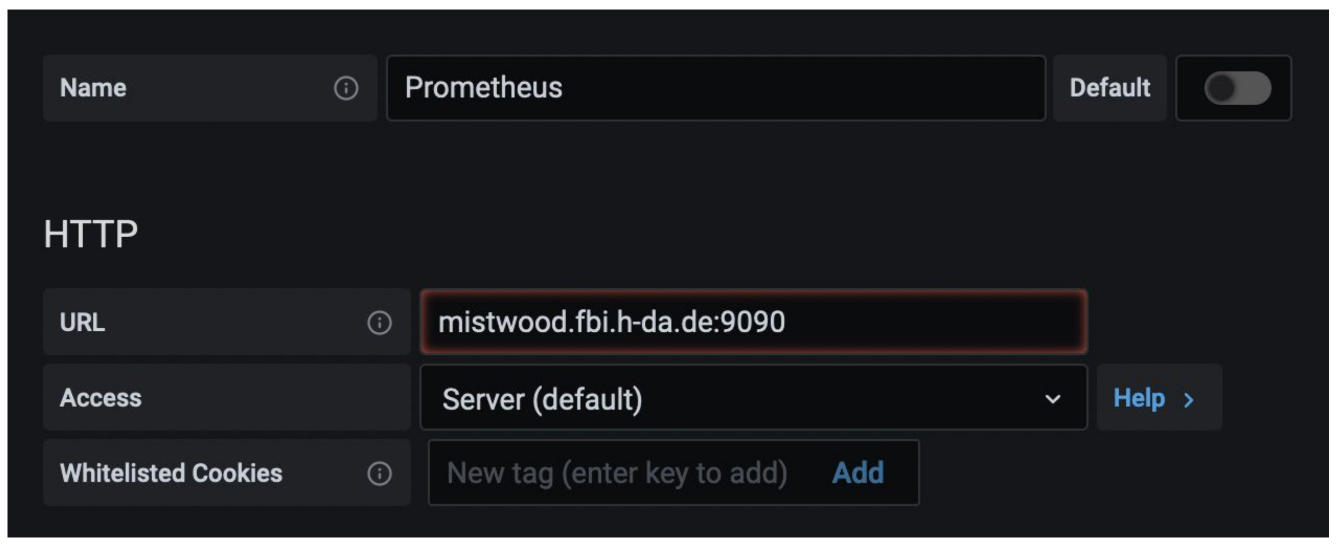

Welcome to Grafana

Don't get in the way of the data

Email or username: email or username

Password: password

On a fresh installation, Grafana has a default user **admin** with password **admin** that needs to be changed right after the first login to something else. You can also add more users with different privileges to see only certain dashboards, but right now we need to connect to our Prometheus metrics first. This is done by adding a data source under the gear icon on the left (Configuration -> Data Sources). Click the blue button on the left labeled "Add data source".


Name: Prometheus

Default

HTTP

URL: mistwood.fbi.h-da.de:9090

Access: Server (default)

Whitelisted Cookies: New tag (enter key to add)

