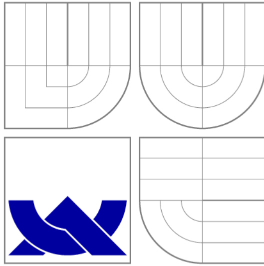
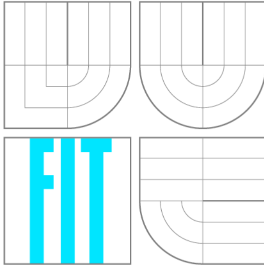


BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

LIBVIRT ADMINISTRATION API

APLIKAČNÍ ROZHRANÍ PRO ADMINISTRACI

PROJEKTU LIBVIRT

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE
Bc. ERIK ŠKULTÉTY

SUPERVISOR
VEDOUCÍ PRÁCE
Ing. ALEŠ SMRČKA, Ph.D.

BRNO 2016

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2015/2016

Zadání diplomové práce

Řešitel: **Škultéty Erik, Bc.**

Obor: Počítačové sítě a komunikace

Téma: **Aplikační rozhraní pro administraci projektu Libvirt**
Libvirt Admintration API

Kategorie: Operační systémy

Pokyny:

1. Nastudujte projekt Libvirt pro správu virtuálních počítačů.
2. Analyzujte požadavky komunity na aplikační rozhraní (API) projektu Libvirt pro jeho vlastní správu. Navrhněte aplikační rozhraní administrace Libvirt. Rozhraní bude zahrnovat správu připojených klientů, změnu počtu obslužných vláken a změnu konfigurace démona Libvirt včetně nastavení a filtrace záznamů.
3. Implementujte aplikační rozhraní pro administraci Libvirt v jazyce C. Aplikační rozhraní zdokumentujte na úrovni přijatelné pro upstream projektu.
4. Aplikační rozhraní ověřte pomocí kombinačního testování. Testy pokryjte všechny klíčové části API.

Literatura:

- Bolte, M.; Sievers, M.; et al. Non-intrusive Virtualization Management using Libvirt. In Proc. of DATE 2010. doi:10.1109/DATE.2010.5457142
- Yamato, Y.; Nishizawa, Y.; Nagao, S.; Sato, K.; Fast and reliable restoration method of virtual resources on OpenStack. IEEE Transactions on Cloud Computing, Issue: 99. 2015. doi:10.1109/TCC.2015.2481392
- Dokumentace API Libvirt. URL: https://libvirt.org/html/index.html

Při obhajobě semestrální části projektu je požadováno:

- První bod bod zadání a část návrhu API.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Smrčka Aleš, Ing., Ph.D.**, UITS FIT VUT

Konzultant: Prívozník Michal, Ing., RHcz

Datum zadání: 1. listopadu 2015

Datum odevzdání: 25. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních.technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstract

This thesis addresses the virtualization topic, more specifically, it deals with libvirt virtualization management library, the goal of which is to provide a common and stable layer to manage virtual machines that deals with all the hypervisor or virtualization solution specifics transparently to the user. Most of the functionality exposed by libvirt is implemented in form of services within a daemon called libvirtd. One of the main reasons why libvirt utilizes a daemon is to provide a remote management of virtual machines running on hypervisors which do not support remote management. However, the daemon lacks support for managing itself during runtime. Although its configuration can be altered via a configuration file, the configuration is persistent only. Additionally, each time the configuration is changed this way, the daemon needs to be restarted, which might not always be the optimal solution. Therefore, an idea of exposing an administration interface through libvirt library which would provide users with libvirtd's runtime management arose. The main goal of this thesis is to design and implement a set of administration application interfaces which would provide features including adjustment of number of workers in a server's threadpool, modifying logging levels, filters, and logging outputs, as well as remote client management.

Abstrakt

Tato práce se zabývá problematikou virtualizace, konkrétně virtualizační knihovnou libvirt, cílem které je správa virtuálních strojů a podpora různých typů hypervizorů a virtualizačních řešení jednotným způsobem transparentním pro uživatele. Podstatná část funkcionality knihovny libvirt je na pozadí implementována formou démona libvirtd. Ačkoliv libvirtd démon poskytuje služby pro správu virtuálních strojů, neumožňuje správu sebe samého, kromě změn hodnot parametrů v konfiguračním souboru. Pro změnu nastavení je pak standardním přístupem změna v konfiguračním souboru a následný restart démona. Jelikož uvedený způsob mění pouze perzistentní konfiguraci a restart démona nemusí být vždy optimální řešení, vznikla idea administrativního rozhraní knihovny libvirt, které by umožnilo správu démona za běhu. Hlavním přínosem této práce je návrh a popis implementace aplikačního rozhraní pro administraci knihovny libvirt. Konkrétně pro tuto práci byla zvolena rozhraní pro konfiguraci počtu obslužných vláken, nastavení úrovně a filtrovacích parametrů pro žurnálovací podsystém a správu připojených klientů na straně démona libvirtd.

Keywords

virtualization, libvirt, virtual machine, hypervisor, libvirtd, admin API

Klíčová slova

virtualizace, libvirt, virtuální stroj, hypervizor, libvirtd, admin API

Reference

ŠKULTÉTY, Erik. *Libvirt Administration API*. Brno, 2016. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Smrčka Aleš.

Libvirt Administration API

Declaration

I declare that this thesis and the work presented in it are my own and that it has been created by me as the result of my own research under the supervision of Ing. Aleš Smrčka, PhD. and Ing. Michal Prívozník, consultant from Red Hat Czech, s.r.o. Additional information and details about the libvirt library internals were provided by members of the Red Hat's libvirt virtualization team. I confirm that I have acknowledged all main sources of help.

. . . .
Erik Škultéty
May 24, 2016

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Ing. Aleš Smrčka, PhD., for his continuous insightful suggestions and comments on my writings. I am also grateful to my co-advisor and consultant from Red Hat Czech, Ing. Michal Prívozník for valuable discussions which pointed me in the right direction and helped me resolve the practical aspects of my work. Finally, I would like to thank all members of the Red Hat's libvirt virtualization team for providing details that helped me understand libvirt's internals better.

© Erik Škultéty, 2016.
This thesis was created as a school work at the Brno University of Technology, Faculty of Information Technology. The thesis is protected by copyright law and its use without author's explicit consent is illegal, except for cases defined by law.

Contents

1 Introduction **3**

2 Virtualization Fundamentals **5**
2.1 Dual-Mode Operation and Protection Rings 6
2.2 Classical Virtualization 8
2.2.1 Problems with Virtualizing x86 Architecture 8
2.3 Software Virtualization 9
2.3.1 Binary Translation 9
2.3.2 Paravirtualization 9
2.3.3 Xen 9
2.4 Hardware Assisted Virtualization—x86 Extensions 11
2.4.1 KVM 12
2.4.2 QEMU 14
2.4.3 Virtio 15
2.5 Container-based Virtualization 16

3 Libvirt Project: The Virtualization Library **17**
3.1 Libvirt's Objective 17
3.2 Libvirt as a Middleware Layer 18
3.3 Libvirt's Architecture 19
3.3.1 Stateful and Stateless Drivers 20
3.3.2 Connection Establishment 21
3.4 Libvirtd Daemon 23
3.4.1 Communication 23
3.4.2 Message Processing and Task-based Model 24

4 Administration Interface Specification **27**
4.1 Objective 27
4.1.1 Load Balancing 27
4.1.2 Runtime Introspection 28
4.1.3 Client Management 28
4.2 Interface Overview 28

5 Implementation Details of Selected Parts of the Administration Interface **33**
5.1 Common Data Types 33
5.2 Load Balancing: Managing a Threadpool's Attributes 35
5.2.1 Problem Details 36
5.2.2 Solution 37

5.3 Runtime Introspection: Managing a Daemon's Logging Parameters 37
5.3.1 Problem Details 38
5.3.2 Solution 38
5.4 Virt-admin Command-line Interface 39

6 Testing of Selected Administration Interfaces **41**
6.1 Modifying Daemon Logging Settings 42
6.2 Setting Threadpool Parameters 46
6.3 Testing Toolset Details 49

7 Conclusion **51**

Bibliography **52**

Appendices **55**
List of Appendices 56

A Libvirt Administration API Reference **57**

B Libvirt Administration API Usage Examples **71**
B.1 Listing Available Servers on Daemon 71
B.2 Listing All Clients Connected To a Given Server 72
B.3 Getting and Setting Logging Level 74
B.4 Getting and Setting Logging Filters 76
B.5 Getting and Setting Logging Outputs 77
B.6 Getting and Setting Server Threadpool Parameters 78
B.7 Getting and Setting Client Limits on Server 79
B.8 Retrieving a Client's Identity Information 81
B.9 Closing a Client's Connection Forcefully 84

C Contents of the CD **86**

Chapter 1

Introduction

Virtualization technology can be accounted a topic of a broad interest, receiving very close attention these days. It managed to revolutionize and reshape the computer industry top to bottom since its introduction. Although it is often referred to as a “new” technology, because of its immensely growing integration into corporate infrastructures throughout the recent years, mainly due to different kinds of regulations, economic factors and more competition which forces companies to make use of the advantages of virtualization, the idea itself is in fact more than a half-century old, first remark being in the paper *Processing Time Sharing in Large Fast Computers* by Christopher Strachey back in 1959 [25]. The basic idea was to improve man-machine interaction by providing the users concurrent, interactive access to the computer. IBM later achieved this by giving each user a virtual machine, which transparently enabled time-sharing and resource-sharing on the (at that time) expensive hardware.

There are several types of virtualization with the most common ones being server virtualization and storage virtualization. Regardless of virtualization type, the principle of providing an abstraction from physical resources and their characteristics remains the same. By breaking the fixed “one owns all” relationship between the operating system and physical assets sitting below, it optimizes the physical resources for efficiency. To clarify the principle in simple terms, virtualization is a combination of software and hardware engineering that mainly creates

- several virtual resources on top of one physical resource or
- one virtual resource on top of several physical resources.

There are several benefits that come with the virtualization technology, with the most notable ones being

- *reduced expenses and energy saving* - migrating physical servers to virtual ones and consolidating them to fewer physical machines reduces the costs related to power consumption and air conditioning needs,
- *isolation, testing, and security* - testing labs within isolated networks that provide separate controlled environments for a tested application to be deployed to, and
- *reliability and availability* - using migration process to transfer a virtual machine’s state or the underlying storage to a different host, in order to diminish downtime of a service in case of a failure of the original host.

Thesis Motivation

Despite all the benefits virtualization provides us with, without any management of virtual resources, virtualization would only remain a concept rather than becoming a technology putting the benefits into production use. There are countless virtualization management tools available on the market today, intended to be used with specific virtualization type. However, for the purpose of this thesis, the only relevant management tool discussed will be *libvirt*. In exact terms, libvirt is not only a management tool for platform virtualization, it is rather a toolset encompassing three components within itself—an open source library, a daemon, and a management tool. The fundamental goal of libvirt is to provide users with a uniform management interface for different kinds of virtualization solutions. Although libvirt offers means to configure all of its components, there is no way to do it during runtime so far. Turning the focus towards *libvirtd* daemon, since it is the key part of the whole toolset implementing most of libvirt's features, it lacks support for managing itself despite the fact that it is accounted as being the management backend. The only configuration available for libvirtd today is through a configuration file. However, this type of configuration is persistent and only serves the initial setup.

Thesis Contribution

The idea of having a separate administration interface that would allow runtime management of the libvirtd daemon to certain extent arose after a customer request was created[1] to expose libvird's current state, so they could monitor the number of connected clients to it proactively, thus being able to tweak the limit to the maximum number of allowed client connections, therefore adapting it to the current load. Without any information about libvirtd's current state, it was rather difficult to set the limit appropriately in the configuration file. Having such a feature implemented would allow them to prevent the daemon from suddenly stopping to accept any more connections with an error once the limit was reached. This idea was later extended to also support reconfiguration of daemon's logging settings to enhance runtime introspection of a potentially malfunctioning virtual machine, reconfiguration of the current number of worker threads in the daemon, as well as forcefully disconnecting individual client connections. The goal of this thesis then is to design and implement administration interfaces for the aforementioned use cases in libvirt library.

This chapter provided an introduction of virtualization technology, the base principle and some benefits that it offers. The concept is further explored by Chapter 2 which immerses deeper into the explanation of the principle, focusing on server virtualization and major approaches to it, as well as giving some credit to different hypervisors. Understanding the virtualization fundamentals is crucial in order to fully comprehend libvirt's architecture and the way it internally works, both of which are examined by Chapter 3. Chapters 4 and 5 address the key parts of the thesis—design and implementation of the administration interfaces. Selected application interfaces are then tested using the equivalence partitioning methodology with the details covered by Chapter 6. Lastly, Chapter 7 then confronts the results and presents possible follow-ups of the project.

[1] https://bugzilla.redhat.com/show_bug.cgi?id=735385

Chapter 2

Virtualization Fundamentals

Introduction mentioned the existence of server and storage types of virtualization, but the list of the most common virtualization types that can be encountered in the computer industry nowadays is a bit longer, starting with

- server virtualization,
- storage virtualization,
- network virtualization,
- I/O virtualization, and
- client virtualization.

While this thesis covers the server virtualization of x86 architecture as one of the major types of the virtualization, some credit to I/O virtualization is also given later in this chapter. Description of the remaining types of virtualization mentioned above can be found in [12, 23, 32, 6]. Before engaging in server virtualization approaches, describing different types of hypervisors and concepts behind them, it is necessary to first establish some vital knowledge base as the information offered in further sections build upon it. This includes some virtualization terminology, operation modes of an operating system as described in [22], and protection rings mechanism. Although it is not difficult to find more virtualization related terms, following are the absolutely necessary ones to know.

Guest Operating System

An operating system running in a virtual machine environment that would otherwise run directly on the hadware as a separate system. It has no knowledge of the existence of another guest operating system running on the same physical system nor has it any knowledge of the host system itself (be it a hypervisor or a conventional operating system, see below).

Hypervisor

As a term, most resources reference hypervisor also as a *Virtual Machine Monitor (VMM)*, which is in fact much older term, already used back in 1960s. But this is not the case of VMware which strictly differentiates between a VMM and a hypervisor (more details can be found in [29, 30]). This thesis however, is going to follow the definition used in [13, 15], which defines a hypervisor as a piece of software responsible for creation and management

of virtual machines which share the underlying physical host's hardware. Physical resources are individually divided into "slices" that are managed by the hypervisor in amounts and time duration as every virtualized operating system needs. In terms of classification, there are officially two categories of hypervisors at the moment:

- *type-1*, also called a *bare-metal* hypervisor – which runs directly on top of host's hardware creating the hardware abstraction for guest operating systems running above, and
- *type-2*, also called a *hosted* hypervisor – which runs as part of a conventional operating system the same way as any other program does. These hypervisors support the broadest range of hardware configurations.

This distinct, as it might seem, classification unfortunately cannot be applied to all hypervisors currently available. A typical example would be Red Hat's *KVM* and FreeBSD's *Bhyve* hypervisors which are kernel modules that supplement the kernel, thus effectively allowing the host operating system to act as a type-1 hypervisor [8]. But since both are part of a conventional operating system and are subject to context switching, they can also be classified as type-2 hypervisors.

Emulation

Emulation is a process that takes the properties of one system, trying to reproduce it with a different kind of system. To put it in virtualization context, it's a process of making an exact copy of the host's hardware resources and all of its functionalities. The advantage is, that one can easily run software compiled for a certain architecture (typically some legacy software) on a completely different architecture. The inherent drawback of this approach is performance, compared to hypervisors conforming to the definition above. What is worse, is that it emulates the host's architecture even though the piece of software is native to the host's architecture.

2.1 Dual-Mode Operation and Protection Rings

In order to ensure a proper execution of the operating system, it is necessary to distinguish between the execution of operating system code and user-defined code. The reason for this is to improve protection of the operating system's integrity from malicious user application that would try to exploit the system. However, it would be impossible without any hardware aid, since every application would execute its code on the CPU unrestricted. Thus, a hardware support to differentiate among various modes of execution was needed. Every operating system needs at least two modes of execution, namely *kernel mode* (also *privileged mode*) and *user mode*. Typically, these two are also the only ones operating systems tend to implement. The aforementioned hardware support then resides in the method how a CPU keeps track of the current execution mode, or in other terms, current privilege level. To fully understand the method, one needs to be familiar with virtual memory management and segment selectors, which is outside of the scope of this thesis, but can be further studied in [22]. What is important to mention however, is the code segment register. And that is because this register contains a 2-bit field called *Current Privilege Level*, which is managed by the CPU itself. The value this field holds is always equal to CPU's current privilege level. Thus four different privilege levels are supported, but as it was already mentioned,

usually not all of them are utilized by operating systems. These protection levels are organized hierarchically into a ring structure, depicted on Figure 2.1, with the inner most ring corresponding to highest privilege[1].
Before the transition between user mode and kernel mode is covered by the next paragraph, it is necessary to say, that in order to perform protection level switch, some hardware support in the CPU is needed. For this purpose, Intel's *sysenter/sysexit* and AMD's *syscall/sysret* instructions enable fast entry to kernel mode, avoiding the interrupt overhead caused by earlier approaches to protection level switching.

