Design and Implementation of Bhyve Management Daemon

Yuichiro Naito *
SOUM Corporation †

Abstract

This paper introduces the Bhyve Management Daemon (hereafter BMD), which manages bhyve[1] virtual machines according to configuration files. BMD is not a replacement for existing bhyve management tools[5]. The design focuses on lightweight, simple configuration and secure operations for unprivileged users.

1 INTRODUCTION

Bhyve is a powerful hypervisor for FreeBSD and has been widely used for many years. Bhyve supports popular operating systems that run on amd64 or i386 architectures. Recently, ARM64 and RISC-V have been included.

The UNIX-like guest OS can be booted headless and all operations can be performed on the terminal, as long as there is a serial port and no expensive VGA console is required. Of course, bhyve can offer a VGA console for GUI operating systems like Windows. We can choose the operation style for CUI or GUI.

The design of BMD focuses on CUI operations, the traditional BSD daemon management style, such as editing configuration files and starting the daemon and then the daemon starts up whole virtual machines.

Basically, the UEFI boot loader requires a VGA console. Modern UEFI firmware supports headless boot, but some operating systems require setting loader options. If we want to boot with a serial console without VGA, we need to choose *bhyveload*[2] or *grub-bhyve*[6] boot loader. If the FreeBSD boot loader finds that it booted on bhyve, it will use a serial port as the console. The combination with *bhyveload* and headless boot works well. For NetBSD and OpenBSD, we have to use *grub-bhyve*, which requires the boot command *knetbsd* or *kopenbsd* and must pass the root device, the kernel filepath and the serial port. BMD supports locating the NetBSD and OpenBSD kernels in the virtual machine's disk and ISO images.

There are many uses of virtual machines, including renting computer resources to a user. A user can run a virtual machine with permitted CPU, memory and disk resources. For example, if a host machine has ample resources and is shared by several users and if users own their virtual machines and operate them individually, granting root privilege to boot a virtual machine is too much of a right. Asking the system administrator to boot a virtual machine will be annoying. BMD allows the virtual machine's owner to boot the virtual machine and also connect to the console.

All virtual machines have mandatory configurations, such as the number of CPUs and memory size. We often copy them from a template and reuse them. So, the template feature should be supported as configuration syntax. The common configurations are written in a template and a virtual machine configuration will include it. It simplifies configuration and makes it easy to manage future updates to filepaths or network connections.

However, each virtual machine must have unique devices, such as tap interfaces, nmdm devices and a VGA port number. The common template will assign the same device name. The template feature doesn't solve this issue. BMD manages the auto-assignment of these devices. The configuration should specify a bridge name to which a tap interface is added, rather than the tap interface name. BMD looks up the unused nmdm device name and assigns it. Users can know the device name from BMD. The VGA port can be calculated from a unique ID number for each virtual machine and also distributed via mDNS.

BMD also supports a plugin interface to extend the handling of each virtual machine. Users can invoke commands at prestart, poststop and during virtual machine state changes through the plugin interface and experimental support is available to change the hypervisor to QEMU.

2 DESIGN

2.1 configuration syntax

The basic syntax is based on *jail.conf(5)*. Because it is in the FreeBSD base system and well-known to FreeBSD users. The parser implementation is already in the *jail(8)* source so that we can refer to it. The key feature of the configuration syntax is its support for templates. So, the configuration syntax must support at least 2 sections: VM configuration and template. Variables are also helpful for reusing values in the configuration, such as the prefix for disk image paths and the VM's name. Variables are classified into 2 scopes: global and per-VM. The disk image path prefix will be used for all VM configurations. So it should be in the global scope. The VM name variable is used only for

*naito.yuichiro@gmail.com
†https://www.soum.co.jp

the individual VMs; it should be the VM scope. To define global variables, add a global section. To summarize, the configuration syntax has 3 section types: global, VM and template.

Listing 1 shows the configuration examples.

Listing 1: sample config

```
global {
  cmd_socket_mode = 0666;
  $defaults = common;
  $imgpath = /dev/zvol/zpool/images;
  $isopath = /zpool/iso;
}

template common(ncpu = 2, mem = 4G) {
  boot = no;
  ncpu = $ncpu;
  memory = $mem;
  x2apic = no;
  virt_random = yes;
  comport = auto;
  network = bridge0;
  stop_timeout = 60;
  loader_timeout = 180;
  err_logfile = /var/log/${NAME}.log;
  .apply default_disk;
}

template default_disk {
  disk = ${imgpath}/${NAME};
}

template graphics(pw = password) {
  graphics = yes;
  graphics_port = $((5900 + ${ID}));
  graphics_listen = 0.0.0.0;
  graphics_res = "1024x768";
  graphics_vga = "io";
  graphics_wait = yes;
  graphics_password = ${pw};
  xhci_mouse = yes;
  keymap = jp;
}

vm freebsd {
  .apply $defaults(2, 2G);
  iso = ${isopath}/\
  FreeBSD-14.3-RELEASE-amd64-disc1.iso;
  loader = bhyveload;
}

vm ubuntu {
   .apply $defaults, graphics;
   iso = $isopath/\
   ubuntu-24.04-desktop-amd64.iso;
   loader = uefi;
}
```