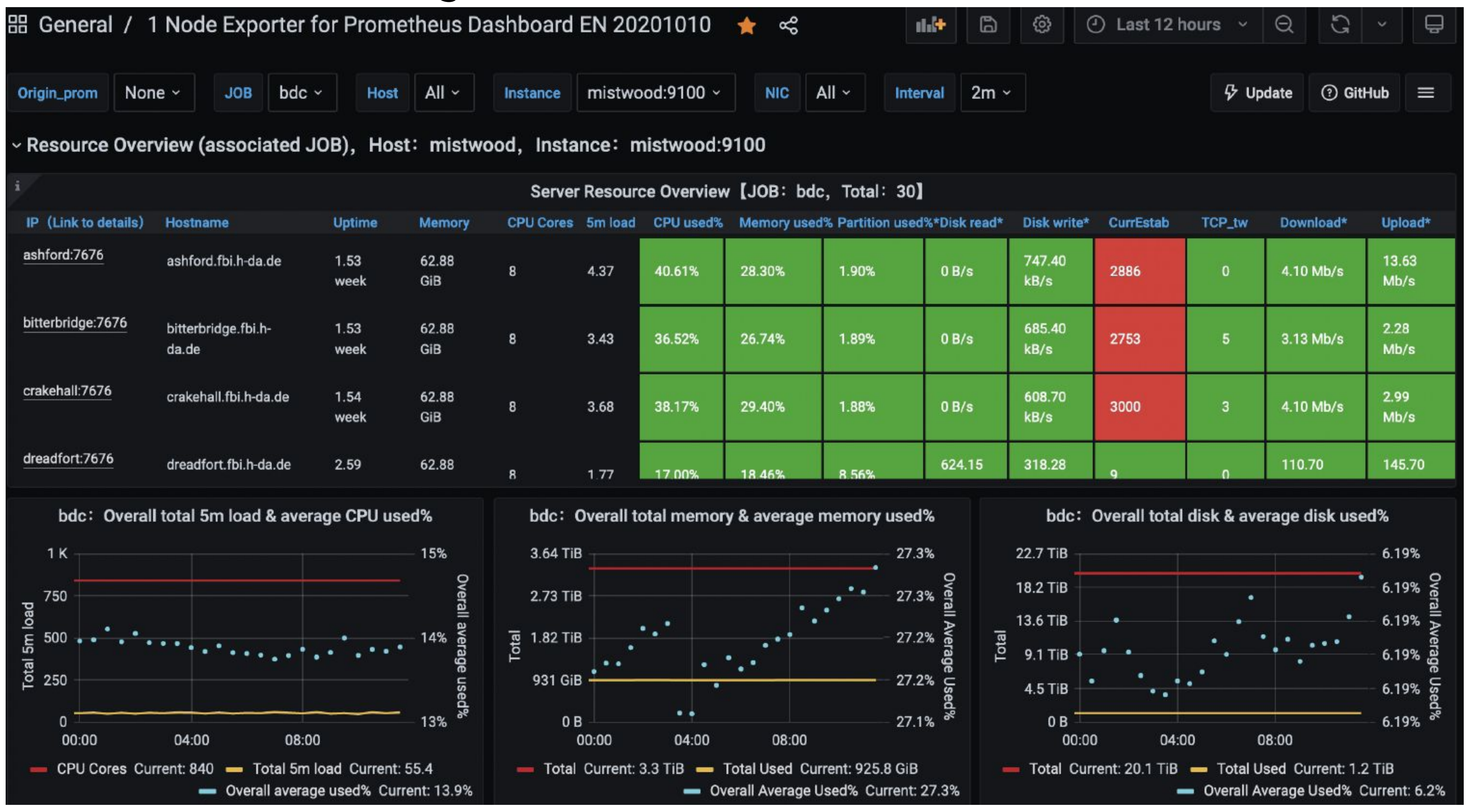
We give our datasource a descriptive name and provide the URL that we used earlier to access the Prometheus web UI on port 9090. At the bottom, click the "Save & Test" button to check if Grafana can reach your data source. Note: The configuration provided here is the most basic, which means it is focused on functionality and less on security. In production environments, you definitely need to have authentication and encryption of your metrics to not give attackers a clue about your infrastructure by reading the metrics. The Prometheus and Grafana webpages both provide documentation on how to do so.

Now that we have a datasource, we want to visualize the data coming from it. We can design our own dashboards, but my artistic talents go only so far. Other people have put in time and talent to create beautiful dashboards and provided them for everyone to use at the Grafana website. You can find them on https://grafana.com/grafana/dashboards/ along with filters on the left side to only show dashboards based on Prometheus data sources. Filtering further in the Collector Types pull-down to have Node exporter, the right side of the page automatically updates based on your filter criteria. Click on one of the search results to see a preview as well as additional information about it. On the right side, copy the dashboard ID to the Clipboard and change back to your Grafana browser tab.

Go to the dashboards tab on the left and select "Manage". On the left, there is a button labelled "Import". When clicked, you're brought to a screen that lets you paste the dashboard ID you selected earlier and load the dashboard. It's easy and convenient. Assign the data source created earlier and finish the import. For your convenience, here are a couple of dashboard IDs that I use (the last one is even built for FreeBSD use):

- 1860
- 11074
- 4260

You can find the dashboards listed on the Dashboards tab on the left and get to them by clicking their name. Some have filters at the top to pick a single host to display or select other criteria like network card or storage media.


General / 1 Node Exporter for Prometheus Dashboard EN 20201010
Last 12 hours
Origin_prom None
JOB bdc
Host All
Instance mistwood:9100
NIC All
Interval 2m
Update
GitHub
bdc: Overall total 5m load & average CPU used%
Total 5m load
Overall average used%
CPU Cores Current: 840
Total 5m load Current: 55.4
Overall average used% Current: 13.9%
bdc: Overall total memory & average memory used%
Total
Overall Average Used%
Total Current: 3.3 TiB
Total Used Current: 925.8 GiB
Overall Average Used% Current: 27.3%
bdc: Overall total disk & average disk used%
Total
Overall Average Used%
Total Current: 20.1 TiB
Total Used Current: 1.2 TiB
Overall Average Used% Current: 6.2%