Every time user application requests a service from the operating system, it must transition from user mode to kernel mode using a system call to fulfill the request. Figure 2.2 depicts this scenario. Whenever a trap or interrupt occurs, the hardware transitions from user to kernel mode (setting the mode bit accordingly) and vice-versa before passing the control back to the user program. By designating some of the instructions from the instruction set as privileged, hardware enforces execution of these instructions in kernel mode only. If however, an attempt to execute a privileged instruction is performed in user mode, the hardware does not execute it, it rather causes a general-protection exception, trapping to the operating system, which eventually results in terminating the user process.

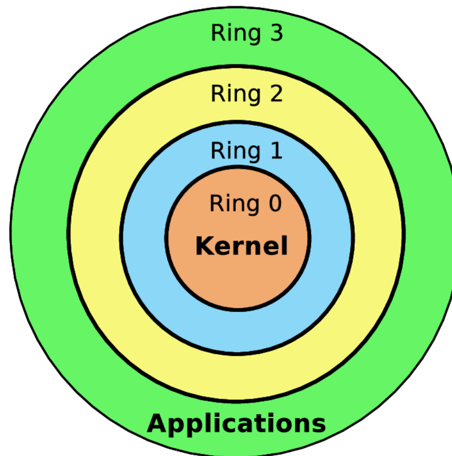


Figure 2.1: Protection rings on the x86 architecture.

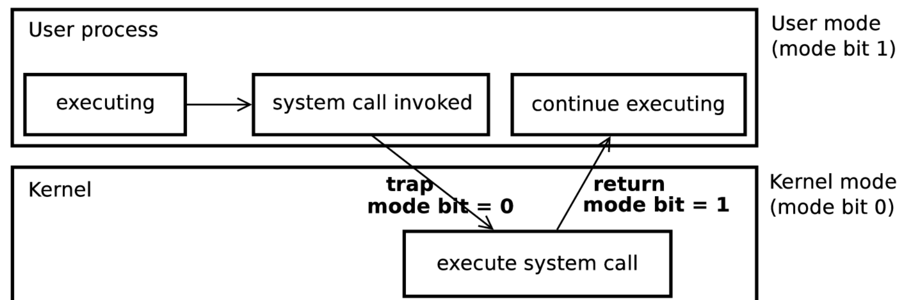


Figure 2.2: Transition between kernel and user mode of execution [22].

[1]The ring comes from the Multics system where the hardware supported up to 64 modes and the access rights changed even between ordinary user procedures stored in users' segments [7].

2.2 Classical Virtualization

By the term classical virtualization is according to [1] understood any architecture, that can be virtualized entirely by trap-and-emulate principle. Regarding this fact, x86 architecture is not classically virtualizable, but it is still virtualizable according to Popek and Goldberg's criteria which they published in their article [19]. They defined three essential features for a system software to possess, in order to be classified as a VMM:

- *fidelity* – all virtual machine instructions running on a VMM execute identically to their execution on hardware,
- *performance* – majority of guest instructions are executed by the hardware without any intervention of the VMM, and
- *safety* – VMM manages all hardware resources.

In a classically virtualizable architecture, all instructions that read or write privileged state can be made to trap when executed in an unprivileged context. A classical VMM executes guest operating systems directly, but at reduced privilege level. The VMM then intercepts traps from the deprivileged guest and emulates the trapping instruction against the virtual machine state. The base idea behind a VMM, according to [1] is to provide an execution environment which meets the guest's expectations about the state of the virtualized hardware, which naturally differs from that of the actual underlying hardware.

There are several special-purpose hardware-based data structures called *primary structures* with typically only one copy in the system. Therefore, the VMM has to derive and maintain copies of these structures called *shadow structures* for each guest. Some of these structures (typically on-CPU structures) can be handled by the VMM trivially, while others like page tables can be challenging, since accesses to the page tables may not always pair with trapping instructions. The details about how coherency between shadow structures and primary structures is achieved by the VMM, as well as how address translation is done both in *memory management unit* (MMU) software virtualization and using Intel's and AMD's hardware support called *nested paging* can be further explored in [1].

2.2.1 Problems with Virtualizing x86 Architecture

Because x86 architecture was not designed baring all the virtualization aspects in mind, there are some rather complicated issues that virtualization has to face as reported by [26], with the most notable ones being:

- *ring aliasing* – a guest operating system is able to figure out the fact it is not running at level 0,
- *non-faulting privileged instructions* – there are quite a few privileged instructions[2] in the x86 instruction set that do not trap when executed in user mode[3], and
- *address space compression* – the VMM has to use a portion of guest's virtual address space to manage the transition between guest operating system and itself.

[2] Adams et al. [1] only mention `popf` instruction as an example.

[3] Remember the guest has been deprivileged, i.e. a VMM replaced it at ring 0, thus moving the guest kernel one level up.

2.3 Software Virtualization

This section is going to elaborate the approaches to tackle the x86 flaws as described in previous section. One of them is emulation as defined and described in Chapter 2. The other ones, that are going to be explained next, are binary translation and paravirtualization. The section concludes with description of Xen hypervisor as the major representative of paravirtualization.

2.3.1 Binary Translation

The fundamental principle of binary translation stays the same for majority of solutions with only slight differences. This paragraph is backed by [1] that explains the principle on VMware's binary translator. The technique VMware used is based on traditional direct execution with runtime binary translation. Unlike other translators which translate code between different CPUs with different instruction sets, VMware's translator works with nearly identical instruction sets, which makes the translation much simpler. The translator is capable of running privileged mode code, while patching guest's privileged x86 instructions as it reads them from guest's memory. Instructions are filled into a translation unit which produces translated blocks ready to be executed, leaving all the non-privileged instructions unmodified. More details about improving the performance and maximizing the overall efficiency are provided by [1, 20].

2.3.2 Paravirtualization

Paravirtualization tackled the performance challenges of having to translate each system call in binary translation. But for this approach to virtualization, it is inevitable that the kernel of the guest operating system is modified. According to [31], the modification in paravirtualization is accomplished by replacing all the critical instructions with hypercalls. Hypercalls enable the guest operating system to communicate directly with the hypervisor layer, in other words, the guest operating system knows it is running in a virtualized environment. In that case for instance, most of the memory management is done by the guest operating system itself and the hypervisor would be invoked for a page table update or DMA access.

The essential benefit from using paravirtualization therefore lies in lower virtualization overhead, significant performance gain and simpler VMM design. The sacrifice, however, is the inability to run legacy software and proprietary operating systems. The latter is definitely the biggest drawback of this approach, enforcing anyone who is determined to make use of paravirtualization, to deploy open source operating systems to all virtual machines. Figure 2.3 shows the general principle of paravirtualization. The most interesting thing is that the general concept puts the modified guest OS into privilege level 0, like a non-virtualized operating system would require, without assigning any explicit privilege level to the hypervisor. The actual implementation used in practice may however vary among companies providing paravirtualized solutions.

2.3.3 Xen

This section focuses on Xen hypervisor and some interesting details about its paravirtualization solution. Information provided here are mainly backed by the article [2]. Before

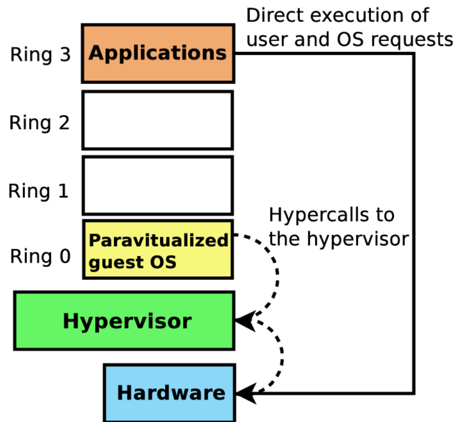


Figure 2.3: Paravirtualization concept.

immersing into the details, it is worth noting that Xen developers adopted a different term for a virtual machine – a *domain*.

Architecture

The authors of [2] state that the main goal of the Xen hypervisor design was to make it as simple as possible, providing only basic control which is available to authorized domains through an exported interface. The interface is called *control interface* and provides the ability to create and to destroy other non-authorized domains, as well as the ability to control the scheduling parameters and physical memory allocations, network interfaces and both creation and deletion of block devices. By default, there is one single authorized domain created at the boot time called *Domain0*. Figure 2.4 depicts both authorized and standard user domains, as part of Xen's architecture.

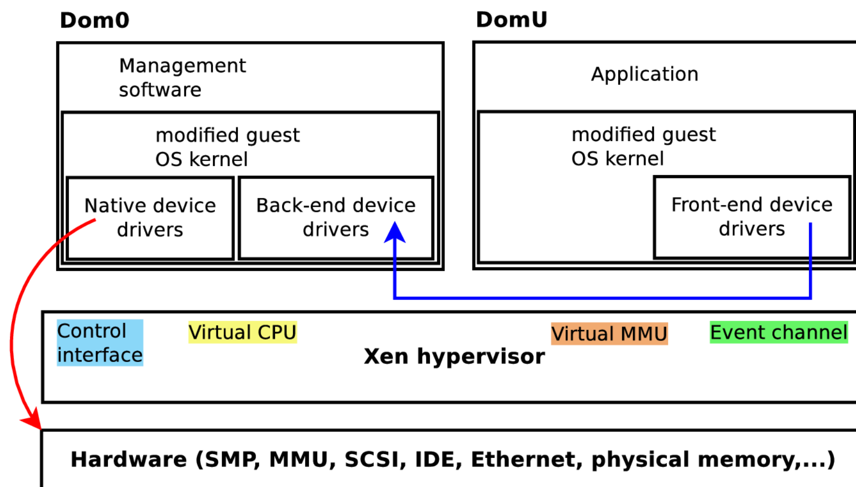


Figure 2.4: Xen architecture (modified original of [2]).

This domain is responsible for hosting the application-level management software which is connected to the control interface and enables *Domain0* to provide higher-level services (besides those mentioned above) like domain network activity monitoring, creating network filters and network traffic control (throttling included).

Interesting thing about domain-hypervisor communication is, that domains always communicate by executing hypercalls which is a synchronous type of communication. After a request completion, Xen returns control to the calling domain. Xen also implements a neat optimization to guest kernels to diminish the amount of hypercalls issued by queuing them and executing them in a batch. On the other hand however, Xen talks to the domains through an asynchronous mechanism which is meant to replace the usual delivery mechanisms for device interrupts. These events can stack in a per-domain bitmap and the guest operating system specifies an event handler which is also responsible for resetting the bitmap of pending events.

This was one of the first images what Xen looked like couple of years ago, according to [2]. Though the concept stayed the same, many new features have been introduced and besides paravirtualization, Xen also made use of the CPU extensions Intel and AMD added to their processors. Not only does Xen support full virtualization with unmodified guests at the moment, it can also combine both approaches.

I/O Virtualization

Following the information from [4], instead of providing physical devices, *Domain0* provides only virtualized views of them to other user domains. Since *Domain0* is privileged and does have full access to all the hardware below, it can export a specific subset of devices to each user domain depending on each domain configuration. As it was mentioned above, the user domain has only a virtualized view of the device. Xen calls this mechanism *class devices*, because each device falls to a certain category, being a block device, character device, network device, etc. As Figure 2.4 shows, the communication is conducted between frontend of the device located on the user domain and the backend which is located on *Domain0*. The communication itself takes place in memory, and Xen actually provides several mechanisms to accomplish this including shared memory, interrupts or event channels. *Domain0* then handles the I/O request, performs the operation on the actual hardware and propagates the results back to the user domain.

2.4 Hardware Assisted Virtualization—x86 Extensions

The main goal of the hardware assisted virtualization is to eliminate the need for CPU paravirtualization and binary translation techniques and to finally support the concept of classical virtualization. Both leading CPU manufacturers, Intel and AMD, provide hardware support for x86 CPU VMMs. Both are similar, but because most resources (including [1, 26, 31, 20]) turned their attention to Intel, this section further describes Intel's VT technology as presented in [26]. Intel's VT introduced two new modes of operation:

- *VMX root* – similar to original supervisor mode originally intended for the host, this one is intended for the VMM, and
- *VMX non-root* – which provides an alternative x86 environment, including most of the privileged and sensitive instructions, controlled by the VMM.

Both of these operation modes fully support all four protection levels, allowing the guest operating system to run at its intended level, and the VMM to make use of multiple privilege levels. In other words, the guest operating system executes almost natively without VMM's intervention, including kernel services, as long as the system call itself would not lead to

critical instruction execution. The instruction set further distinguishes between instructions which would result into a virtual machine exit unconditionally, and those that can be configured to trigger virtual machine exit conditionally. This enhances the flexibility of a VMM that is able to specify instructions and events which result in a virtual machine exit using various control bitmaps. An example would be setting the *VMCS* to exit from non-root mode on guest operating system page faults, TLB flushes or address space switches in order to maintain the shadow page tables. Figure 2.5 depicts transitions between these two operation modes which starts by executing `VMXON`, thus putting the processor into vmx root operation mode. The transition to *non-root* operation mode is illustrated by two different instructions:

- `VMLAUNCH` – used only on initial entry, and
- `VMRESUME` – used on all subsequent entries

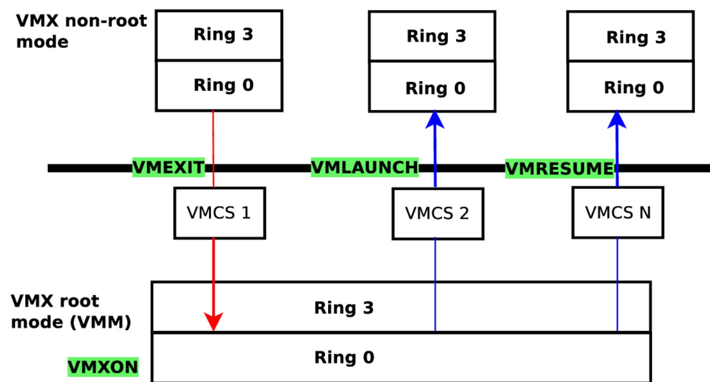


Figure 2.5: Intel VT: Transition between root and non-root operation modes [21].

The small blocks labeled as *VMCS* represent new in-memory data structure called *virtual machine control structure* which is logically divided into *guest-state area* and *host-state area*, each containing fields corresponding to different components of the processor state. Virtual machine entries (either VMLAUNCH or VMRESUME) load the CPU state from the guest-state area, whereas virtual machine exits save the CPU state along with detailed information specifying the reason of the exit to the guest-state area (using dedicated diagnostic fields) and load the CPU state from host-state area instead.

2.4.1 KVM

Kernel Virtual Machine (KVM) is a feature of Linux allowing the host Linux system to act as a type-1 hypervisor itself (according to definition in Chapter 2). As a result of integration of the hypervisor into the host Linux kernel as a loadable module, massive management simplification and performance gain can be achieved. Rather than taking the other path that involves creation of a new small featured kernel, the approach the KVM developers took, brought a number of benefits. Being represented as a regular Linux process, being scheduled by a conventional Linux scheduler, each virtual machine is thus able to profit from all new features Linux kernel provides. From the implementation point of view, KVM currently supports all main architectures including x86, IBM's PowerPC and ARMv8 (also called AArch64). As with the Xen hypervisor, the goal of this section is to provide a light insight to the hypervisor concept, so the reader can put the hypervisors into an overall

architecture comparison. The section relies mainly on information delivered by [14], [10], and [9].

Architecture

As section 2.2 stated, x86 architecture was not designed with virtualization support in mind. KVM, as a full virtualization and both type-1 and type-2 hypervisor representative, relies completely on the hardware support provided by Intel VT and AMD's SVM technology.

At the center of the architecture, there is a character device named `/dev/kvm`. The operations which this device exposes to the user-space include:

- virtual machine creation,
- virtual machine memory allocation,
- virtual CPU register reading/writing,
- injecting an interrupt into a virtual CPU, and
- running a virtual CPU.

Running a guest operating system (including its own user and kernel execution modes) is not entirely possible within user-space process. Therefore KVM introduced a new execution mode – *guest mode* in which the virtual machine, unless an I/O request or an external event[4] occurs, runs uninterrupted. Figure 2.6 depicts the overall architecture. KVM does not provide any hardware emulation, thus it is always used in conjunction with a hardware emulator such as QEMU for instance. When QEMU is preparing to start a guest, it sends various requests for hypervisor-specific functions through `/dev/kvm` to KVM. Once the preparation phase is finished, QEMU instructs KVM to start executing the guest system.