The global section includes BMD's global variable definitions and configurations. If multiple global sections appear in the configuration, they are concatenated into a single global section. The template section will be included by the *.include* macro. The *.include* macro can also be used in the template section, like the *default_disk* template in the example. Variables defined in the template and the VM section have the VM scope. The predefined variables are uppercased; the *NAME* variable in the sample holds the VM name and the *ID* variable holds the unique number for each VM, starting from 0. The *common* template has two arguments. If we want to fix the template value for the specific VM, supply the template arguments. The template arguments can be omitted and the default values will be set. If the default value is also omitted, an empty string will be assigned. All variables are treated as strings except in the arithmetic expression.

The arithmetic expression must be embraced by the *$(( ))* like a shell script. Strings and variables that are in the parentheses will be considered as numbers. The result of the arithmetic calculation will be converted to a number string and set in the VM configuration. The value of *graphic_port* can be calculated from the *ID* value. For *ID*, which is unique to individual VMs, the value of *graphic_port* will be unique.

So the *graphic_port* is assigned automatically, how do we know the port number? BMD plugin *avahi* will publish the port number as a VNC service. The VNC client can see the port number via mDNS. The GTK application *bvnc* looks up the VNC service and then connects to the VNC port by the VNC client. And also Finder on macOS will do the same.

If we want to separate the configuration for each VM, the *.include* macro will help us. It is helpful that a provisioning tool like Ansible writes the configuration. The *.include* macro reads another file or files and expands the configuration into the original one. Including a non-privileged user file is available, but it definitely violates security issues. Bhyve runs with root privileges, so that any block devices and disk image files can be read and any file system can be read via the virtio-9p protocol. A malicious user can configure the guest VM to read sensitive files. So, the completely trusted user files should be included. At least, to prevent BMD from disclosing sensitive files, the configuration parser runs under the file owner's credentials. This privilege separation prevents sensitive root-owned files from being included in a user's configuration.

2.2 control interface

The *bmdctl* is a command-line interface to BMD. It connects to BMD via a Unix domain socket, sends a request and receives the result. The *bmdctl* is the same binary as BMD; it is hard-linked to it. The request data format uses the *nvlist_t* structure. The *nvlist_t* is a multi-purpose data structure for C language and it can be serialized to a single binary array and vise vesa. BMD and *bmdctl* binaries are the same, so that the same nvlist library will be linked. It is always safe to ensure that serialized data can be deserialized between the command and the daemon.

Table 1 shows the subcommands of the *bmdctl*. Usually, booting the VM requires root privileges because

Table 1: control subcommands

boot	start the VM
install	boot from ISO image
shutdown	ACPI shutdown the VM
poweroff	foced power-off
reset	reset
list	show all VMs
console	connect to the VM's console

the kernel interface allows access only from the root user. So BMD must run with root privileges to invoke bhyve as the root user. How about the *bmdctl*? The *bmdctl* connects to BMD; it doesn't use the VM kernel interfaces, so it doesn't require root privilege. If a non-privileged user uses the *bmdctl* command to control the VMs, all VMs will be allowed to be controlled by that user. To limit the user access, I introduce the owner property for each VM. If we set *owner=alice;* in the VM configuration, user *alice* can control the VM. To simplify configuration, each VM has a single owner. If we allow multiple users to control, we use an owner group. Setting *owner=alice:staff;* allows members of the *staff* group to control the VM. And the root user can control all VMs.

BMD accepts a control connection via a Unix domain socket, which can see the credentials of the peer client. The credential is set by the kernel so that the client can never fake it. BMD can verify the client credentials reliably and securely.

2.3 auto assignment

Configuring an individual resource, such as a port number or a device name, is a minor but annoying thing. We need to look up all existing configurations and assign an unused number or name. It is a small step at first, but it becomes larger as the number of configurations increases. BMD assigns individual resources automatically as possible it can.

If a VM has network interfaces, we need to assign a tap interface for each. A tap interface can be created and destroyed on the fly on the FreeBSD kernel. There is no reason that tap interfaces are predefined and assigned. Just before starting the VM, we can create tap interfaces for each network interface. So we don't need to configure tap interfaces, but specify to which bridge to be added. BMD automatically creates a tap interface and adds it to the specified bridge. After shutting down the VM, BMD destroys the tap interface.