Resource Overview (associated JOB), Host: mistwood, Instance: mistwood:9100

Server Resource Overview【JOB: bdc, Total: 30】

IP (Link to details)	Hostname	Uptime	Memory	CPU Cores	5m load	CPU used%	Memory used%	Partition used%*	Disk read*	Disk write*	CurrEstab	TCP_tw	Download*	Upload*
ashford:7676	ashford.fbi.h-da.de	1.53 week	62.88 GiB	8	4.37	40.61%	28.30%	1.90%	0 B/s	747.40 kB/s	2886	0	4.10 Mb/s	13.63 Mb/s
bitterbridge:7676	bitterbridge.fbi.h-da.de	1.53 week	62.88 GiB	8	3.43	36.52%	26.74%	1.89%	0 B/s	685.40 kB/s	2753	5	3.13 Mb/s	2.28 Mb/s
crakehall:7676	crakehall.fbi.h-da.de	1.54 week	62.88 GiB	8	3.68	38.17%	29.40%	1.88%	0 B/s	608.70 kB/s	3000	3	4.10 Mb/s	2.99 Mb/s
dreadfort:7676	dreadfort.fbi.h-da.de	2.59	62.88	8	1.77	17.00%	18.46%	8.56%	624.15	318.28	9	0	110.70	145.70

Some of the dashboards can be a bit overwhelming in the amount of data they display at once. I find that an overview dashboard showing me all machines is a good start to see what is going on. When I identify something out of the ordinary, I drill down into that host with another dashboard that shows me that machine in more detail. Especially the long term trends that Prometheus provides this way give me a good understanding of whether a certain spike in memory usage is expected and normal.

For each host that should be monitored, install and start the `node_exporter` on it. On the Prometheus host, add the URL to the targets under the `static_configs` in `prometheus.yml` and then restart the prometheus service. That's fairly straightforward and is easily automated for a large number of hosts using configuration management tools like Ansible and others. Try out other `node_exporters` available on FreeBSD and find a good dashboard (or create one yourself) that fits your monitoring needs. I find that Prometheus can show me a lot more metrics than my previous setup. Alerting about certain events is also possible and there are packages available on FreeBSD to do so. I'll leave that as a learning exercise for you.

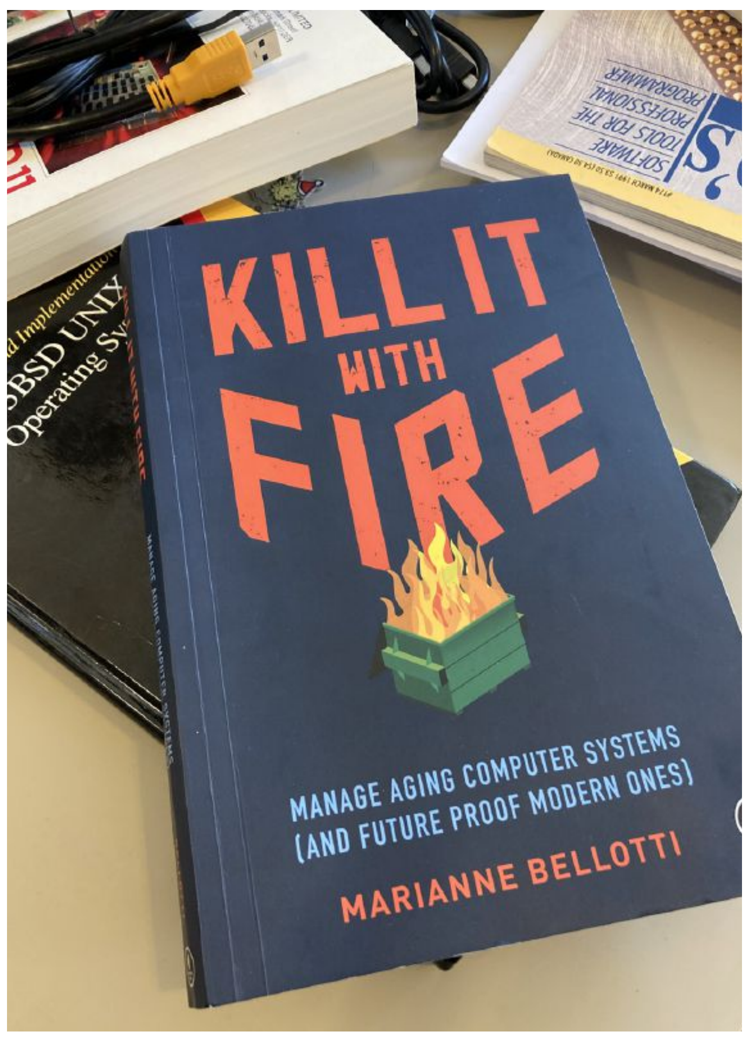
Prometheus is straightforward to set up and extend with more hosts to monitor. It's time for you to steal a little bit of fire from the gods to get better insight into the dark depths of your hosts and services.

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.