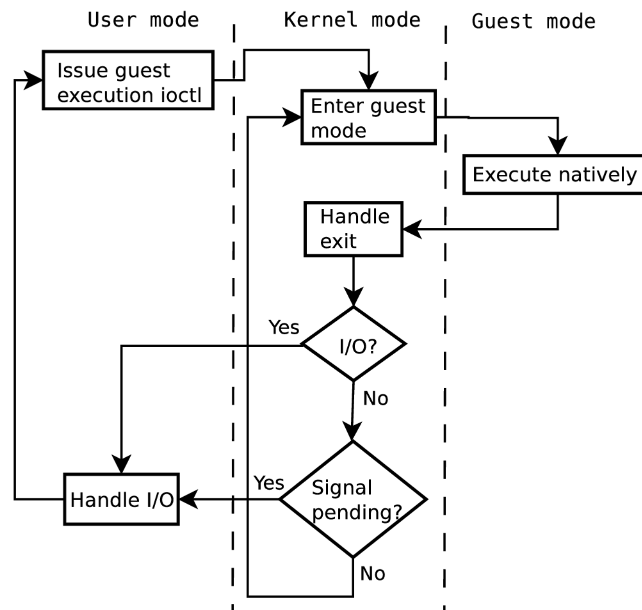


Figure 2.6: KVM architecture [14].

[4]Such event might be shadow page table fault, network activity or a timeout.

KVM Paravirtualization

The need for paravirtualized devices comes from the slow nature of I/O access, because it requires the guest to exit the native execution and letting the emulator handle the request. The standard approach in KVM is to handle the I/O requests in user-space where the emulation takes place. Most often, this is achieved by employing QEMU to simulate the behaviour of I/O. For these purposes, QEMU adopted a virtual I/O interface called *virtio*. Section 2.4.3 delivers an overview about the virtio virtualization driver.
The base concept could be compared to Xen's approach - all the guests need to support virtio[5]. Since the host implementation of virtio is in QEMU, the host itself does not need any virtio driver. It is worth noting, that virtio does not eliminate the need for emulation (like Domain0 touching the hardware directly does), it provides a significant performance improvement of network and disk operations.

2.4.2 QEMU

QEMU (*Quick Emulator*) is a machine emulator capable of emulating several CPU architectures, for instance x86, ARM, PowerPC, and SPARC. It operates in two modes – full system emulation and user process emulation. When used with the former, full system emulation, QEMU emulates all aspects of the machine, including a CPU, MMU, graphical adapter and peripherals. In the latter, user mode emulation which is only supported on Linux powered hosts, QEMU launches Linux processes executing binaries for CPU architectures that are not native to the host. The emulation follows principle already mentioned in section 2.3.1 – interpreting each guest instruction, translating it to the host's architecture and producing translated block of code which is then executed. Like [1] describes for VMware, to overcome the performance overhead to the full extent possible, QEMU utilized a translation cache for the most recently used translated block in the same way.

The concept of the translator, which is the most crucial part of the whole emulation process, was different to what QEMU uses nowadays. According to [3], QEMU favored the idea of lightweight portability to new architectures and general maintenance since the very beginning and presented it as an advantage over the competition. Rather than having multiple translators for all known guest-host pairs, QEMU based its design on *micro-operations*. Each guest instruction was then split into one or more micro-operations represented by small C functions [3, 27, 16]. QEMU then utilized host's GCC compiler to produce object files holding native code for each micro-operation. Omitting details about the complex internal process of translation, QEMU translated each guest block to a string sequence of micro-operations, the native binaries of which were then linked together to produce a translated block[6].

Later versions of GCC turned out to be problematic to comply with the design, so QEMU replaced its translation engine with *Tiny Code Generator* (TCG) which is used to date. TCG is a just in time compiler working over a small set of operations coded in an intermediate representation. According to [11], when a block of guest code is fetched, a translator, conforming to guest architecture, translates it into TCG intermediate code.

[5] As for Windows OS, there are available virtio drivers to install; all Linux kernels newer than 2.6.25 support virtio.

[6] Micro-operations used to appear in many translations, thus they existed in several places across the memory. Since micro-operations were generic enough, issues related to relocation of external code and variables occurred. For this purpose, QEMU used a tool called *dyngen* to analyze the relocation records and resolve the issues.

TCG then proceeds with some simple optimizations of the code and eventually translates the intermediate representation into native code. TCG is host specific (the architecture QEMU runs on top of) which is quite the opposite to the original concept, since for every target host architecture that should be supported, a dedicated piece of code has to be added.

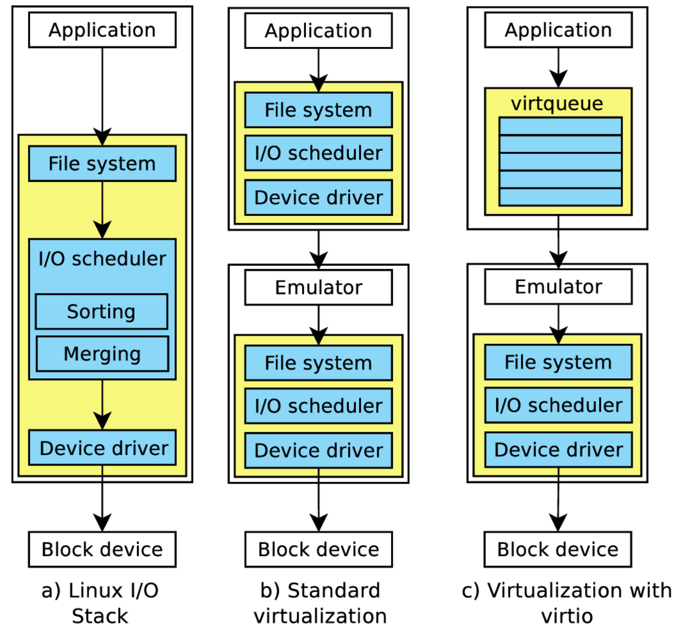


Figure 2.7: Comparison of virtio with standard Linux I/O stack and duplicated I/O stack in virtualization [33].

2.4.3 Virtio

Section 2.4.1 already prefaced the performance issue caused by I/O requests. With virtualization however, the performance degradation is even easier to observe. This is due to duplication of so-called Linux I/O stack which, in broad overview, consists of a file system controlling the data, an I/O scheduler sorting and merging I/O requests[7], and a device driver accessing the hardware device itself as depicted by Figure 2.7a. In virtualization environment, an I/O stack exists in both the host and the guest (Figure 2.7b), thus every I/O request generated by a process in the guest causes a trap to the emulator, and is further propagated to the I/O stack of the host. Since a guest system is nothing else than a user-space process, I/O requests generated by this process are handled the same way as requests from other processes in the host – gathered and rescheduled by host's I/O scheduler. It is therefore clear that I/O requests coming from the guest are scheduled twice.

Virtio on the other hand, is exploited by the virtual machine as a separate device driver. It is generic enough to support various drivers such as network driver, block device driver, and memory ballooning. As illustrated by Figure 2.7c, the I/O stack in the guest is replaced by *virtqueue*. The virtqueue is shared between host and guest and each time an I/O request is placed into the queue, the execution is returned back to the host (CPU mode is changed

[7] Sorting reorders I/O requests to reduce the seek time of physical disks, while merging reduces the number of requests by combining the adjacent ones. [33]

from guest mode back to host mode) which then handles the I/O request. Once the request is complete, host inserts a reply into the virtqueue and switches the CPU mode back to guest mode. Following this principle, the overhead of I/O stack duplication can be reduced significantly.
This section was backed by [33]. The article also provides details about the actual performance of virtio, since number of experiments and benchmarks using various I/O schedulers have been made.

2.5 Container-based Virtualization

Compared to the other types of virtualization described earlier in this chapter, there is a significant difference with containers – there is no hypervisor virtualizing all the underlying hardware for a guest operating system. Instead, they run on top of the same shared operating system kernel, running one or more processes within themselves. But what is a container? There is actually no rigorous definition, but from the interpretation of [8, 17], one can understand that a container is a conventional Linux user-space process. Because running a set of processes over shared kernel avoids the hardware abstracting overhead, this kind of virtualization is often denoted as *lightweight* approach. Continuing with the lightweight essence of containers, they also enable higher density of virtualized instances which became the keystone of most *Platform as a Service – PaaS* cloud solutions available today. The shared host kernel is also one of the disadvantages of the containers, because the kernel is exposed to the containers which might pose an issue for multi-tenant security.

There have been quite a few container-based solutions on the market for several years, including *Linux-VServer, OpenVZ, Solaris Zones, BSD Jails*, etc., but this section focuses on the most recent approach taken in this field, based on kernel support for *namespaces* and *cgroups*. Linux manual page[8] provides an exact definition of what a namespace is. It says, it is a mechanism wrapping a global system resource in an abstraction creating an illusion for all processes within the namespace of having an exclusive isolated instance of the global resource, which is why containers have no visibility or access to objects outside the container. Namespaces is a large topic worth an independent article, so description of its internals is out of scope of this thesis.

The other mechanism mentioned, cgroups, is a subsystem used to limit and isolate resources usage for a group of processes, thus for instance, allowing a container to be resized just by changing the limits of its corresponding cgroup.

In terms of security, considering isolation as mentioned above, the container is prohibited to interact with the outside of its isolated environment. However, there is a subset of system calls which are not aware of the namespaces and thus potentially posing a security vulnerability. By using *seccomp* library, the significance of this threat can be diminished. What it does, is that it allows a process to specify a list of system calls it is allowed to perform. Should the process try to make a system call it is not allowed to make after entering this “secure state”, `SIGKILL` signal will be delivered to it. This system call blacklisting is done at an early stage of starting a container, so that any application that is supposed to run in the container is affected by this change.

[8] http://man7.org/linux/man-pages/man7/namespaces.7.html

Chapter 3

Libvirt Project: The Virtualization Library

This chapter will discuss libvirt virtualization toolkit, its fundamental goal, its purpose in the virtualization world, libvirt's driver-based architecture, and the difference between client and server side drivers. As part of description of libvirt's internals, libvirtd daemon and aspects regarding libvirtd, including libvirtd's purpose, communication with the daemon, client request handling, task-based model, and data serialization/deserialization will be addressed. The contents of this chapter is backed by information obtained from libvirt's documentation [28], as well as from article [5].

3.1 Libvirt's Objective

Motivation

Disregarding virtualization types, as described in Chapter 2, every virtualization solution offers a set of programming interfaces to corresponding operations to be used with their virtual machines. Lack of such management programming interfaces would render any solution useless. Although there was an effort to create a standard for managing virtual environment, eventually resulting in *Virtualization Management* (VMAN) standard[1] which was published by American National Standard Institute, it has not been adopted by the virtualization market yet. Thus, building universal compatible virtualization management solutions supporting arbitrary virtualization environments is a rather complicated task. As authors of [5] state, this eventually resulted in different management solutions, especially commercial ones like VMware and Citrix, tailored and optimized to specific hypervisors. Some open source projects trying to stay hypervisor-independent like oVirt and OpenNebula came up as well, but they needed to find a way to overcome the problem of management programming interfaces incompatibility. To make it even harder, some virtualization solutions tend to change application interface between releases, breaking any backwards compatibility completely. Even kernel might change the structure of some of its entities, like cgroups, with newer versions. All of these aspects needed to be taken into consideration.

[1] http://www.dmtf.org/standards/vman

Goal

To fulfill the aspects mentioned in the previous paragraph, a management tool would either require complex internal modifications or make use of a middleware layer which would take care of these inconsistencies and provide a stable abstract interface. Following the latter, the goal of libvirt virtualization toolkit is then to provide such an abstraction middleware layer that deals with all underlying hypervisor specifics transparently and is sufficient to securely manage virtual machines, or in terms of *libvirt* — domains on a host node. In terms of stability, it is worth noting that all application interfaces libvirt library exposes strictly follow a philosophy of a long-term stability which also can be accounted a crucial benefit in favor of the management solutions using libvirt.

3.2 Libvirt as a Middleware Layer

The information from previous Section 3.1 still does not clarify libvirt's place in the application hierarchy though, therefore Figures 3.1 and 3.2 are both meant as an aid to better understand it. The former, depicting the basic concept of libvirt being an intermediary for some userspace management tools to communicate to various hypervisors or virtualization solutions only serves as an introduction to the latter which is more complex, so the following paragraphs divide Figure 3.2 into individual scenarios and describe each one separately.

Management Tools and Libvirtd Daemon

Figure 3.2 depicts communication with three different virtualization types as per information provided by Chapter 2. Consider a management tool, like virsh or OpenStack, that, given the specific use case, invokes an application interface designed to achieve the requested operation. The library needs to know what kind of hypervisor, emulator, or container solution it should communicate with. This is because not all hypervisors support remote management, so a daemon that implements remote communication, among other functionality, is needed at the remote side. This daemon is called libvirtd and in most cases, the library will direct the communication through it. By further analyzing Figure 3.2, it can be seen that libvirtd (top left) then uses the same library in order to achieve the requested effect of the operation. In fact, when libvirtd uses the library, it invokes the very same library method as the original caller on the client side did. This is possible due to libvirt's driver-based architecture that allows the daemon to associate a different driver with accomplishing the operation than the client is allowed to use. The driver-based architecture and details regarding the communication with libvirtd daemon are further addressed by Sections 3.3 and 3.4 respectively, and in context of this section and Figure 3.2 are irrelevant.

Managing Xen

Depending on the nature of the operation and virtualization type involved, libvirtd then uses libvirt library in variety of scenarios. The first scenario, as Figure 3.2 depicts (top right), involves a bare-metal hypervisor – Xen in this case – which is then responsible for contacting the privileged domain (refer to 2.3.3 for Xen's architectural details) and performing the operation.

Managing Containers

The second scenario involves operating system level containers (bottom right). Recalling Chapter 2, all containers, independent of any specific solution, share the host's kernel. Since many configurations on containers are achieved via cgroups editing, libvirtd can then use libvirt library to edit cgroups, or even talk to various kernel modules or drivers, which might include SELinux, network drivers, device drivers, etc. The other, more straight-forward and expected way to use it, is to invoke a method of the container engine.

QEMU-KVM

In the last scenario (bottom left), libvirt can still be used to manipulate various kernel entities, but the focus should be directed to QEMU emulator. Although the hypervisor used in this scenario is KVM and QEMU is only responsible for I/O emulation, the philosophy of KVM is to keep the module as simplest and smallest as possible and either let the guest run uninterrupted or hand the control over to QEMU (red arrows originating from KVM). So besides scanning for presence of `/dev/kvm` device and querying for KVM's capabilities, only to later format them as QEMU command line options, libvirt never talks to KVM. So whenever a client requests an operation over a domain to be carried out, libvirt talks to QEMU via a monitor QEMU exposes.

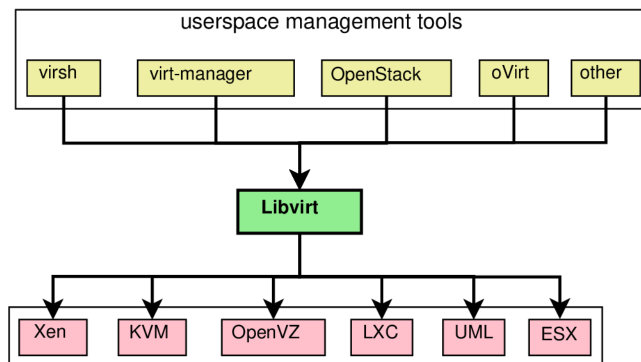


Figure 3.1: Libvirt connecting management tools to various virtualization solutions.

3.3 Libvirt's Architecture

Originally, libvirt project started as a Xen wrapper[2] rather than a unified management library for different hypervisors. But its architecture allowed other hypervisors to be added later on, thus shaping it to its present form.

The architecture is conceptually divided into a hypervisor agnostic and several hypervisor specific parts, called drivers. The implementation of these drivers is exposed by a generic public API for the applications to use, which then maps to appropriate internal driver functions.

However, high diversity of hypervisors that libvirt supports inherently caused some issues that complicated the overall design, e.g. lack of remote management support and

[2] Xen, as an open source project, used to be very popular regarding its paravirtualization implementation. Since hardware-assisted virtualization and KVM introduction, the situation has slightly changed, though Xen still remains one of the major virtualization solutions available on the market.