An nmdm device is used for console access for a VM. It's always created in a pair of ports, A and B are used for the suffix of them. An nmdm can be named in any characters that are allowed to be used in the devfs. The complete nmdm device path will be */dev/nmdm$name[AB]*. No special method is needed to create an nmdm device file. Just open the nmdm device file in which name we prefer. The A side of nmdm is connected to the B side and input data of the A side will be transferred to the B side. The B side works the same. So we need the same name must be shared between the VM and console application. The nmdm name can be made systematically, for example, VM's name plus port number. If an nmdm name is predictable, it could potentially open the same device by accident or to trick the VM. BMD assigns an nmdm name on the fly to reduce the possibility of these accidents or attacks. Definitely, the method that searches for used nmdms and opens a new one will have a possibility of a race condition. We don't have an atomic nmdm creation method, so we cannot solve it completely. The assigned nmdm name is held in BMD. The control command will retrieve the name from BMD and connect to the nmdm device file.

A VNC port is also shared between the VM and the VNC client and an individual port needs to be assigned for each VM. However, BMD doesn't assign the port automatically, because it's hard to find an unused port. To see if the port is free or used, BMD needs to bind a socket to the port once, but the socket cannot be sent to the VM. There is no way to pass the socket. If BMD closes the socket, it definitely has the possibility of a race condition. So the VNC port can be calculated in the configuration file using a unique ID number for each VM to reduce the manual assignment effort.

A disk image is usually an individual resource for each VM. However, BMD doesn't create it automatically. The configuration is intended to be written by a non-privileged user. If BMD creates a disk image based on the user configuration file, this results in BMD allowing a user to create files or block devices. Whether it is intentional or not, this will violate system security. A user must create a disk image file on their own file system or create a ZFS zvol block device on a file system that they are allowed to create.

For the same reason, BMD doesn't create bridge interfaces. All bridges must be created by the administrator. A user only uses it.

2.4 auto inspection

For the use of *grub-bhyve* for a serial console boot, we need to specify which kernel file in the disk image to be loaded for NetBSD and OpenBSD. Table 2 shows the kernel path names to load. NetBSD's kernel filename[3] is always */netbsd* in any case. In the OpenBSD case, we have to choose the correct kernel file path for each boot case: normal boot[4], upgrading the system[7] to the new version and installing from an ISO image. These are systematic names and can be found automatically by inspecting the disk image. BMD mounts the disk image and looks for the kernel file, then generates the boot command for the *grub-bhyve*. This feature allows users to try out any version of OpenBSD simply by specifying an ISO image file. And also, the OpenBSD *sysupgrade* command runs without any VM configuraion update.

Table 2: boot kernel

start up type	NetBSD	OpenBSD
boot from HDD	/netbsd	/bsd
upgrading	/netbsd	/bsd.upgrade
install from ISO	/netbsd	/*version*/*arch*/bsd.rd

2.5 console access

The *cu* command in FreeBSD is often used to access the VM's console via an nmdm device. The *cu* has a long history since 4.2BSD. The original purpose of the *cu* was connecting to the serial devices, which is called a UART now. The other application that also uses a serial device was *UUCP*. *UUCP* was periodically invoked from *cron* and the invoke timing was hard to know for users. So, to work with *cu* and *UUCP* exclusively, they tried to get a file lock in the */var/spool/lock*. These days, *UUCP* has gone to the FreeBSD ports. However, the *cu* still obtains the file lock. The */var/spool/lock* is writable to the root user and users in the *dialer* group. The other user cannot use the *cu*.

The *nmdm(4)* implementation doesn't notify peer closing. It's useful when the peer has changed to bhyve from *bhyveload*. The user's console connection remains online and all messages are displayed. However, after bhyve stops, the connection also remains. The user must manually close the console connection. BMD knows when bhyve process terminates. If it notifies the console subcommand, the console can close the connection.

2.6 error log file

As described before, BMD basically assigns an nmdm device to the VM console and won't assign stdio. Even if bhyve shows an error message to the stderr, users won't have a chance to see it. So BMD redirects bhyve's stdout and stderr to the specified file in the VM configuration. Users may see the error messages and will be able to debug the configuration, or some errors happening to bhyve. For the VM owner can be a non-privileged user, the error log file must be written under the owner's credentials.

2.7 graceful shutdown

While the host operating system is shutting down, it should wait for all VMs to shut down gracefully. BMD manages the state of the VMs. It sends a TERM signal to the running bhyve while BMD receives a TERM signal to stop the daemon. And BMD waits for all VMs to shut down. If a VM won't shut down within a stop timeout, the VM is forcibly killed. The stopping BMD delays after all VMs stop. So the host operating system shut down also delays.