BOOK REVIEW

Kill It with Fire: Manage Aging Computer Systems (and Future Proof Modern Ones) by Marianne Bellotti

BOOK REVIEW BY TOM JONES


KILL IT WITH FIRE
MANAGE AGING COMPUTER SYSTEMS (AND FUTURE PROOF MODERN ONES)
MARIANNE BELLOTTI

FreeBSD is a legacy system. Wait! Before you start finding things to throw, it is worth sitting and thinking about how we use this term. Legacy systems are the things we are stuck with, many of the things we are stuck with are old, creaky and have seen much better days. Legacy comes with a notion of neglect. Legacy systems are painful to use. I know I have a bias here, but I don't think FreeBSD is painful to use. But it does have one heck of a legacy. FreeBSD is quickly coming up on its 30th year as an established open source project. Our roots extend much further back into time, UNIX topped 50 a couple years ago and BSD UNIX is going to do so, too.

Throw a stone in any direction (not at me!) and you are going to hit a tool or subsystem that feels like it has been dredged out of the past. Large chunks of FreeBSD are like sharks or crocodiles, perfected long ago and mostly forgotten by evolution.

On the other hand, throw another one of your stones (again, not at me!) and you'll hit something in FreeBSD which is painfully modern and cutting-edge. While this seems counter to my initial claim, I still think that FreeBSD belongs in the legacy system crowd.

Kill it with Fire (KIWF) by Marianne Bellotti, takes the same stance as I do on FreeBSD (or maybe it helped me get to this position). It begins with an important point about legacy systems, they are almost by definition, incredibly successful. Many software projects struggle to see the light of day at all, but legacy projects not only make it into continued production, but they also become integral components of much larger systems. So much so, that something about their age causes them to become a liability.

FreeBSD is certainly a massive success--we haven't managed to take over the entire world, but we are still part of the conversation about modern operating systems. For many of the engineers that use our platform, it has been a key component of their ability to have outsized success.

But you really can't argue that FreeBSD isn't a legacy system, and it is a legacy in two ways. First, it has components that are aging and a little long in the tooth. Second, FreeBSD, being almost too easy to maintain, becomes a legacy component in other systems.

KIWF doesn't aim to be a technical manual to help you maintain a legacy code base, for that it points you to the renowned Working Effectively with Legacy Code by Michael C. Feathers. Instead, *KIWF* talks about the political and social mechanisms that lead systems to be considered legacy and the approaches we need to take within an organization to help move on from them or to solve the major hindrances they generate.

Many organizations that use FreeBSD as a component see our favorite OS as a legacy component. It is notably more difficult to hire engineers to work on FreeBSD systems--the supply side is much smaller than what is available for Linux or Windows platforms. FreeBSD also has a marketing problem. We may be loved by the engineers that use our platform, but there are few startups using FreeBSD as a core selling point.

Both are political issues for the continued use of FreeBSD. It is easy to sell someone on a new idea to replace a painful product as new is without bounds and limits and we all know the real pain of the tools we use day to day.

Many organizations that use FreeBSD as a component see our favorite OS as a legacy component.

KIWF advocates for understanding why a system has succeeded enough for it to be considered legacy and provides some tools for approaching a design for new systems or refactoring to improve the existing components.

KIWF carries the subtitle: *Manage Aging Computer Systems (and Future Proof Modern Ones).* The lessons from this book carry forward and should help you build protections into systems so they can succeed more easily and be more manageable in the long run when they do.

KIWF is not a tutorial. While the introduction promises exercises, they can be a little thin. They do serve well as a starting point for helping your thinking about existing systems and your understanding of how to manage modernization projects. The advice here can help bridge the gap to the humans behind the maintenance and management of existing systems. There is advice for breaking down problems with teams, working through modernization projects, and keeping motivation high when replacing a component or updating it to match better with more modern practices.

KIWF is a great read if you have to defend the continued use of FreeBSD or another similar component in your environment. The measurement practices suggested by Marianne can be used to get quantified information about failure rates and usage, great tools for understanding how things are used, and evaluating any moves to other platforms.

I found a lot of value in this book; it has helped with my thinking about FreeBSD and how our OS is used in real environments. If FreeBSD is a part of your environment, or if you want to move from something else, then *Kill it With Fire* is a quick read that will help you have harder data to work from.

TOM JONES, FreeBSD Developer and co-host of the BSDNow Podcast, 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 April 2023

BY ANNE DICKISON

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

FOSDEM 2023

February 4-5, 2023
Brussels, Belgium
https://fosdem.org/2023/

FOSDEM is a two-day event organized by volunteers to promote the widespread use of free and open source software. The event offers open source and free software developers a place to meet, share ideas and collaborate. Renowned for being highly developer-oriented, FOSDEM brings together some 8000+ developers from all over the world.

SCALE 20X

March 9-12, 2023
Pasadena, CA
https://www.socallinuxexpo.org/blog/scale-20x

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

AsiaBSDCon 2023

March 30-April 2, 2023
Tokyo, Japan
https://2023.asiabsdcon.org/

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

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.