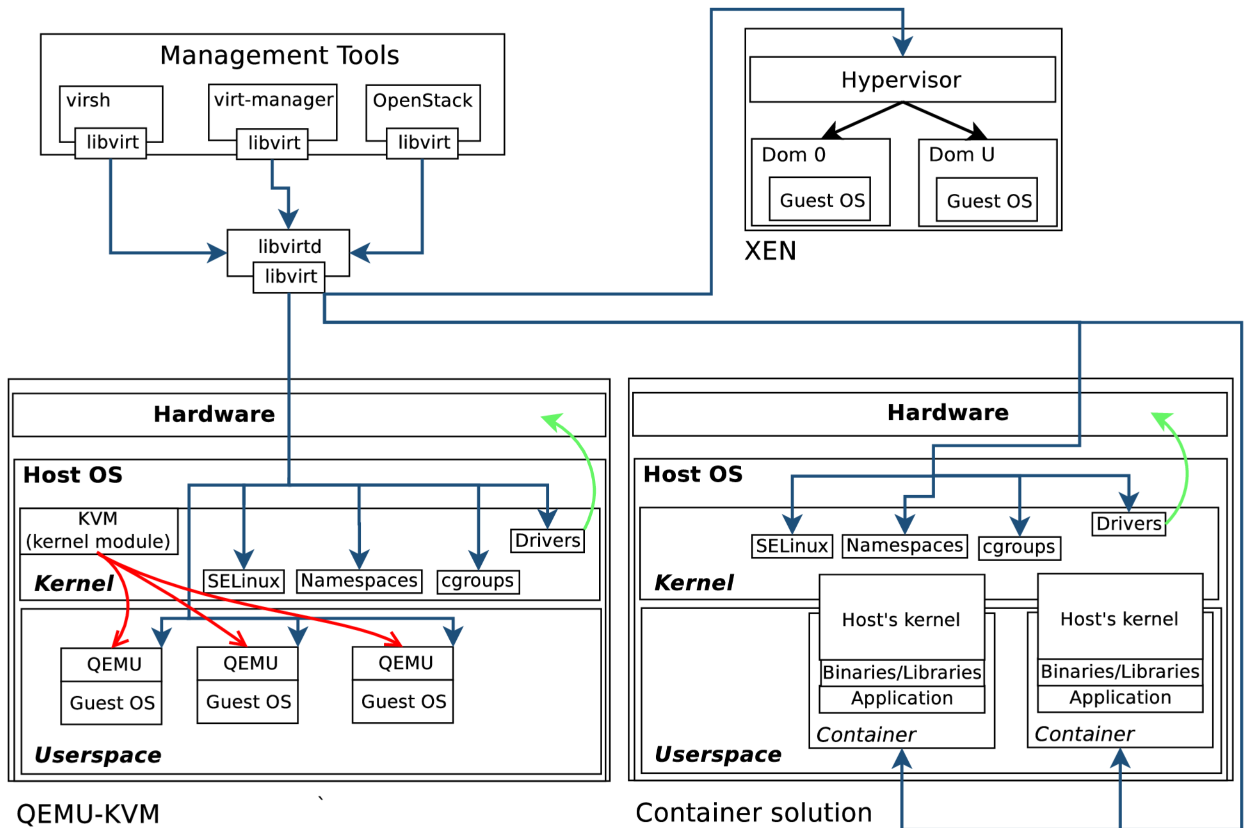


Figure 3.2: More complex view of libvirt connecting to different virtualization solutions.

an architecture which does not preserve the state of a domain, which means that although there might be an internal representation of a state available, it is volatile only and not preserved between restarts or in case of a crash. As Section 3.2 already prefaced, libvirt tackled this issue by implementing a client-server model where libvirtd daemon implements the server side (Section 3.4 provides more details about libvirtd daemon).

3.3.1 Stateful and Stateless Drivers

Not all hypervisors however, are compatible with this remote management daemon concept. These are mostly proprietary, closed source solutions which do expose their own remote management interface. Because of this, libvirt drivers are divided into two disjoint sets, the first one comprising of client-side only drivers and the second one containing server-side (daemon-side) drivers. As it was mentioned, since most proprietary solutions do not need a daemon to tunnel remote connection and the hypervisor is capable of restoring a running domain's state after a crash, it is understandable that the client-side group of drivers consists of such drivers[3]. Libvirt often addresses this set of drivers as *stateless*, because it is the virtualization solution itself who is responsible for restoring a domain that crashed to its last known state prior to the crash. Conversely, daemon-side only set of drivers consist of drivers that represent hypervisors or virtualization solutions which need an intermediary to tunnel a remote connection, as well as to preserve the domain's state —

[3] At the moment, the current set of client-side only drivers, excluding remote driver and Xen, corresponds to following proprietary hypervisors: Microsoft's HyperV, VMware's ESX, IBM's phyp and Parallels.

for this reason libvirt refers to these as *stateful* drivers. The last driver to cover is the hypervisor-agnostic remote driver which extends the client-side set of drivers. The purpose of the remote driver is to offer the client a mechanism to establish a remote connection to a libvirtd-managed hypervisor and provide an encapsulation of the communication between both sides for the whole lifetime of the connection. It is clear that all the other client-side drivers need to implement communication with the hypervisor on their own, according to specifications provided by the hypervisor. The situation regarding both types of connection, using stateless drivers and tunneling through remote driver, are depicted on Figure 3.3.

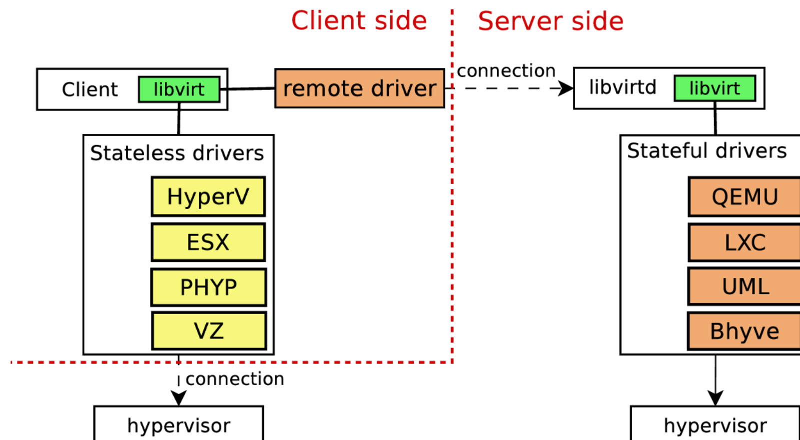


Figure 3.3: Hypervisor connection difference using stateless drivers and remote driver

3.3.2 Connection Establishment

Before any communication with a hypervisor takes place, a connection must be established first. This was already prefaced by section 3.3.1 and Figure 3.3. The choice of driver used for connection and further management is transparent to the client and libvirt does it automatically. Client however needs to instruct libvirt in which driver should be probed for connection establishment. This is achieved via a connection URI which follows the pattern below.

```
driver[+transport]://[username@][hostname][:port]/[path][?extraparameters]
```

The decisive part for libvirt to choose a driver to open a connection with is the URI scheme[4]. Should the scheme be not recognized by any stateless driver, remote driver is selected. From implementation point of view, a client establishes a connection to hypervisor by initiating `virConnectOpen` call, specifying a fully qualified URI[5] to the hypervisor. Figure 3.4 illustrates a communication between virsh client and libvirtd daemon. In this case, virsh specifies `qemu://host/` as the fully qualified URI. Since QEMU is managed by libvirtd, as explained in previous section, no stateless driver will be able to recognize the URI schema, thus, remote driver will be used. Recalling Figures 3.2 and 3.3, it should be noted that the library used by a client and the one used by libvirtd daemon may only differ in their version number which has an impact only on the number of features both sides support, but has no practical impact on the communication itself. That means when

[4] https://en.wikipedia.org/wiki/Uniform_Resource_Identifier
[5] `libvirt.conf` allows setting up URI aliases for frequently used URIs.

a request to establish a connection arrives to libvirtd, an identical call to `virConnectOpen` is issued, but this time using one of the stateful drivers. In this case QEMU driver is used. Libvirtd returns the response from QEMU driver and the initial handshake between virsh and QEMU emulator is then complete.

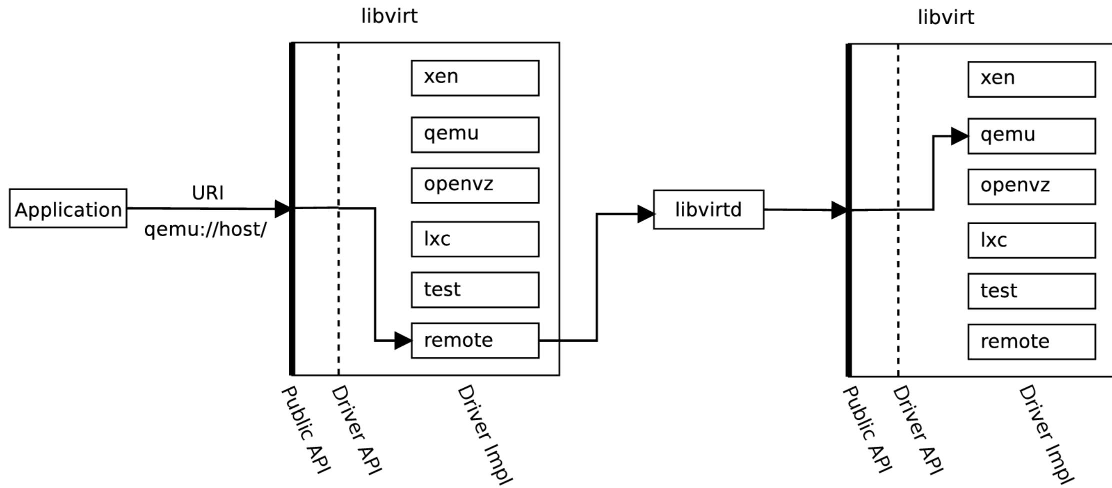


Figure 3.4: Connecting to QEMU emulator using the remote driver [28].

This driver-based architecture proved to be very flexible, so several other drivers, including storage handling and network management have been added, as Figure 3.5 demonstrates. Focus should be moved towards storage driver, because this particular driver is, to some extent, different from other drivers. And that is because it is further divided into a number of sub-entities, called backends, each responsible for different storage implementation. This way, libvirtd is able to manage both local and remote storage technologies, including SCSI, iSCSI, Ceph, Sheepdog, LVM based storage, etc.

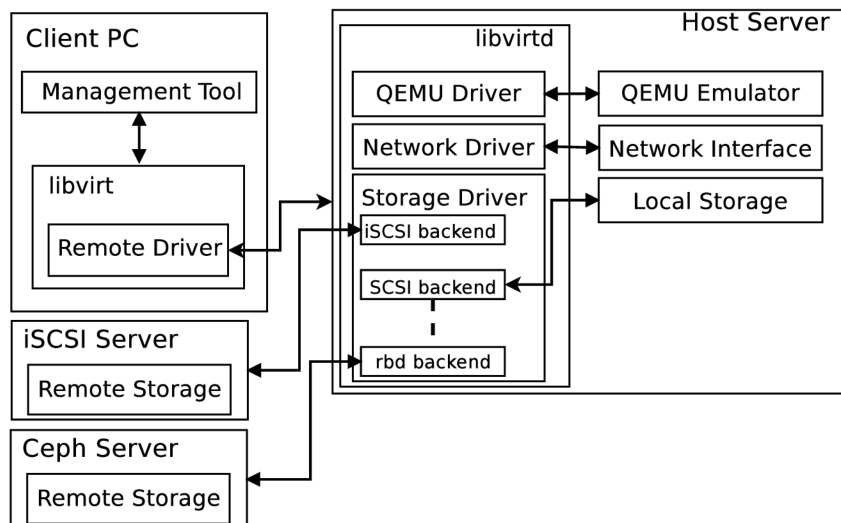


Figure 3.5: Driver based libvirt architecture [5].

3.4 Libvirtd Daemon

This section focuses on libvirt's daemon – libvirtd – it's purpose, communication details with a client, and task-based model used to handle client requests.

As Section 3.3 prefaced, there are two main reasons for having a daemon as part of libvirt library. First one, not every hypervisor supported by libvirt provides mechanisms (application interfaces) for remote management. Secondly, some hypervisors (virtualization solutions) do not provide users with mechanisms to safely store virtual machines states. That way, in case of a crash, not only the virtual machine is not restarted automatically, all configurations related to such a virtual machine are lost. Libvirt's daemon implements features to address both of these issues and following paragraphs are dedicated to explaining the details.

3.4.1 Communication

Based on previous sections, it can be easily noticed that libvirt utilizes client-server model of communication, using request-response message-passing system. Libvirt made a decision to use RPC as communication technique. This fact, however, is completely transparent to the client. What each client has to do when establishing a connection to libvirtd, is to specify whether the desired connection is supposed to be local only or remote, using a transport layer. But this is only a basic bisection, in fact, libvirt does support a range of transports, including:

- *Unix sockets* – since these are available on local machine only, the communication is not encrypted and libvirtd uses Unix permissions and SELinux for access control,
- *TLS* – daemon uses an authenticated and encrypted socket, listening on a public port number,
- *ssh* – communication is conveyed over ssh connection using OpenSSH binary,
- *libssh2* – like the classic ssh protocol transport, but uses *libssh2* library instead of OpenSSH binary, and
- *TCP* – in this case, daemon would use an unencrypted TCP/IP socket, thus, this type of transport is not advised for production use.

Enabling a specific transport for a connection is only a matter of changing schema part of the connection URI.

Recalling Figure 3.4, virsh client does not specify any additional transport details in the connection URI's schema, thus, local connection is requested. Should a client request a TLS transport for instance, the schema would need to be changed in accordance with Listing 3.1, where `<driver>` is the name of hypervisor requested. However, it should be noted, that in order to use TLS transport, client and server authentication certificates have to be generated first.

```
<driver>+tls://host
```

Listing 3.1: Libvirt's generic connection URI format

So far, previous sections described how clients connect to hypervisors managed by libvirtd daemon via libvirt's driver mechanism. Naturally, the process of connection establishment is more complex and some internal details about daemon's architecture have been purposely neglected. For most clients connecting to libvirtd, in order to perform some domain management tasks, this sort of information is transparent and unnecessary to know. However, the daemon's architecture needs to be further explored to a degree necessary to understand how the administration interface, explained by Chapters 4 and 5, is designed.

Although libvirtd does represent the server in the client-server model used by libvirt, the architectural detail is that libvirtd implements a server object, the only responsibility of which is to accept clients' connections. In fact, libvirtd's architecture is flexible and modular enough to implement multiple servers contained within itself[6]. However, only one server is currently enabled within libvirtd daemon at the moment, with another one – administration server – waiting to be enabled once the administration interface is ready to be released. Each server then implements multiple service objects that basically can be divided into two groups, services that accept local connections (i.e. connections to UNIX sockets) and services that accept remote connections. It is actually the service, that operates the socket which clients connect to, only to then pass the client to the server for further processing, e.g. to determine whether the client is trying to connect to a valid hypervisor. A service can also be responsible for a client's authentication, if an authentication method is allowed in libvirtd's configuration file. Figure 3.6 summarizes the details mentioned in this paragraph, with two servers contained within libvirtd, the original one – now called *libvirtd* – and the second one called *admin*. As confusing as naming a server exactly the same as the daemon itself might seem, this decision was made to reflect the fact that this is the server clients were connecting to the whole time transparently. Figure 3.6 also illustrates different types of services, remote ones that may or may not support traffic encryption, possibly authentication as well, and local ones that operate on sockets with different access permissions.

When a client is then trying to establish a connection, depending on the service, a new server-side client representation is created, along with some identity information libvirt was able to gather.
Next section will provide information regarding the whole process from receiving a request, through extracting data from RPC, to actually performing the task.

3.4.2 Message Processing and Task-based Model

Every client request that arrives to libvirtd is subject to several stages of processing which include multiple levels of dispatching the raw data received on a socket, through creating a libvirt job and placing it into a queue, deserializing the data and finally invoking the hypervisor-specific driver to accomplish the task. Although important, a detailed description of libvirtd's internal message processing implementation is out of scope of this thesis, but information found libvirt's documentation [28], as well as code introspection may be helpful resources to acquire all the necessary details. However, it is still important to provide a general insight on the mechanism used to execute a task.
It is obvious that with growing number of connected clients, serializing tasks and executing them sequentially may have a significant impact on performance. Libvirt therefore supports

[6] Although libvirtd has been the only daemon mentioned so far, libvirt distinguishes and supports two other daemons compatible with this design, namely *virtlockd* and *virtlogd*. Both however, are out of scope of this thesis and will not be further addressed.