2.8 plugins

A plugin extends the handling of VM. The feature is provided by shared objects that include the plugin interfaces against BMD. Users can select the features without recompiling, just load the shared object. The interfaces are defined as callback functions. BMD calls the interface for each VM when a promised event happens.

3 plugins are ready now. One is the hook command plugin that invokes a command before a VM starts, after a VM terminates and when the state of a VM changes. Second is the Avahi plugin that publishes the VM's VNC port as an RFB (remote frame buffer) service. Users can look up the VNC port via the mDNS protocol. The last is experimental support of the QEMU plugin that changes the hypervisor to QEMU. The QEMU hypervisor supports CPU emulations so that users can boot a VM that is a different CPU architecture from the host.

2.9 Wake on LAN

If a VM is rented for a user who doesn't have shell access to the host machine, the user has no way to boot the VM. The Wake on LAN feature will help in this situation. BMD watches WoL packets to the bridge interface that the VM network interface is configured to be added to. A MAC address must be supplied to the VM network interface to which the WoL packet triggers to boot.

3 IMPLEMENTATION

3.1 parser

While parsing the configuration file, the template syntax cannot determine the VM scope variables. They are defined in the VM sections and may have different values for each VM and also template arguments. So the parser reads all configuration files, parses the structure of each section and builds an abstract syntax tree (hereafter, AST). The parser is implemented by *yacc(1)* & *lex(1)*. Most of the implementation is inspired by the *jail.conf(5)* parser.

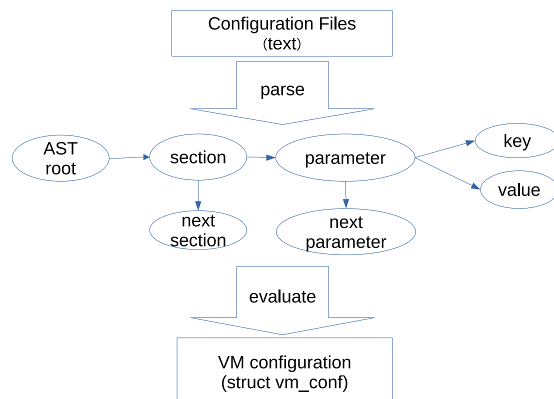

Figure 1: Parsing Flow

As described in the Design section, the parser must run under the configuration file owner's privilege. This is intended to prevent the disclosure of sensitive information, even if malicious data exploits the parser. Complete privilege separation needs to call *setuid(2)* and *setgid(2)*. A process that issues these system calls never regains root privilege. However, BMD must continue to run under root privilege. So BMD creates a child process for the parser and the child process inherits the file owner's privileges.

To return the AST structure, shared memory is allocated between BMD and the parser child process. All nodes of the AST are assigned from the shared memory. This allocation is simple because there is no need to free nodes. All nodes are necessary until parsing is finished. If the allocated memory is not enough for a configuration file, the child parser exits with an error code. When BMD detects a child parser failure, it rolls back the AST node before parsing, then allocates another shared memory and connects to a linked list of shared memories. BMD re-creates the child process and starts parsing again.

The next step evaluates the AST. The AST includes 3 sections: global, template and VM. First global sections are evaluated and global variables are defined. Next, VM sections are evaluated. While each VM section is evaluated, it has its own context, including its own variable tables. A variable table holds pairs of variable names and their values. Note that the value of the variable is a simple string, not any structures. This limitation simplifies implementation and avoids a loop over variable references. There are 3 types of variable tables: global, VM and argument scope. The global scope table is shared among all VM sections. All contexts have the same reference to the global scope variable table. The VM scope variable table is dedicated to the VM section. The variables defined in the VM section will be held in the VM scope variable table. The argument scope table will be used while applying a template.

The template arguments are held in the argument scope table. The template section will be evaluated only if it is applied from the VM section. The reference to a variable will be solved in the order of argument scope, VM scope and global scope. This order overrides the same-named variable with the higher one. Rewriting the same-named variable is a little tricky. Variables defined in the VM and template sections must be stored in the VM scope variable table. If the same-named variable is in the argument table, it won't be recognized as changing because the argument scope reference order is higher than the VM scope. So, the same-named argument scope variable must be removed while writing the variable. The VM scope variable always overrides the global scope variable.

The VM section has parameters and their values for VM configurations. For example, the *ncpu* parameter indicates the number of CPUs the VM has. Basically one parameter has one value. The *disk* and *network* parameters have multiple values that means a VM may have multiple disks and network interfaces. These parameters contains a list of values.

The value contains strings, variables and arithmetic expressions. To evaluate a value, strings are concatenated, variables are concatenated by referencing the variable table and the result of calculating the arithmetic expression is concatenated.