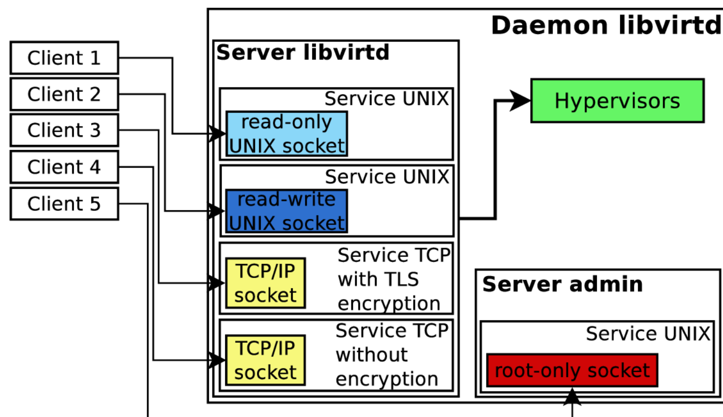


Figure 3.6: Another view of how client is connected to libvirtd daemon

concurrent task execution. Because of the nature of operations performed, some tasks are not allowed to be executed at once over the same resource, i.e. operations modifying internal state of a domain will queue on a domain's lock. But in context of message processing, this fact is irrelevant. The concurrent execution is achieved by utilizing a threadpool, or as libvirt refers to it – a workerpool which is signalized each time a task, which holds the procedure identification to be invoked and the data to be passed to the procedure, was placed into the queue. But since the data the task holds are still raw the last stage of processing needs to be performed. First available worker that removes the task from the queue invokes a procedure-specific dispatcher which is responsible for deserializing the XDR format used for the data and finally executes the task, passing the deserialized data as an argument. As is was mentioned above, by neglecting the fact that not all tasks can be executed concurrently, a theoretical hypothesis, that the only limiting factor to the performance besides hardware is the actual number of workers in a threadpool, can be formed.

Workerpool Limits

Libvirt does not use a constant number of workers, it is rather dynamically increased, which means that when a task occurs and all workers are currently busy with some time-consuming operations, a new worker is created within the threadpool, so that the task can be carried out. Spawning too many threads can however pose a significant performance drop for the whole host system, so each threadpool implements limits[7] to maximum and minimum number of threads that can be active in a threadpool. What this does is, that before a worker can be created, first the current number of workers is confronted with the allowed limits. If no other worker can be created, the task stays in the queue until the first available worker takes it out. However, it may happen that the maximum limit of workers has been reached and all workers are occupied with a task requiring communication with a hypervisor. After instructing the hypervisor to accomplish an operation, libvirt waits for a response. But because libvirt cannot guarantee that such a response will always arrive, typically if something goes wrong within the hypervisor, the task may hang, possibly making a domain unresponsive, in which case the domain needs an intervention by performing a hard reset. But since all workers might be occupied with executing a task or by waiting for a lock to access the very same domain, there would not be any available workers to

[7]These limits can be configured in libvirtd's configuration file `libvirtd.conf`

perform such a critical operation. For these purposes, the threadpool is divided into a set of ordinary workers, as described above, and a set of workers dedicated to operations which do not rely on communication with a hypervisor, making libvirt able to guarantee that such a task would always finish. Libvirt refers to these workers as priority workers and the amount of them is constant, i.e. does not adjust in any way during libvirtd's execution. It is obvious that in order to utilize this model, each procedure available through libvirtd must be tagged with high or low priority with a further constraint that a high priority operation is guaranteed to always finish. That way, an ordinary worker is still able to perform a critical operation, but a priority worker is only allowed to perform high priority operations.

Chapter 4

Administration Interface Specification

This chapter describes the specification and design of administration interface. Before examining the specification to various programming interfaces which are part of the administration interface itself, the chapter first justifies the need for such an administration interface. It then continues with a broad overview of the interfaces an their typical use cases, including some practical examples.

4.1 Objective

This section is going to present the motivation behind the administration interface. All libvirtd's configurations are achieved through a configuration file dedicated to this purpose[1]. Libvirt refers to such a configuration as persistent, which means that whenever libvirtd is restarted, its configuration is loaded from the configuration file and remains the same for the daemon's lifetime with no current mechanism to change its runtime configuration. The idea of an administration interface originated from the following aspects in production environment:

- load balancing,
- runtime introspection (using dynamic logging settings), and
- client management.

4.1.1 Load Balancing

Having all workers occupied during a server load causes a delay in delivering a service to customer. By having the ability to control the number of active workers on a server dynamically, without having to restart libvirtd each time the configuration needs to be changed, system administration becomes more convenient as the configuration change can be automated.

[1] `/etc/libvirt/libvirtd.conf`

4.1.2 Runtime Introspection

Occasionally, domains experience failures and a troubleshoot is necessary. By allowing the administrator to tweak debugging settings on demand, certain anomalies in daemon's behaviour can be inspected in current session as there are rare occasions where problems disappear with daemon restart.

4.1.3 Client Management

Allowing the administrator to access information regarding connection details of currently connected clients, possibly logging their activity may further result in integration of a policy mechanism. Depending on the policy itself, certain clients may be restricted from the access to libvirtd or certain domains it is managing, including the ability to force close an existing connection of a malicious or long inactive client. The problem with inactive clients is caused by the maximum limit to the number of clients connected to the daemon. So either the limit is increased or/and these inactive clients are disconnected forcefully, while the decision for disconnecting a specific client still being a third-party management application's responsibility.

4.2 Interface Overview

The goal of this section is to provide an overview of the interfaces being designed and their typical use cases. Examples shown in this section utilize libvirt's console administration program `virt-admin`. Application source code examples demonstrating the sequence of calls necessary to issue are shown in the Appendix B.

The major components of the administration interface that are going to be covered by this section include:

- load balancing - tweaking the number of available workers on a daemon's server,
- runtime introspection - changing parameters of a daemon's logging subsystem, and
- client management - listing clients connected to the daemon, setting limits to number of clients allowed, and disconnecting a specific client from the daemon.

Load Balancing

As Section 4.1.1 already prefaced, server load can cause delay in service delivery. This can especially be the case if the daemon's configuration is left unchanged (on default), rather than modified and tailored to the specific usecase. If such a situation occurs, chance is that the maximum number of active workers is too low. Increasing the value will allow more tasks to be accepted and accomplished. Note however, that increasing the number of threads too much can have rather opposite effect to the desired one, since the hardware might not be able to manage and switch between threads without affecting the overall performance of the system.

Workers are always modified on per-server basis, therefore the administrator first needs to know which server suffers from the load the most. To find out what servers are available on the daemon, the command depicted on Listing 4.1 is issued.

The administrator can then inspect the current state of a workerpool and later increase its limits by issuing the commands shown on Listings 4.2 and 4.3

```
$ virt-admin srv-list
 Id    Name
---------------
 0     admin
 1     libvirtd
```

Listing 4.1: Retrieving the list of servers

```
# virt-admin srv-threadpool-info libvirtd
minWorkers     : 5
maxWorkers     : 20
nWorkers       : 5
freeWorkers    : 5
prioWorkers    : 5
jobQueueDepth  : 0
```

Listing 4.2: Retrieving workerpool attributes from libvirtd server

```
# virt-admin srv-threadpool-set libvirtd --max-workers 40 --prio-workers 10
```

Listing 4.3: Modifying workerpool attributes on libvirtd server

Runtime Introspection

For the next paragraph, consider a domain that is misbehaving and only errors are being logged. Libvirt recognizes four levels of logging in the following order: *debug*, *info*, *warning*, and *error*. The order in the list is important, because the levels compose an inclusive hierarchy - each level also includes levels ordered further in the list, with *debug* including all the other levels, e.g. setting the logging level to *warning* means that both warnings and errors will be logged. To inspect the domain's behaviour thoroughly, the daemon needs to be configured to log all debugging information. Following virt-admin commands show current logging setting retrieval, as well as changing the current global logging level to the desired value.

```
# virt-admin dmn-log-info
Logging level: error
Logging filters: 4:event 3:json 3:udev 3:util.object
Logging outputs: 1:file:/var/log/libvirt/libvirtd.log 3:stderr
```

Listing 4.4: Retrieving all logging settings from libvirtd daemon

```
$ virt-admin long-define --level 1
```

Listing 4.5: Changing global logging level to debug

Debug information is definitely useful for troubleshooting, some modules (like RPC or `virobject.c` for instance) however, can be extremely verbose and produce fair amount of log output which, in turn, can be overwhelming when investigating an issue. For these purposes, libvirt supports logging filters which are specified in per-module manner and

override the global logging level. Listing 4.6 shows the format of a filter, with `level` being the maximum logging level for a module (overrides the global setting) and `module_name` being related to libvirt's source tree hierarchy, e.g. if a source module is located in directory `util` and its name is `virobject.c`, then to define a filter discarding all messages other than warnings or errors from this module, Listing 4.7 shows the appropriate virt-admin command to be issued (note the "vir" prefix is omitted, which is a common libvirt practice with all modules):

```
level:module_name
```

Listing 4.6: Format of a logging filter

```
# virt-admin dmn-log-define --filters "3:util.object 4:rpc"
```

Listing 4.7: Defining a new set of filters to log only warnings and errors originating from virobject.c module and only errors from RPC

The default output for daemon to log to is `stderr`, but there are three additional different types of outputs libvirtd can log to: regular file, system log, and systemd's journal. Syntax of a logging output is demonstrated by Listing 4.8. As it can be clearly seen, it is very similar to filters, with the only difference being the pattern to define a new output—an additional field for file and syslog-based outputs is appended. Examples below show the additional bit of data to either be an absolute path to a file for file-based logging or an identifier prepended to every libvirt's message for syslog-based logging.

```
level:(file|syslog|journald|stderr):additional_data
```

Listing 4.8: Format of a logging output

```
# virt-admin dmn-log-define libvirtd --outputs "1:file:/var/log/libvirtd.log\
> 3:syslog:libvirtd"
```

Listing 4.9: Defining a set of outputs to log all messages to a file but only warnings and errors to the system log

Client Management

The last essential part of the administration interface is client access restriction, i.e. altering the number of connected clients. As with modifying the number of workers in a workerpool, the default number for maximum allowed connected clients might not suit every possible use case. Therefore, the administration interface provides a way how to change the limit during runtime, either to allow more clients or quite the opposite, to refuse any more connections.

```
# virt-admin srv-clients-info libvirtd
nclients_max            : 100
nclients_current        : 10
nclients_unauth_max     : 15
nclients_unauth_current : 2
```

Listing 4.10: Retrieving the current maximum number of connected clients to libvirtd server

```
# virt-admin srv-clients-set libvirtd --max-clients 150
```

Listing 4.11: Resetting the maximum number of clients connected

A situation may occur where the administrator has to restrict access to libvirtd daemon for some clients. Administrator then needs to know what clients are currently connected, then choose the desired one and close the connection to it forcefully. The problem though is that the client can reconnect the the daemon almost instantly and libvirt cannot prevent it from doing so, since libvirt does not have enough information about a client to put it on a blacklist. In fact, two clients connected to libvirtd are only distinguishable from each other in the way they are connected, i.e. either locally, using a UNIX socket or remotely, using a TCP/IP network socket. Although libvirt is able to inspect which user the client process belongs to, each user may utilize a number of clients, possibly all of them being connected in the same way (either remotely or locally). Thus, the decision of selecting a victim to disconnect cannot be made in libvirt; it is administrator's (or a management application) responsibility to implement a policy and take all necessary countermeasures to prevent a client (or a set of clients) from reconnecting to libvirtd daemon. It is worth noting that libvirt is only a tool to achieve a certain goal, not to analyze the conditions under which an operation should be accomplished. Listing 4.12 demonstrates how a list of currently connected clients can be retrieved.

```
# virt-admin client-list libvirtd
 Id     Transport         Connected since
------------------------------------------------
 1      unix              2016-04-15 17:12:06+0200
 2      tcp               2016-04-15 17:11:16+0200
 4      unix              2016-04-15 17:11:47+0200
```

Listing 4.12: Retrieving list of currently connected clients from libvirtd server

The output may be sufficient for a summary of how many and what kind of clients are connected to the server, however, a management application would typically require more detailed information about a particular client, for instance, what kind of authentication was used for this client, which username the client provided when authenticating, network socket address of the remote endpoint, and several other transport-dependent attributes. For this purpose, `virAdmClientInfo` interface has been designed which can be utilized from within the virt-admin client in the following way:

```
# virt-admin client-info 1 --server libvirtd
Id:                     1
Connected since:        2016-04-15 17:12:06+0200
Transport:              unix
User Id:                1000
User Name:              eskultety
Group Id:               1000
Group Name:             eskultety
Process Id:             12653
```

Listing 4.13: Retrieving identity information about a client connected through a UNIX socket

```
# virt-admin client-info 2 --server libvirtd
Id:                     1
Connected since:        2016-04-15 17:11:16+0200
Transport:              tcp
Socket Address:         192.168.10.1:16234
SASL Username:          eskultety
```

Listing 4.14: Retrieving identity information about a client connected through an unencrypted TCP/IP socket and SASL authentication

Now that the administrator has all the information libvirt has about a client, a victim to be disconnected can be then picked, and the connection can be closed forcefully by issuing `client-disconnect` command from within virt-admin, specifying the victim.

```
# virt-admin client-disconnect 2 --server libvirtd
```

Listing 4.15: Disconnecting client 2 from server libvirtd

This section provided a brief overview of the key components of libvirt's administration interface, how virt-admin can be used to accomplish a certain task, as well as mentioning typical use cases for them. For C language-based examples, these are both available on the attached optical medium, as well as in Appendix B.

Chapter 5

Implementation Details of Selected Parts of the Administration Interface

This chapter covers details and problems faced when implementing the following parts of the administration interface:

- the choice of data types,
- load balancing - managing a threadpool's attributes, and
- runtime introspection - management of a daemon's logging parameters.

This selection reflects the fact that although interfaces which implement listing available servers on a daemon, listing clients connected to a specific daemon, and getting identity information about those clients are also part of the administration interface and were implemented as part of the thesis, there were not any particularly troublesome issues that needed to be solved during implementation and therefore these are not covered by this chapter. This chapter also does not cover description of individual interfaces, since the complete API reference in Appendix A already provides description of all functions, as well as the exported macros.

5.1 Common Data Types

Before stepping into sections dealing with load balancing and runtime instrospection, the common data types that were designed to be exposed through the library should be addressed first, since as Appendix A shows, those are used throughout the whole administration interface. There are essentially only three data types exposed by the library so far, more specifically `virAdmConnect`, `virAdmServer`, and `virAdmClient`. What these have in common is that they are all client-side representations and essential for all accessors of larger server-side objects. A `virAdmConnect` object[1] represents an active connection to the libvirtd daemon and is obtained through invocation of `virAdmConnectOpen` method, which

[1]Although libvirt is a C library, it does employ object oriented programming model by implementing means that allow it to do so. Therefore libvirt refers to majority of its heterogeneous data types as classes, while referring to variables declared as these types as objects

is a crucial part of the whole administration interface, and its existence is a prerequisite to all programming interfaces listed in Appendix A. Although the `virAdmConnect` data type already existed prior to writing this thesis, other important data types are based on this type and thus should serve merely as an introduction to this chapter.

Server Object

Both client management and threadpool configuration take place within attributes of a daemon's server object. Despite having an active connection to the administration server, the client also needs a way of identifying which server it would like to work with. However, libvirtd treats all its servers as anonymous, so a form of identification had to be added first. From libvirt's perspective, several approaches to unique identification could be taken:

- a numeric ID,
- a 128-bit UUID string in canonical format, or
- a string name.

But since the major aspect of any programming interface should be convenience, a unique[2] name is the best choice, which unlike both the numeric ID and UUID provides users with the ability to distinguish two servers and their purpose from each other. `virAdmServer` type then pairs the server identification with the connection object just to provide a different level of abstraction.

```
struct _virAdmServer {
    virObject object;
    virAdmConnectPtr conn;
    char *srvname;
};
```

Listing 5.1: A private client-side server representation

Client Object

Analogically, for programming interfaces that, to some extent, manage client applications connected to the server, a client-side representation (a datatype) of a client object is necessary. Since a client object needs to be looked up within a server first in order to be used, this client-side representation also needs a server reference. As with server objects, clients need a unique identification as well (for the duration of a connection). But unlike servers, which are static and their purpose can be expressed with a single name, a client can be classified as a transient entity, i.e. it requests the daemon to accomplish a set of tasks and then disconnects. Hence, naming individual clients would render pointless, especially when two clients connected to the daemon from the same remote host, using the same connection transport method are, as Section 4.2 mentioned, from libvirtd's perspective indistinguishable. Therefore, using a 64-bit wide numeric ID turned out to be the only option.

But listing solely client IDs would not be much of a help to any user, thus more information like connection transport method (TCP, TLS, or a UNIX socket) and connection

[2] Libvirt internally exposes a mechanism to decline attempts to use a duplicate names

```
struct _virAdmClient {
    virObject object;
    virAdmServerPtr conn;
    unsigned long long id;
    long long timestamp;
    unsigned int transport;
};
```

Listing 5.2: A private client-side client representation

timestamp were added. There is more static data that a server stores about a connected client, but since the nature of that data depends mostly on the connection transport used by the client, a separate method `virAdmClientGetInfo` was implemented for this purpose. The final client-side `virAdmClient` data type is then defined according to Listing 5.2.

Typed Parameters

Apart from employing the data types described above for server and client throughout all application programming interfaces, an appropriate way of both sending and receiving a set of arguments had to be chosen as well. Since libvirt is a library guaranteeing its stability, a public function signature cannot change, so arguments should be packed in a structure. But defining public structure is tricky, since introducing a new attribute and then adding it to such a structure would most probably either change the size of the structure or its padding, thus inherently breaking backwards binary compatibility. Making such a structure completely private would usually help, but because libvirt utilizes RPC which does not support communication protocol versions, adding a new attribute to a private structure would only work if a reasonably large array would be added to XDR data structures, just as a place holder for the private structure to be able to support more attributes later[3]. And that is exactly how libvirt tackled this issue, by placing a reasonably large array filled with a universal data type called `virTypedParameter` shown on Listing 5.3, capable of holding any scalar types, at the end of specific XDR data structures. Thus, administration interfaces that are likely to send a number of parameters to the remote host made use of this existing data type libvirt exports. A list of typed parameters is then taken as an argument by majority of functions exposed by adminstration interface.

5.2 Load Balancing: Managing a Threadpool's Attributes

Recalling *Workerpool Limits* from Section 3.4.2, each of libvirtd's servers utilizes a threadpool to achieve concurrent execution. It also described two types of workers that are contained within the threadpool: workers responsible for non-privileged tasks, and priority workers that perform tasks which are under libvirt's full control and are guaranteed to always (high-priority tasks) finish. It also mentioned that, while the count of the former is constrained by the upper and bottom limits and the actual number of threads is managed

[3]Withouth this approach, data growth by even a small fraction would cause compatibility problems during data RPC deserialization on the remote host. Consider a scenario when the deserializer could actually understand the procedure number but the daemon is linked with a version of libvirt library that is several releases behind the one used on the client, the deserializer would then fail to deserialize the data received from the client, due to the extra portion of data.

```
struct _virTypedParameter {
    char field[VIR_TYPED_PARAM_FIELD_LENGTH];
    int type;
    union {
        int i;
        unsigned int ui;
        long long int l;
        unsigned long long int ul;
        double d;
        char b;
        char *s;
    } value;
};
```

Listing 5.3: Libvirt's universal data type that is capable of holding any scalar type

automatically, the count of the latter is constant. It is then clear that unlike priority workers, which use their "count" attribute available for both reading and writing, non-priority workers use it for reading only and all configurations are done via its limits. Following is then the complete list of supported attributes (read-only attributes are marked):

- `minWorkers` – the bottom limit to number of active workers,
- `maxWorkers` – the upper limit to number of active workers,
- `freeWorkers` – the number of currently unoccupied workers (read-only attribute),
- `nWorkers` – the current number of active workers, both occupied and free workers (read-only attribute),
- `prioWorkers` – the current number of priority workers, and
- `jobQueueLength` – the number of tasks waiting in queue to be processed (readonly attribute).

5.2.1 Problem Details

The only issue with daemon-side logic was thread termination. As opposed to thread creation which is done in a generic and expected manner – calling `pthread_create` – termination was not a straight forward action. The question was which thread was the most suitable one to be terminated. Preferably, the victim should be one of the threads waiting for a task to perform. Since there is no way for libvirt to address a specific thread, nor does libvirt track the current state of a thread, the original idea of how to tackle this issue was to exploit the nature of threadpool's task-based model, i.e. creating a task that would force a thread to call `pthread_exit` and terminate immediately[4]. A typical scenario would then result in following actions:

1. Create as many termination tasks, as there are threads to be terminated.

[4]Libvirt's threadpools are composed of detached threads, thus it is unnecessary to wait for the exit status of a thread.

2. Insert all termination tasks into the task queue.
3. Signal the threadpool's condition to wake up workers.
4. Finalize the API.

However, this design suffered from a flaw. The problem were libvirt's lockable structures (or objects in libvirt's terminology). Each time such an object needs to be modified, it needs to be locked first. Threadpool is one of such objects, therefore any modification to it requires a lock to be acquired. So the administration interface acquires a lock to make adjustments to the threadpool and then tries to invoke another method (to insert into threadpool's task queue) which also requires a lock on the threadpool object. It is obvious, that such an execution would lead to an instant deadlock. Although a solution to this problem exists, it was rather a complex one, involving code replication which definitely cannot be counted as an optimal solution for upstream.

5.2.2 Solution

Naturally, a different approach was taken. Each thread executes in an infinite loop, taking out, executing, and waiting for new incoming tasks. When a thread starts, it receives a reference to the threadpool it belongs to from the main thread, just to be able to work with threadpool's task queue. Having a threadpool reference, the thread has everything it needs to determine whether it should or should not terminate. For this purpose a helper function `virThreadPoolWorkerQuitHelper` comparing the current number of threads with the upper limit was created. Since some threads may be busy with a certain task during the time the upper limit decreases, while some are waiting on a condition, this helper method has to be invoked:

- after a thread stops waiting for a condition, but before inspecting the queue for new tasks, and
- after a thread just finished execution of a task, but before it becomes passive to wait for a new task to arrive.

5.3 Runtime Introspection: Managing a Daemon's Logging Parameters

Like with threadpool management, concurrency was the major issue during logging management implementation. This section first clarifies the functionality of a daemon's logging subsystem, how it worked prior to implementation changes addressed in this section, then the specifics of the concurrency problem as well as the steps towards a working solution are described.

The logging subsystem comprises of three parts:

- global log level,
- logging filters, conforming to format 4.6, and
- logging outputs, conforming to format 4.8.

For more information about log level, its hierarchy, and different kinds of log outputs, refer to Section 4.2. The purpose of logging filters is then to implement behaviour similar to a blacklist or a whitelist, depending on what the global log level setting is—in case of the former the global level equals to `DEBUG`, which implies that every single module becomes as verbose as possible, which is is not optimal and therefore filters to suppress some modules need to be defined; for the latter, both the global level setting, as well as the filters work in the opposite way. Logging outputs then restrict which messages (according to their level) are allowed to be forwarded to certain output.

5.3.1 Problem Details

Each worker might want to log some information about its activity during a task execution. Once the worker enters the logging subsystem, filters are applied first to decide whether the message should be logged or dropped[5]. Only after then the thread acquires a lock on the logging subsystem to forward the message to all defined log outputs. Finally, it releases the lock for other potential threads waiting to log their messages.

Libvirtd uses a configuration file to modify its logging settings. These settings are static, applied during daemon start-up and stay the same throughout the daemon's lifetime. However, the way new filters and outputs are defined had to be changed significantly, in order to be exposed by administration interface. The problem was the string form of both logging filters and logging outputs. Once read, the string is divided into individual filters/outputs which are then passed to a parser. It is the parser who is responsible not only for parsing but for defining the filter/output as well. Eventually, this means that to define a filter/output, critical section has to be acquired, the filter/output is appended to a list and the critical section is then released. For one-time initialization, this approach works well, but the lack of atomicity in the defining operation poses a problem when doing this during runtime. This situation is depicted by Figure 5.1. Consider thread 1 and 2 being actors where thread 1 is trying to define a new set of logging filters. But the operation as a whole is not atomic, it is rather divided into several sub-actions – clearing an existing set of filters and defining a new one gradually, i.e. one filter at a time – that might be interrupted by thread 2 trying to log its actions that are part of its task execution (creating a domain for instance). Since thread 1 did not apply all its changes to the logging subsystem before thread 2 interrupted it, thread 2 breaks the log consistency. It is clear that any changes to the logging subsystem must be done atomically to prevent such scenarios. Once the operation of defining a new set of filters/outputs is made atomic, thread 2's action would inherently happen before or after adjustments to logging settings have been made (green arrows) and the problem would be then solved.

5.3.2 Solution

Both filter and output parser were refactored so that the defining logic could be stripped away to a separate method. Naive solution would be to enclose both the parser and the defining method in a critical section, but since the parser itself can generate errors that are being logged, an instant deadlock would occur. The approach taken to tackle this problem was read-copy-update method[6]. Instead of changing the setting gradually, thread 1 would

[5]For historical reasons related to a long critical section and significant performance drop, checking whether filters for that specific module the message comes from are up-to-date, as well as applying the filter is not done in a mutual exclusive manner.

[6]https://en.wikipedia.org/wiki/Read-copy-update

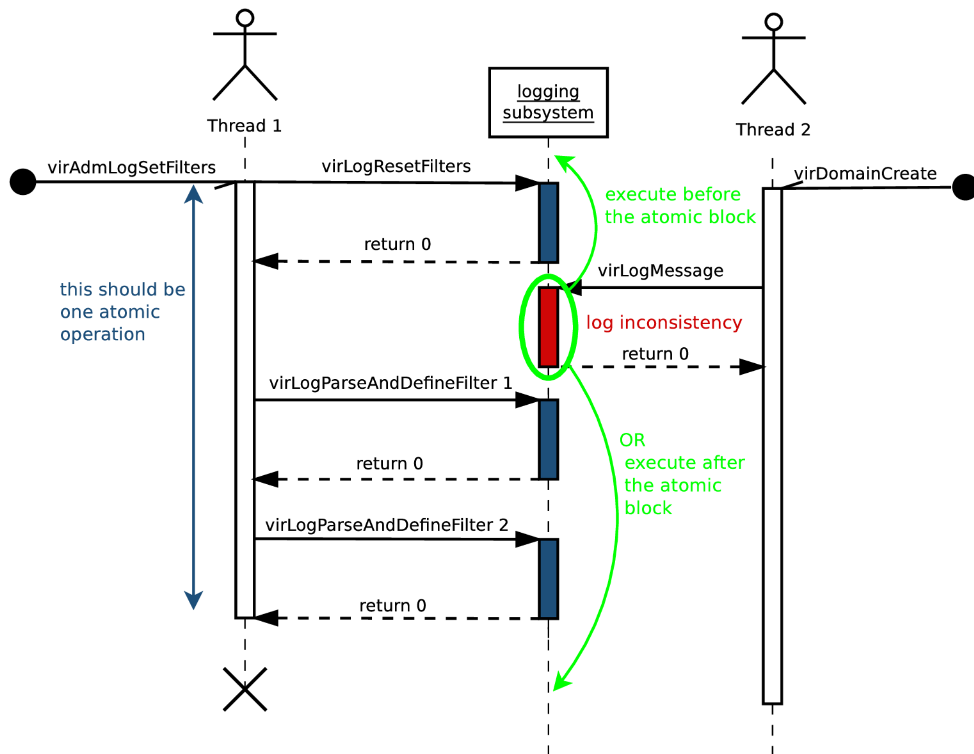


Figure 5.1: Non-atomic definition of a new set of logging filters

parse its input, create a copy of the existing settings, apply the changes to its private copy and only then proceed with defining new settings. This concept works well for logging filters, it also works well for creating or dropping an output, but has to be further tweaked to work with syslog if the identifier that syslog prepends to every message needs to be changed. Syslog keeps its file descriptor to system daemon private, thus for every identifier change, syslog needs to be reopened, which needed to be deferred to the very last moment of updating the logging settings. If it were reopened any earlier, where an error can still occur, the logging settings would remain unchanged, except for the syslog which at this point would cause inconsistency in the log file.

5.4 Virt-admin Command-line Interface

Chapter 1 mentioned that although libvirt is mainly recognized as a virtualization management library, it is a a virtualization toolkit. Apart from the library, this toolkit is composed of libvirtd, virtlockd, virlogd daemons, and a console client called *virsh* as well. Virsh client is an interactive shell, a tool, for performing management tasks on libvirt-managed domains, storage, and networks. From average user's perspective, it is merely a convenience tool, which is scriptable, to accomplish various management tasks without having to write their own applications linked with libvirt library.

For the very same purpose—to provide a tool to demonstrate libvirt-admin capabilities—*virt-admin* client has been created as part of this thesis. The client itself is based on virsh client but unlike virsh, which can be run both as user root and other users, virt-admin can be run with user root's privileges only. The reason for this is that administration interface

essentially grants the user full control of libvirtd daemon. When the idea of managing libvirtd itself arose, a decision that only user root should have the privilege to manage libvirt daemon was made. Therefore, the UNIX socket which virt-admin connects to has its permissions set in a way that only user root can connect to it.

Since the key functionality of the administration interface has been demonstrated using virt-admin client in Section 4.2, to conclude this chapter, a list of all supported commands in virt-admin is provided:

```
# virt-admin help
Grouped commands:
 Virt-admin itself (help keyword 'virt-admin'):
    cd                     change the current directory
    echo                   echo arguments
    exit                   quit this interactive terminal
    help                   print help
    pwd                    print the current directory
    quit                   quit this interactive terminal
    uri                    print the admin server URI
    version                show version
    connect                connect to daemon's admin server

 Monitoring commands (help keyword 'monitor'):
    srv-list               list available servers on a daemon
    srv-threadpool-info    get server workerpool parameters
    srv-clients-info       get server client's processing controls
    srv-clients-list       list clients connected to <server>
    dmn-log-info           view daemon's current logging settings
    client-info            retrieve client's identity from server

 Management commands (help keyword 'management'):
    srv-threadpool-set     set server workerpool parameters
    srv-clients-set        set server's client processing controls
    dmn-log-define         change daemon's logging settings
    client-disconnect      disconnect client from server
```

Listing 5.4: virt-admin client commands

Chapter 6

Testing of Selected Administration Interfaces

This chapter focuses on testing of the proposed application programming interfaces. The test suite that will be further explored in this chapter covers the following test subjects:

- setting global logging level on libvirtd daemon,
- setting a new set of logging filters on libvirtd daemon,
- setting a new set of logging outputs on libvirtd daemon, and
- setting a server's threadpool attributes.

The reason for choosing such a limited set of subjects to test is because the corresponding interfaces are the most interesting ones in terms of number of parameters, as well as the nature of the parameters to test. In other words, those are functions that take arguments which are necessary to modify the current state of libvirtd (*setters*), while the rest of the administration interface being composed of functions that only retrieve values (*getters*) from remote host and return it to the caller either by value or by reference, with arguments passed by reference being allocated by libvirt library automatically, and functions that take only one parameter[1] – connection object, server object or client object – that is checked the same way as with the other interfaces.

The technique used for testing of the aforementioned subjects was black-box testing and its *equivalence partitioning* methodology which is used to divide a set of test conditions into groups or sets that are considered equal. These equivalence classes are usually derived from the specification of input parameters that will directly influence the way an object will be processed. In equivalence-partitioning technique, there is a need to test just one condition from each partition (class, or block as referred to by testing literature). According to Myers [18], this is because one can reasonably assume that testing a representative value for each class is equivalent to testing any other value from that class. Myers also says, that each test case should consider different combinations of inputs thoroughly in order to reduce the total number of test cases necessary, as opposed to exhausting testing where a test case for every possible combination of inputs would exist.

[1]Technically, the minimal number of arguments accepted by any of the designed interfaces is two, but `flags` argument is explicitly documented not be used yet, otherwise the call will fail. Therefore, this argument is not subject to testing of any of the above mentioned interfaces.

There are definitely some risks coming with the selected subjects. The first one, also addressed by Section 6.2, is related to `nparams` argument which represents the number of elements in a container. While it is perfectly valid to pass an empty container to a specific method, trying to allocate zero elements during data serialization in order to send them over the network might also fail, depending on the platform. For this reason, a characteristic of `nparams` equal to zero has been omitted. So there is a chance that while trying to send an empty container to the server and expecting success, it could in fact return failure if certain conditions are met.

Another risk worth mentioning would be a possibility of a hidden flaw in one of the daemon-side internal functions. Since a fair amount of logic had to be added in order to be able to wire up the administration interface to the daemon, such a flaw might not get exposed by just testing various combinations of input arguments to application interfaces on the client side, rather than being triggered by a specific sequence of internal calls on the daemon side. Because of this, unit testing would make a good candidate to supplement the actual method used and test some of the daemon-side internal functions as well.

6.1 Modifying Daemon Logging Settings

This section covers an analysis of the following interfaces:

- `virAdmConnectSetLoggingLevel`,
- `virAdmConnectSetLoggingFilters`, and
- `virAdmConnectSetLoggingOutputs`.

Before diving into each one individually, it is worth realizing that they only differ in one argument and have two arguments in common, being the connection object `conn` and `flags`. The `flags` arguments is explicitly documented not to be used yet, since besides future extensibility of all libvirt interfaces, it is of no purpose at the moment. Because the connection object argument is common to all above mentioned functions, its complete analysis is going to be carried out globally.