The evaluation up to this point determines which parameter will take on which specific value. Some parameters have modifiers on their values. For example, the *disk* parameter may take a disk controller type before a disk image path. These modifiers are different for each parameter, so each has its own parser. The parser specified by the parameter name interprets the value and creates the VM configuration. Note that the modifiers are not the syntax of the yacc parser. If the yacc syntax includes the modifiers, the variable table must contain the structure of the syntax so that users can contain modifiers in a variable. It may have a variable in the variable table value. A loop reference among the variables has to be solved.

3.2 auto inspection

The goal of auto inspection is to make a boot command for the *grub-bhyve*. Analyze the disk image and find a boot kernel and pass the serial port name if specified in the VM configuration. For NetBSD, it is easy, because the boot kernel is consistently named *netbsd* in both disk images and iso images. Mount the image and check if the /netbsd exists. If it is found, pass the disk image as the root filesystem and also pass the serial port specified in the VM configuration.

For the OpenBSD installer, the boot kernel is under directories of a version number and an architecture name. Look for a directory that contains numbers and a period and check an architecture directory in it. If it is found and it contains a kernel file named *bsd*, we can make an install kernel file path. The OpenBSD install kernel contains a root filesystem in the binary file, so it doesn't need an external root filesystem. Passing the serial port for the console VM completes the boot command.

For the OpenBSD boot disk, BSD label partitioning is often used. There is no need to boot from UEFI for a serial console server. The *grub-bhyve* finds the boot kernel image in a root filesystem of a BSD label partition. The inspection process needs to mount the root filesystem in a BSD label partition. However, the offset in BSD labels is different between FreeBSD and OpenBSD. It is a historical reason. While porting BSD to PC, we needed to place the BSD label partitions in the MBR, nowadays GPT. The FreeBSD's BSD label offset is the relative position within the MBR partition where the BSD label is located. The OpenBSD's BSD label offset is the absolute position within the whole disk. OpenBSD can access any partitions, regardless of disk partition type, via BSD labels if all offsets are correctly written. So the FreeBSD kernel cannot access via the OpenBSD's BSD labels, except for the first partition. If the FreeBSD kernel mounts the top of the MBR partition, it can access the first BSD label

partition, which is usually the root file system. The inspection process looks for the *bsd.upgrade* kernel file first for upgrading the system, then next to the *bsd* kernel file for booting.

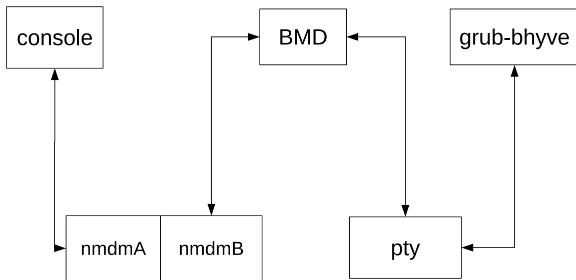

Figure 2: Pseudo terminal for grub-bhyve

After finding the boot kernel and building a boot command, BMD has to pass the boot command to the grub-bhyve process. BMD snoops the dialog between the grub-bhyve and the user console. Just after the grub-bhyve sends the first prompt, BMD inserts the boot command. It will boot the target kernel. For snooping the dialog, BMD creates a pty for grub-bhyve and transfers the dialog from the pty to the nmdm device that the user console opens. And also transfers the user input strings from the nmdm to the pty. Figure 2 shows this sequence.

3.3 state machine

BMD manages the states of all VMs. Figure 3 shows the VM's states. Basically, BMD waits for an event, performs the associated task and then sets the VM's next state. The events are process termination, receiving a signal (for reloading VM configurations), socket receive (from the bmdctl command) and timers (delayed boot and loader timeout). All events can be captured by the *kevent(2)* system call. BMD runs an event loop that processes events returned by *kevent(2)*. Because events occur infrequently, a single thread is sufficient. With a single thread, there is no need to lock structures held within BMD and the VM state is always reliable.

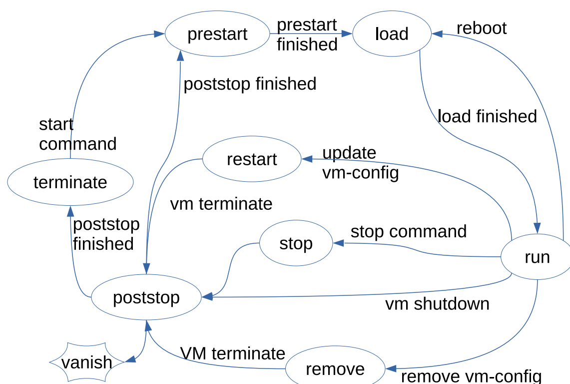

Figure 3: VM state transition diagram

Initially, a VM is terminated. If a VM is booted by the boot command or auto-booted by BMD, its state changes to it prestart while the prestart plugins run. After finishing the prestart plugin, the VM's loader is started. The *load* state is used for *bhyveload* or *grub-bhyve*. If no loader is needed, the *load* state is skipped and the state is set to *run*. After finishing the loader, its state also changes to *run*.