The `conn` object of type `virAdmConnectPtr` is used to specify an active connection a client is maintaining with the daemon and there is only one valid way how such an object can be created (besides doing a typecast from a different object)—calling `virAdmConnectOpen`, which will return a newly created connection object. The implementation of the object is private and libvirt only supplies the callers with accessors to retrieve some of its attributes, thus once such an object is created there is no way for the caller to change it besides trying to mangle the binary data in memory forcefully. Following these facts, there is only a single characteristic of `conn` object with three classes (A, B, and C), see the Table 6.1.

Characteristic	**Valid Equivalence Classes**	**Invalid Equivalence Classes**
Connection status	active connection (A)	closed/timed-out connection (B), object is `NULL` (C)

Table 6.1: Input characteristic of `virAdmConnect` object.

Setting Global Logging Level

The first function to examine is `virAdmConnectSetLoggingLevel` (signature below), which sets the global logging level for a daemon.

```
int
virAdmConnectSetLoggingLevel(virAdmConnectPtr conn,
                             unsigned int level,
                             unsigned int flags);
```

Listing 6.1: virAdmConnectSetLoggingLevel function prototype.

The `level` attribute is a numeric representation of one of four supported logging levels – debug, info, warning, error – described in Chapter 4 and is thus limited to hold only values from range 1-4. Partitioning to equivalence classes is shown in Table 6.2.

Characteristic	Valid Equivalence Classes	Invalid Equivalence Classes
Value range	$1 - 4$ (1)	< 1 (2), > 4 (3)

Table 6.2: Input characteristic of `level` argument.

Recalling Table 6.1 of the `virAdmConnect` object analysis, Relation 6.1 then denotes T_1' as being Cartesian product of the first two input arguments to the function.

$$\begin{aligned} T_1' = Connection\ status \times Value\ range = \{&(A,1),(A,2),(A,3),\\ &(B,1),(B,2),(B,3),\\ &(C,1),(C,2),(C,3)\} \end{aligned} \tag{6.1}$$

Following the rules stated by Myers in [18], each test case should cover one, and only one invalid equivalence class. Myers claims the reason for this to be erroneous-input checks masking each other. Therefore, all pairs, such that both elements are invalid equivalence class representatives, need to be removed from the set T_1' to produce the final set of test cases T_1, denoted by Relation 6.2.

$$T_1 = \{(A,1),(A,2),(A,3),(B,1),(C,1)\} \tag{6.2}$$

Setting Logging Filters

Next function to examine is `virAdmConnectSetLoggingFilters` (signature below), which allows to set one or multiple logging filters that are applied before sending each message to output.

```
int
virAdmConnectSetLoggingFilters(virAdmConnectPtr conn,
                               char *filters,
                               unsigned int flags);
```

Listing 6.2: virAdmConnectSetLoggingFilters function prototype.

The `filters` attribute conforms to the format described in Section 4.6, with an identical constraint for `level`, as discussed in previous section. Recalling Listing 4.6, the contents of `module_name` part of the match string is not further specified by libvirt, since there are lots of modules within libvirt library (with new ones being added) and `module_name` works only as a match string that is compared to every message's source module before the message can be sent to the output. In other words, libvirt does not care about contents of the `module_name` string and the worst that can happen is it will be silently ignored.

Next constraint, other than format, on the `filters` argument is that if multiple filters need be defined, they have to be delimited by spaces. Table 6.3 reflects individual characteristics and corresponding equivalence classes. Before proceeding with the Cartesian product

Characteristic	**Valid Equivalence Classes**	**Invalid Equivalence Classes**
String pointer content	one filter (1), N filters (2), empty string (3)	`NULL` (4)
String starts with a number	yes (5)	no (6)
Number value	$1 - 4$ (7)	< 1 (8), > 4 (9)
Level and match string are delimited by a colon	yes (10)	no (11)
Match string is empty	no (12)	yes (13)
More filters delimited by spaces	yes (14)	no (15)

Table 6.3: Input characteristic of `filters` argument.

of inputs, Table 6.3 should be examined thoroughly. What it can be seen from the table is that while it is perfectly achievable (and desirable as well) to produce a dedicated test case for every invalid equivalence class, only two test cases can be produced to test a valid input—one for a single filter conforming to the specified format and one for multiple filters delimited by spaces. This is because the majority of characteristics presented by Table 6.3 are related to the logging filter format and therefore they do form a hierarchy (more specific constraints put on individual parts of the format). For this reason, classes 12 and 14 cover all valid inputs for argument `filters`.

It is also worth noting that a case where there are multiple filters contained within the `filters` argument with some of them being valid an some invalid is not considered. This is because of the principle of how the parser on the daemon-side works. When it starts parsing the string, it first divides it into individual filters according to the delimiter. Each filter is then parsed separately, possibly triggering an error if a format mismatch has been detected. Therefore, the final set of necessary test cases T_2 can be then denoted by Relation 6.3.

$$\begin{aligned} T_2 = \{&(A,3),(A,4),(A,6),(A,8),(A,9),\\ &(A,11),(A,12),(A,13),(A,14),(A,15),\\ &(B,12),(B,14),(C,12),(C,14)\} \end{aligned} \tag{6.3}$$

Setting Logging Outputs

This section covers analysis of the `outputs` attribute to function `virAdmConnectSet-LoggingOutputs` (signature below).

```
int
virAdmConnectSetLoggingOutputs(virAdmConnectPtr conn,
                               char *outputs,
                               unsigned int flags);
```

Listing 6.3: virAdmConnectSetLoggingOutputs function prototype.

Recalling the format for an output string from Section 4.8, the situation with setting logging outputs is quite similar to setting filters. Unlike filters though, the match string which follows the logging level number now matters, because only the following strings are supported: *stderr*, *journald*, *syslog*, and *file*. There is another constraint put on the format, which is completely output dependent, with the constraint being that file-based and syslog-based outputs require one more string to denote some additional data—an identifier, which is prepended to every message for syslog-based output and a valid absolute path to a file for file-based outputs. The content of the additional data passed to syslog-based output does not matter to libvirt at all, since that is passed to `openlog` call untouched. With the file path for file-based output however, it can only be a valid absolute path to a file.

Taking all facts into account, Table 6.4 can be constructed analogically to Table 6.3 from previous section. As with filters, the very specific format of an output caused most

Characteristic	**Valid Equivalence Classes**	**Invalid Equivalence Classes**
String pointer content	one output (1), N outputs (2), empty string (3)	`NULL` (4)
String starts with a number	yes (5)	no (6)
Number value	1 – 4 (7)	< 1 (8), > 4 (9)
Level and output defining string are delimited by a colon	yes (10)	no (11)
Output string's content	[stderr, journald, syslog, file] (12)	random string (13)
Output requires additional data	[syslog, file] (14)	[stderr, journald] (15)
Data-requiring outputs and data are separated by a colon	yes (16)	no (17)
Path to a file for file-based output is valid	yes (18)	no (19)
More outputs are delimited by spaces	yes (20)	no (21)

Table 6.4: Input characteristic of `outputs` argument.

characteristics to form a hierarchy. Also, a test case for having multiple outputs with some of them being valid, while some of them being invalid has been omitted for the very same reasons discussed in previous section. Therefore, regarding all the valid equivalence classes, individual test cases for classes 12, 18, and 20 also satisfy all the remaining valid equivalence classes in the table. The complete set of pairs of inputs, excluding all pairs such that both

elements are representatives of invalid equivalence classes, for individual test cases T_3 is then denoted by Relation 6.4.

$$
\begin{aligned}
T_3 = \{&(A,3),(A,4),(A,6),(A,8),(A,9),(A,11),(A,12),(A,13),\\
&(A,15),(A,17),(A,18),(A,19),(A,20),(A,21),\\
&(B,12),(B,18),(B,20),\\
&(C,12),(C,18),(C,20)\}
\end{aligned}
\tag{6.4}
$$

6.2 Setting Threadpool Parameters

This section is going to cover `virAdmServerSetThreadPoolParameters` function (signature below). It is obvious from the signature that the data types required for the input arguments differ from all functions examined so far, so with the exception of `flags`, all of them will be addressed and analyzed properly.

```
int
virAdmServerSetThreadPoolParameters(virAdmServerPtr srv,
                                    virTypedParameterPtr params,
                                    int params,
                                    unsigned int flags);
```

Listing 6.4: virAdmServerSetThreadPoolParameters function prototype.

First to examine is the server object attribute `srv`. Like with the connection object, the server object, declared by Listing 5.1, also does have its implementation kept private by libvirt, thus is created automatically by the library whenever a call to `virAdmConnect-LookupServer` or `virAdmConnectListServers` is issued. Disregarding the structure internals, caller has only a pointer to such an object available, therefore the pointer can reference a valid server object (an existing one) or it is `NULL`[2]. However, the function still relies on an active connection, thus the connection object being an external dependency of the function. To simplify the analysis, rules of the black-box testing need to crippled to some degree, i.e. knowing the internal implementation of the `virAdmServer` object, the state of a connection (active, closed/timed-out) can be tracked from within the server object itself and therefore can be accounted as one of possible values for the server object function argument. This fact is the reflected by Table 6.5.

Characteristic	**Valid Equivalence Classes**	**Invalid Equivalence Classes**
Object is valid	valid reference (J)	connection is closed/timed-out (K), object is `NULL` (L)

Table 6.5: Input characteristic of `virAdmServer` object.

Continuing with `nparams` and `params` arguments. Although they could be considered a single input sub-domain due to being strongly connected, a simple constraint – the value for `nparams` can be either negative or positive, reflecting the actual number of parameters in

[2]Technically, there also could be a non-`NULL` invalid object reference, but only if the remote server could be shut down, which is an unsupported operation at the moment.

`params`[3]– could still be considered. Strictly speaking, the value could also be equal to zero and could be tested as well, but such a test case's behaviour would be non-deterministic, since data serialization and deserialization make a call to `calloc`, the return value of which in case of calling it with size equal to zero is implementation dependent—it can either return `NULL`, making the test fail or it can return a valid address[4], which in turn would make the test case pass. Another thing is that trying to pass a combination of `nparams` equal to zero and a non-empty container eventually results in sending no data to the server, which, again depending on the implementation of allocation functions, might return success although a failure due to an invalid content of the container was expected. Therefore, a case when `nparams` is equal to zero will not be included in the final set of test cases.

Characteristic	**Valid Equivalence Classes**	**Invalid Equivalence Classes**
Numeric value	> 0 and reflecting the actual number of parameters in `params` (a)	< 0 (b)

Table 6.6: Input characteristic of `nparams` argument.

Lastly, `params` is a either a `NULL` pointer or points to a valid container of typed parameters. The container is constrained to hold at least one element and in case of multiple elements, these have to be distinctive, i.e. no duplicate elements are allowed within the container. For the internal structure of an element, as declared by Listing 5.3, libvirt exports type-specific methods, `virTypedParamsAddInt` and `virTypedParamsAddString` for instance, to add an element into a list. Because they are type-specific, callers only specify a string-based field identifier. The type for the field identifier is selected automatically depending on the function called. Moreover, each element to be set on the remote host does have its unique string identifier that the remote host will expect to find. For this particular method, the supported identifiers are defined as shown by Listing 6.5.

```
# define VIR_THREADPOOL_WORKERS_MIN "minWorkers"
# define VIR_THREADPOOL_WORKERS_MAX "maxWorkers"
# define VIR_THREADPOOL_WORKERS_PRIORITY "prioWorkers"
```

Listing 6.5: Field identifiers supported for virAdmServerSetThreadPoolParameters `params` argument.

The documentation (refer to Appendix A for details) also specifies the expected types for all field identifiers. Putting all the facts together, Table 6.7 specifies all characteristics used to partition the input sub-domain `params`.

As with the other examinations performed so far, because of an existing hierarchy of characteristics for an argument, creating test cases for the right valid equivalence classes would also make these test cases cover most of the remaining valid equivalence classes as well. More specifically, classes 6 and 10 do satisfy conditions of the remaining valid classes as well. Therefore, these will be the only valid classes taken into consideration.

[3] This simple division of the input sub-domain is due to the fact that since libvirt only receives a pointer to the list of typed parameters, it cannot possibly verify without triggering segmentation violation, whether size of the list of parameters truly corresponds to the value determined by `nparams`.

[4] Section *Memory management functions* in ISO/IEC 9899 states that in case a valid pointer is returned, it shall not be dereferenced, which is not the case with libvirt.

Characteristic	Valid Equivalence Classes	Invalid Equivalence Classes
Pointer content	a single element (1), a list of elements (2)	`NULL` (3)
Field identifier	[minWorkers, maxWorkers, prioWorkers] (4)	random string (5)
Element's data type	unsigned integer (6)	other (7)
Duplicates in the list	no (8)	yes (9)
maxWorkers and minWorkers relation	maxWorkers >= minWorkers (10)	maxWorkers < minWorkers (11)

Table 6.7: Input characteristic of `params` argument.

Since there are now three input sub-domains to be combined, the final set of test cases necessary T_4 is going to be constructed step-by-step. Let q_1 denote the block of partitions for `virAdmServer` object, q_2 the block of partitions for `params` argument, and q_3 the block of partitions for `nparams` argument.

$$q_1 = \{J, K, L\}, q_2 = \{3, 5, 6, 7, 9, 10, 11\}, q_3 = \{a, b\} \tag{6.5}$$

Next the Cartesian product of each pair of blocks q_1, q_2, and q_3 needs to be produced. The resulting sets Tq_1q_2, Tq_1q_3, and Tq_2q_3 are denoted by Relation 6.6. As it can be seen, pairs such that both elements in the pair are representatives of invalid equivalence classes were omitted from the sets.

$$\begin{aligned} Tq_1q_2 = q_1 \times q_2 = \{&(J,3),(J,5),(J,6),(J,7),(J,9),(J,10),\\ &(J,11),(K,6),(K,10),(L,6),(L,10)\}\\ Tq_1q_3 = q_1 \times q_3 = \{&(J,a),(J,b),(K,a),(L,a)\}\\ Tq_2q_3 = q_2 \times q_3 = \{&(3,a),(5,a),(6,a),(6,b),(7,a),\\ &(9,a),(10,a),(10,b),(11,a)\} \end{aligned} \tag{6.6}$$

Relation 6.7 puts all sets denoted by 6.6 together into a complete set of triplets T_4. Again, triplets such that multiple elements in the triplet are representatives of invalid equivalence classes were omitted from the set.

$$\begin{aligned} T_4 = \{&(J,3,a),(J,5,a),(J,6,a),(J,6,b),(J,7,a),\\ &(J,9,a),(J,10,a),(J,10,b),(J,11,a)\\ &(K,6,a),(K,10,a),(L,6,a),(L,10,a)\} \end{aligned} \tag{6.7}$$

The T_4 set could be further optimized according to the horizontal and vertical growth algorithms described in [24], which ensures that the final set of test cases contains each pair, optimizing out test cases that comprise of pairs which are already covered by other test cases. The optimized set of test cases T_4' would then be denoted by Relation 6.8.

$$\begin{aligned} T_4' = \{&(J,3,a),(J,5,a),(J,6,b),(J,7,a),\\ &(J,9,a),(J,10,b),(J,11,a),(K,6,a),\\ &(K,10,a),(L,6,a),(L,10,a)\} \end{aligned} \tag{6.8}$$

As it can be seen, after the optimization, the T_4' set contains two test cases less than the full set T_4. There is a drawback however, without further context about the nature of

the test cases in T_4, case $(J, 6, a)$, comprising of pairs $(J, 6)$, (J, a), and $(6, a)$, is pairwise already covered by cases $(J, 6, b)$, $(J, 3, a)$, and one of cases $(L, 6, a)$ and $(K, 6, a)$. Therefore, according to [24] it could be optimized out. The same approach was taken with case $(J, 10, a)$ which consists of pairs $(J, 10)$, (J, a), and $(10, a)$. It is already covered by cases $(J, 10, b)$, $(J, 3, a)$, and one of cases $(L, 10, a)$ and $(K, 10, a)$. However, unlike all the test cases in T'_4 which only test erroneous paths (one member of the triplet is always an invalid class representative), the two test cases mentioned above do test successful paths of the execution.