The *run* state means that bhyve is running. Bhyve process can terminate in several ways. One is that bhyve exits spontaneously when the guest OS shuts down. This case changes the state to *poststop*. The other way is for a user to stop the VM using the *bmdctl* command. BMD sends a TERM signal to bhyve and changes the VM state to *stop* while waiting for bhyve to quit. After bhyve process quits, the VM state is changed to *poststop*.

If the VM's configuration is removed, the VM is forcibly stopped. After the VM stops, BMD removes the management structure from the memory. To remember this work, the VM's state is set to *remove*. After bhyve process quits, the VM state is changed to *poststop*, then the management structure is freed.

The *poststop* state in this case definitely violates state transition theory: if a transition from a different state determines the next state, then that state should be different. However, the *poststop* state is used for plugins. Plugins don't care about the previous state and should do the same things in every *poststop* state. Multiple *poststop* states may confuse the plugin developers. So, I decided to use the one *poststop* state and to save the previous state from the *poststop* state to determine the next state.

The same violation happens at the *restart* state. This state is used for the *reboot_on_change* parameter when set to *yes*. After reloading the VM configuration, if one or more parameters have changed, BMD shuts down the VM and then reboots it to apply the new configuration. This sequence must go through the *poststop* and *prestart* states. So the *poststop* state that came from the *restart* state must move to the *prestart* state.

After the poststop plugin terminates, the VM state changes to *terminate*. The VM can be booted at the user's request.

3.4 opening nmdm

Usually, an nmdm device is assigned to a com1 device as console. Users open the nmdm device to access the console. BMD assings the device name automatically so that users need to ask BMD which nmdm device is assigned. The *bmdctl console subcommand* does this internally. While asking the nmdm device name, BMD opens the device on its behalf and returns the file descriptor. By opening on behalf, the operation permissions can be checked in the same way as for other operations such as starting or stopping a VM.

Bhyve opens an nmdm device without blocking (O_NONBLOCK). If both sides of nmdm open without blocking, it fails occasionally and the reason is un-

known. So BMD needs to open with blocking mode. Usually, when bhyve starts up, it opens the nmdm device immediately, so there is very little waiting time on BMD side. But if bhyve fails without opening the nmdm, BMD waits forever and all operations are blocked. Because BMD runs on a single thread, to avoid this blocking, BMD needs to open an nmdm device in a background task, fork a child process that opens the nmdm device and returns a file descriptor.

3.5 reloading configurations

BMD loads all VM configurations at startup. The configuration parser builds up a list of VM configurations, then allocates a VM structure for each VM configuration. The VM structure holds the VM state, pid of bhyve or loader, pipes to stdin/out/err and allocated resources such as tap interfaces and nmdm devices. It also contains both the current and a new VM configuration.

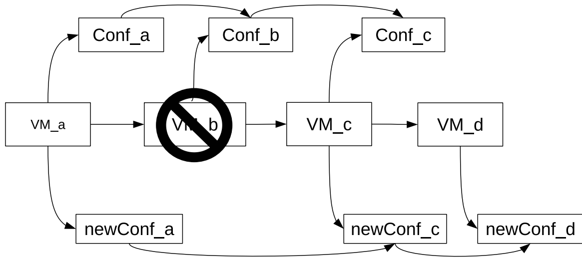

Figure 4: Reloading VM configurations

In Figure 4, the upper line links to the current and the lower line links to the new one. Figure 4 shows that, initially, three VM configurations are loaded and each VM structure is linked; then, upon reloading the configurations, a new list is created. The new list may contain the same VMs' configurations, may update which parameter, may lack the existing VMs, or may add new configurations. In Figure 4, VM b is removed and VM d is added. The VM b will be shut down by a TERM signal, then set to the *stop* state. The VM d is linked to a new VM structure. The new configuration will replace the current one when the VM starts. In other words, the new configuration will be available after the next boot. The running VM must be shut down, not rebooted, to update its configuration. The prestart/-poststop plugins may change during a reload; the same pair of plugins must run on boot and shutdown. The running VM booted with the prestart plugin in the current configuration; it must shut down with the poststop plugin in the same configuration.

3.6 opening error log files

If bhyve process encounters a problem, it prints messages to stdout or stderr. It is helpful to troubleshoot the problem. However, bhyve inherits BMD's stdout and stderr; they are redirected to */dev/null*. No one can see the messages. BMD opens an error log file and redirects its stdout and stderr to it. BMD also runs with root privilege, opening a file according to the VM configuration that an unprivileged user wrote. It is obviously dangerous. Each configuration has its owner and the default owner is the same as the configuration file owner. So, opening an error log file with the owner privilege is reasonable. BMD changes to the owner's credentials to open an error log file.

For changing to the owner credential, *seteuid(2)* and *setegid(2)* are enough. They affect the process wide. However, BMD runs in a single thread. There is no other thread and no problem.

3.7 console access implementation

The console subcommand of the *bmdctl* works as same as *cu* without obtaining a file lock. And it retrieves the file descriptor, which is opened to the nmdm device of the console from BMD. BMD checks the credentials of the *bmdctl*. If it's the owner of the VM, opens the nmdm device and flocks the device file for exclusive access, then returns the file descriptor. This sequence allows the VM owner to access the VM console.

The console subcommand also sends its pid and signal type with the file descriptor request. After the target VM terminates, BMD sends the specified signal to the console subcommand. The signal terminates the console subcommand, causing it to terminate automatically.

3.8 plugins

A plugin is a single shared file that contains a plugin descriptor named *plugin_desc*. A plugin descriptor describes the plugin name, callback methods for each event and extension methods for loader and VM execution. Table 3 shows the list of callbacks. The *initialize* method is called when BMD loads the plugin at boot and should initialize the plugin data. The *finalize* method is called when BMD terminates and performs plugin termination, such as freeing resources allocated by the plugin.

Table 3: callback methods

method	when
initialize	at startup
finalize	on terminate
on_status_change	VM status changes
parse_config	parameter extention
on_reload_config	on reloading configuration
prestart	before VM start
poststop	after VM terminate

When a VM state changes, the *on_status_change* method will be called. This callback method will never affect VM execution, whether it succeeds or fails.

Table 4: loader methods

method	action
load	load VM
cleanup	clean up VM resources

Reporting tasks will be suitable. But the task will never be blocked or wait for anything. Blocking causes BMD's execution to stop and it won't handle events. If the plugin task takes longer, create a child process to handle it. Waiting for the child process will be handled in the daemon's event loop.

The *parse_config* extends a VM configuration parameter for the plugin. The configuration parser passes an unknown parameter name and value to the *parse_config* method. If the plugin knows the parameter, configure the value. If not, ignore the parameter; another plugin probably knows, or it's a mistaken parameter. An *nvlist_t* structure is passed to the method to store the value. The same structure for each VM is passed to all the callback methods except *initialize* and *finalize*. Through the *nvlist_t* structure, the configuration value will be shared among callback methods.

A plugin can also use the *nvlist_t* structure for working storage. If it starts a child process and needs to remember its pid, it can store it in the *nvlist_t* structure. While VM configurations are reloaded, a new *nvlist_t* structure will be created for the new configuration. If a plugin inherits some value (the pid of the example) from the old configuration, it must copy it to the new configuration. The *on_reload_config* callback is called for this purpose.

The *prestart* callback is called before a VM starts and VM execution is delayed until the plugin calls *plugin_start_virtualmachine*. It is suitable for a preconfiguration task. If the task fails, the plugin can stop VM execution by calling *plugin_stop_virtualmachine*. If a *prestart* task takes longer, it should create a child process to handle it. The *poststop* callback is called after a VM terminates. It is suitable for cleanup tasks configured in the *prestart*. After finishing the *poststop*, users can start the VM. While *poststop*, the VM will never start.

The current supported loaders, *bhyveload* and *grub-bhyve*, are defined as the loader plugins internally. BMD looks up the loader by its name using the *loader* parameter value. Then, it loads the VM by the *load* method. After the loader finished, BMD calls the *cleanup* method of the loader plugin. The new loader can be created in the same way as these loaders.

The VM methods in Table 5 are interfaces to handle a VM. BMD calls the *start* method to boot a VM and calls the *acpi_poweroff* method to stop it. Internally, the *bhyve* plugin handles bhyve process. The *start* method implementation generates bhyve's commandline parameters and invokes bhyve. The *acpi_poweroff* method implementation sends a TERM signal to bhyve process. The *reset* and *poweroff* implementations call

Table 5: VM methods

method	action
start	start VM
reset	force reset VM
poweroff	force power off VM
acpi_poweroff	ACPI shutdown VM
cleanup	clean up VM resources

ioctl(2) of */dev/vmm/¡VM name¿*. The new hypervisor can be supported if these VM methods are implemented. The QEMU plugin is implemented experimentally, but not all features are supported.

3.9 Wake on LAN

BMD can see all incoming packets on the host machine and monitor WoL packets on any interface because it runs with root privileges. Usually, WoL works on the same LAN as the target host and a router won't transfer WoL packets to other LANs. So, BMD should monitor WoL packets in the LAN to which a target VM is connected. It is necessary for the POLA (Principle of Least Astonishment). BMD knows to which bridge a VM will be connected from the VM configuration. It watches the bridge and waits for a WoL packet containing the VM nic's MAC address. Capturing packets is implemented by *bpf(4)*. All broadcast packets are captured and checked to see if the MAC address is present in the payload. BMD also supports *vale(4)* switches. Creating a *netmap(4)* port to receive broadcast packets.

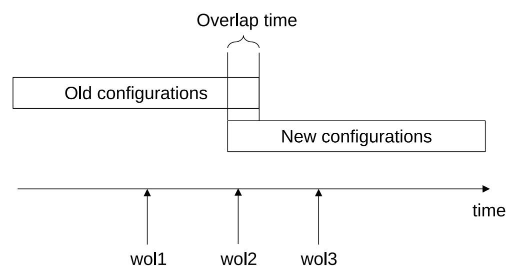

Figure 5: Overlap time

While reloading VM configurations, BMD may change the monitoring bridges. If BMD quits and restarts monitoring, even for a short time, it may miss a WoL packet. So, BMD should quit monitoring with the old configurations after starting with the new configurations. It monitors the old and new configurations for a short time. Figure 5 shows this overlap time.

BMD forks a child process and monitors the bridge interfaces as specified in the VM configuration. After reloading the VM configurations, BMD forks a new

child process and monitors with the new configuration. The previous child process has its own independent memory, which is copied at fork. Reloading of the parent process doesn't affect the child process. The previous child continues to monitor with the old configurations. The new child process has the new configurations because it is forked after the configurations are reloaded. After the overlap time has passed, BMD quits the old monitoring process. The new child process continues to monitor WoL packets.

The current overlap time is one second. If reloading VM configurations occurs frequently, it forks a child monitoring process during the reload. It can be a fork bomb. The number of monitoring processes is currently limited to 10. Reloading over 10 times a second will be ignored.

If a monitoring process finds a WoL packet containing a VM nic's MAC address, it sends the VM name via *socketpair(2)*. Then BMD boots the VM.

4 FUTUER WORK

The current BMD implementation needs to improve security for users' VM management. For example, a disk image access should be permitted to a file or a block device that the VM owner can access. And also, virtio-9p shared file systems. Otherwise, a VM owner can see all the files on the host's filesystem from within the VM. Bridge interfaces should be limited to a white list. Unprivileged users should not be allowed to see all packets to or from the host. These limitations break compatibility with the current configuration, so they will be optional.

The cloud-init feature offers a flexible deployment of VMs. It will support a lot of configurations in YAML format. We need more consideration to expand the configuration syntax or include an external YAML file.

VM migration is a great feature if bhyve supports it. BMD is a daemon that is always running on a host machine. It can sync the VM state to the peer host machine. Expansion to support the VM migration will be tough work, but possible.

5 CONCLUSION

This paper presented the design and implementation of bhyve Management Daemon (BMD), a lightweight and secure management framework for bhyve virtual machines on FreeBSD. BMD focuses on simplicity, configurability and safe operation by unprivileged users, for various kinds of usecases. By adopting a configuration syntax inspired by jail.conf(5), BMD enables reusable templates, variable scoping and automated resource assignment, significantly reducing configuration redundancy and operational complexity.

BMD introduces several practical mechanisms, including automatic assignment of devices such as tap interfaces and nmdm consoles, controlled access through credential-aware Unix domain sockets and extensibility via a plugin architecture. These features allow users to manage virtual machines without direct root privileges, while maintaining strong security boundaries through privilege separation and careful resource handling. Support for headless boot, automatic kernel inspection, graceful shutdown and Wake on LAN further enhances usability in both shared-host and server-oriented environments.

Overall, BMD demonstrates that a daemon-oriented, CUI-focused approach can provide an effective balance between usability, security and extensibility for bhyve-based virtualization.

References

[1] *bhyve(8)*. FreeBSD Manual. URL: `https://man.freebsd.org/cgi/man.cgi?bhyve`.

[2] *bhyveload(8)*. FreeBSD Manual. URL: `https://man.freebsd.org/cgi/man.cgi?bhyveload`.

[3] *boot(8)*. NetBSD Manual. URL: `https://man.netbsd.org/x86/boot.8`.

[4] *boot(8)*. OpenBSD Manual. URL: `https://man.openbsd.org/boot.8`.

[5] Matt Churchyard. *vm-bhyve*. URL: `https://github.com/churchers/vm-bhyve`.

[6] Peter Grehan. *grub2-bhyve*. URL: `https://github.com/grehan-freebsd/grub2-bhyve`.

[7] *sysupgrade(8)*. OpenBSD Manual. URL: `https://man.openbsd.org/sysupgrade.8`.