6.3 Testing Toolset Details

Joining all the test cases from sets T_1 to T_4 together, the number of test cases is rather large to be tested manually, therefore an automated approach was taken. The implemented toolset for the test suite comprises of the following three parts:

- *admin_test_run.sh* script - prepares the environment, possibly running *admin_test.py* afterwards,
- *admin_test.py* script - batch execution of test cases, and
- *admin_test.c* program - individual test case execution.

admin_test_run.sh

At the time of the writing of this thesis, administration API is still not enabled in the code by default yet. Therefore, libvirt needs to be build from sources[5] (from a specific branch) with the administration API support enabled. Once libvirt is built successfully, this script then compiles the *admin_test* program and links it against the *libvirt-admin.so* library that was created in the first step. If not run with option `-n`, *admin_test.py* is automatically executed once the preparation phase is finished.

admin_test.py

This script represents the test automation part of the test suite, rather than specifying each test case via a combination of command-line arguments to *admin_test* program manually for each test case, this script is able to run test cases specified in a batch. After the script is done with parsing the test cases from a file, it executes a libvirtd process on background and then executes the *admin_test* program, formatting the test case parameters on the command-line. Once the test case has finished, the daemon process is restarted to provide a fresh instance for the next test case in the list.

The format that is accepted for a test case is described in *admin_test.txt* file which also serves as a working example of how a test batch file can look like. Listing 6.6 demonstrates how a test batch file can be specified to the script. This example also includes usage of variables `LIBVIRT_GIT` and `LD_LIBRARY_PATH`. While the former is only necessary if a custom git repository location was provided to the *admin_test_run.sh* script, the latter is absolutely crucial for the *admin_test* binary to run, otherwise the program loader will have no information where the libvirt-admin library necessary for linking against should be located, thus causing the program to fail.

[5]To build libvirt from git repository, a significant amount of dependencies have to be installed first. README file which describes the test suite also provides all the necessary information about libvirt's dependencies

```
# LIBVIRT_GIT=/tmp/libvirt.git LD_LIBRARY_PATH=$LIBVIRT_GIT/src/.libs \
> ./admin_test.py -f admin_test.txt
```

Listing 6.6: Running admin_test.py script.

Recalling section *Virt-admin Command-line Interface* in Chapter 5 which described the reason why any client connecting to libvirt daemon via administration interface is required to run with root privileges, the *admin_test.py* script executing both the daemon that is supposed to run as root and the *admin_test* binary, being effectively a client connecting to the daemon, needs to be run with root privileges as well.

admin_test.c

This is the most crucial part of the whole toolset. The program exposes several command-line options to manually specify a test case that should be executed. The test cases are divided into groups according to the interface tested, i.e. the supported test groups include logging level, logging filters, and logging outputs modification, as well as altering a server's threadpool parameters. An example of how an output from the test suite looks like is illustrated by Listing 6.7.

```
20) logging-outputs@empty-string:         PASSED
21) logging-outputs@output-NULL:          PASSED Got expected error: ...
22) logging-outputs@level-not-numeric:    FAILED Error: ...
```

Listing 6.7: admin_test program's output fragment.

To conclude this chapter, after running the test suite on the selected administration interfaces which were discussed throughout the course of this chapter, the results show that all of the test cases that were designed through sections 6.1 and 6.2 passed successfully. Thus, from that perspective, the input domains for the selected administration interfaces can be considered as tested for various combinations. Source codes of the implemented testing toolset are available on the optical medium in `/tests`.

Chapter 7

Conclusion

The goal of this thesis was to design and implement a set of administration application programming interfaces for libvirtd daemon's runtime management. These interfaces would be implemented in C language and would cover features including adjustment of the number of workers in a server's threadpool, modification of logging levels, filters, and logging outputs in libvirtd daemon, as well as remote client management. The complete review of all application interfaces that were implemented as part of this work can be found in Appendix A. Along with the administration interfaces, a command-line client virt-admin has been created to preview the administration APIs' functionality. Selected interfaces were subject to black-box testing using the equivalence partitioning method. As the results showed, the whole test suite passed successfully.

From the upstream's point of view, all interfaces (including virt-admin), except for the logging interfaces that are still in a review process[1], were merged into the master branch of libvirt's git repository[2]. All commits, including refactors, as well as minor bug fixes related to the administration interface are available on the attached optical media in `/commits`. Additionally, since the administration interface for libvirtd daemon is a brand new feature that the upstream community might be interested in, the work of this thesis became a topic for a developer presentation that has been proposed for KVM Forum 2016 event[3].

Future improvements usually depend on individual customer's requirements, but there are several ideas that could be incorporated irregardless of any request. On of such would be sending an event to a third-party management application whenever a client connects or disconnects, providing all information about that client libvirt is able to gather. Another improvement would be to provide an API to tell libvirtd that the host is shutting down, thus all virtual machines' states should be saved in order to be resumed after the system is running again. The current approach utilizes a script which appears as a conventional client, which can be tricky if polkit authentication is turned on because the polkit service might be already off, which in the end causes a virtual machine to loose its last known state[4]. Since libvirtd runs as user root and administration interface therefore requires root privileges to manage it, there is no dependency on polkit, so a virtual machine would essentially resume to its last known state before the host's reboot successfully.

[1] commits available at https://github.com/eskultety/libvirt/tree/logging

[2] http://libvirt.org/git/?p=libvirt.git;a=shortlog;h=refs/heads/master

[3] http://events.linuxfoundation.org/cfp/proposals/11119/9664

[4] https://bugzilla.redhat.com/show_bug.cgi?id=1235522

Bibliography

[1] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *SIGARCH Comput. Archit. News*, 34(5):2–13, October 2006.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003.

[3] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[4] M. Ben-Yehuda and J. D. Mason. The xen hypervisor and its io subsystem. http://www.mulix.org/lectures/xen-iommu/xen-io.pdf, December 2005. [Online], [Accessed: 2015-12-29].

[5] Matthias Bolte, Michael Sievers, Georg Birkenheuer, Oliver Niehörster, and André Brinkmann. Non-intrusive virtualization management using libvirt. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 574–579, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[6] N.M. Mosharaf Kabir Chowdhury and Raouf Boutaba. A survey of network virtualization. *Computer Networks*, 54(5):862 – 876, 2010.

[7] Prasun Dewan. Multics rings. http://www.cs.unc.edu/~dewan/242/f96/notes/prot/node11.html, Nov 1996. [Online], [Accessed: 2016-01-07].

[8] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pages 171–172, March 2015.

[9] Yasunori Goto. Kernel-based virtual machine technology. *FUJITSU Scientific and Technical Journal*, 47(3):362—369, July 2011.

[10] Irfan Habib. Virtualization with kvm. *Linux J.*, 2008(166), February 2008.

[11] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. Hqemu: A multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 104–113, New York, NY, USA, 2012. ACM.

[12] Lan Huang, Gang Peng, and Tzi-cker Chiueh. Multi-dimensional storage virtualization. *SIGMETRICS Performance Evaluation Review*, 32(1):14–24, June 2004.

[13] R. Jithin and Priya Chandran. Virtual machine isolation. In *Recent Trends in Computer Networks and Distributed Systems Security*, volume 420 of *Communications in Computer and Information Science*, pages 91–102. Springer Berlin Heidelberg, 2014.

[14] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. Kvm: the linux virtual machine monitor. In *In Proceedings of the 2007 Ottawa Linux Symposium*, OLS' 07, Jun 2007.

[15] Joseph Migga Kizza. *Guide to Computer Network Security*. Springer-Verlag, London, UK, UK, 2015.

[16] Hidenari Koshimae, Yuki Kinebuchi, Shuichi Oikawa, and Tatsuo Nakajima. Using a processor emulator on a microkernel-based operating system. In *RTCSA 2006 Work In Progress Session*, NICT Australia, pages 31–37, Kensington, Australia, 2006.

[17] R. Morabito, J. Kjallman, and M. Komu. Hypervisors vs. lightweight virtualization: A performance comparison. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pages 386–393, March 2015.

[18] Glenford J. Myers, Tom Badgett, and Corey Sandler. *The Art of Software Testing, Third Edition*. John Wiley & Sons, 2012.

[19] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[20] M. Rosenblum and T. Garfinkel. Virtual machine monitors: current technology and future trends. *Computer*, 38(5):39–47, May 2005.

[21] N. B. Sahgal and D. Rogers. Understanding intel virtualization technology. http://download.microsoft.com/download/9/8/f/98f3fe47-dfc3-4e74-92a3-088782200fe7/twar05015_winhec05.ppt. [Online], [Accessed: 2015-12-29].

[22] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008.

[23] Aameek Singh, Madhukar Korupolu, and Dushmanta Mohapatra. Server-storage virtualization: Integration and load balancing in data centers. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[24] Kuo-Chung Tai and Yu Lei. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering*, 28(1):109–111, Jan 2002.

[25] Vilmar Travassos. Virtualization trends trace their origins back to the mainframe. http://www.ibmsystemsmag.com/mainframe/administrator/Virtualization/history_virtualization/, August 2012. [Online], [Accessed: 2015-12-27].

[26] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, May 2005.

[27] Geoffroy Vallee, Thomas Naughton, and Stephen L. Scott. System management software for virtual environments. In *Proceedings of the 4th International Conference on Computing Frontiers*, CF '07, pages 153–160, New York, NY, USA, 2007. ACM.

[28] Daniel Veillard. Libvirt the virtualization API. https://www.libvirt.org/. [Online], [Accessed: 2016-01-10].

[29] VMware. Understanding full virtualization, paravirtualization, and hardware assist. https://www.vmware.com/pdf/virtualization.pdf, November 2007. [Online], [Accessed: 2015-12-26].

[30] VMware. Virtualization overview. http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf, 2007. [Online], [Accessed: 2015-12-26].

[31] Chwan-Hwa (John) Wu and J. David Irwin. *Introduction to Computer Networks and Cybersecurity*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2013.

[32] L. Yan. Development and application of desktop virtualization technology. In *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on*, pages 326–329, May 2011.

[33] Minhoon Yi, Dong Hyun Kang, Minho Lee, Inhyeok Kim, and Young Ik Eom. Performance analyses of duplicated I/O stack in virtualization environment. In *Proceedings of the 10th International Conference on Ubiquitous Information Management and Communication*, IMCOM '16, pages 1–6, New York, NY, USA, 2016. ACM.

Appendices

List of Appendices

A Libvirt Administration API Reference **57**

B Libvirt Administration API Usage Examples **71**

B.1 Listing Available Servers on Daemon 71
B.2 Listing All Clients Connected To a Given Server 72
B.3 Getting and Setting Logging Level 74
B.4 Getting and Setting Logging Filters 76
B.5 Getting and Setting Logging Outputs 77
B.6 Getting and Setting Server Threadpool Parameters 78
B.7 Getting and Setting Client Limits on Server 79
B.8 Retrieving a Client's Identity Information 81
B.9 Closing a Client's Connection Forcefully 84

C Contents of the CD **86**

Appendix A

Libvirt Administration API Reference

To utilize libvirt administration library, i.e. to include all administration definitions, following directive shall be used in the module:

```
# include <libvirt/libvirt-admin.h>
```

Type Documentation

This section provides a list of data types created as part of the thesis, as well as a brief description for each.

```
typedef enum {
    VIR_LOG_DEBUG = 1,
    VIR_LOG_INFO,
    VIR_LOG_WARN,
    VIR_LOG_ERROR,
} virLogPriority;
```

These levels are recognized by daemon's logger and can also be statically configured in daemon's configuration file (e.g. libvirtd.conf) with option `log_level`. For runtime level retrieval and modification use `virAdmConnectGetLoggingLevel` and `virAdmConnectSetLoggingLevel` methods which set the global logging level that can be further altered by utilizing a set of logging filters.

```
typedef enum {
    VIR_CLIENT_TRANS_UNIX = 0,
    VIR_CLIENT_TRANS_TCP,
    VIR_CLIENT_TRANS_TLS,

# ifdef VIR_ENUM_SENTINELS
    VIR_CLIENT_TRANS_LAST
# endif
} virClientTransport;
```

These constants represent the transport type a client is connected with and are returned int `transport` attribute in client object which can be obtained either by doing a client lookup `virAdmServerLookupClient` or by listing all clients connected to a server with `virAdmServerListClients`.

```
typedef _struct _virAdmServer virAdmServer;
typedef virAdmServer virAdmServerPtr;

typedef _struct _virAdmClient virAdmClient;
typedef virAdmClient virAdmClientPtr;
```

These are private structures posing a client-side representation of client and server objects with their respective pointer types which are the actual types used throughout all administration interfaces.

Macros

This section lists all macros created as part of this thesis and which are the only ones exported by libvirt administration library so far. The intended use of all macros listed in this section is to use them for string attribute `field` of the `virTypedParameter` structure (Listing 5.3) when inserting elements into a list of parameters. Basically, these string constants can be imagined as distinctive names for each element inside a typed parameters container that are recognized by the remote server.

Macros are divided into logical groups according to the interface they are supposed to be used with. Each macro is accompanied by a commentary explaining it's meaning and, most importantly, the expected type identifier it should be paired with.

Threadpool Attributes Managing Macros

```
/**
 * VIR_THREADPOOL_WORKERS_MIN:
 * Macro for the threadpool minWorkers limit: represents the bottom limit to
 * number of active workers in threadpool, as VIR_TYPED_PARAM_UINT.
 */
# define VIR_THREADPOOL_WORKERS_MIN "minWorkers"

/**
 * VIR_THREADPOOL_WORKERS_MIN:
 * Macro for the threadpool maxWorkers limit: represents the upper limit to
 * number of active workers in threadpool, as VIR_TYPED_PARAM_UINT.
 * The value of this limit has to be greater than VIR_THREADPOOL_WORKERS_MIN
 * at all times.
 */
# define VIR_THREADPOOL_WORKERS_MAX "maxWorkers"

/**
 * VIR_THREADPOOL_WORKERS_PRIORITY:
 * Macro for the threadpool nPrioWorkers attribute: represents the current
```

```
 * number of active priority workers in threadpool, as VIR_TYPED_PARAM_UINT.
 */
# define VIR_THREADPOOL_WORKERS_PRIORITY "prioWorkers"

/**
 * VIR_THREADPOOL_WORKERS_FREE:
 * Macro for the threadpool freeWorkers attribute: represents the current
 * number of free workers available to accomplish a job,
 * as VIR_TYPED_PARAM_UINT.
 * NOTE: This attribute is read-only and any attempt to set it will be
 * denied by daemon.
 */
# define VIR_THREADPOOL_WORKERS_FREE "freeWorkers"

/**
 * VIR_THREADPOOL_WORKERS_CURRENT:
 * Macro for the threadpool nWorkers attribute: represents the current
 * number of ordinary workers in threadpool, as VIR_TYPED_PARAM_UINT.
 * NOTE: This attribute is read-only and any attempt to set it will be
 * denied by daemon.
 */
# define VIR_THREADPOOL_WORKERS_CURRENT "nWorkers"

/**
 * VIR_THREADPOOL_JOB_QUEUE_DEPTH:
 * Macro for the threadpool jobQueueDepth attribute: represents the current
 * number of jobs waiting in a queue to be processed,
 * as VIR_TYPED_PARAM_UINT.
 * NOTE: This attribute is read-only and any attempt to set it will be
 * denied by daemon.
 */
# define VIR_THREADPOOL_JOB_QUEUE_DEPTH "jobQueueDepth"
```

Per-server Client Limits

```
/**
 * VIR_SERVER_CLIENTS_MAX:
 * Macro for per-server nclients_max limit: represents the upper limit to
 * number of clients connected to the server, as VIR_TYPED_PARAM_UINT.
 */
# define VIR_SERVER_CLIENTS_MAX "nclients_max"

/**
 * VIR_SERVER_CLIENTS_CURRENT:
 * Macro for per-server nclients attribute: represents the upper limit to
 * number of clients connected to the server, as VIR_TYPED_PARAM_UINT.
 * NOTE: This attribute is read-only and any attempt to set it will be
```

```
 * denied by daemon.
 */
# define VIR_SERVER_CLIENTS_CURRENT "nclients"

/**
 * VIR_SERVER_CLIENTS_UNAUTH_MAX:
 * Macro for per-server nclients_unauth_max limit: represents the upper
 * limit to number of clients connected to the server, but not
 * authenticated yet, as VIR_TYPED_PARAM_UINT.
 */
# define VIR_SERVER_CLIENTS_UNAUTH_MAX "nclients_unauth_max"

/**
 * VIR_SERVER_CLIENTS_UNAUTH_CURRENT:
 * Macro for per-server nclients_unauth attribute: represents the current
 * number of clients connected to the server, but not authenticated yet,
 * as VIR_TYPED_PARAM_UINT.
 * NOTE: This attribute is read-only and any attempt to set it will be
 * denied by daemon.
 */
# define VIR_SERVER_CLIENTS_UNAUTH_CURRENT "nclients_unauth"
```

Client Identity Management Macros

```
/**
 * VIR_CLIENT_INFO_READONLY:
 * Macro represents client's connection permission, whether the client is
 * connected in read-only mode or just the opposite - read-write,
 * as VIR_TYPED_PARAM_BOOLEAN.
 * NOTE: This attribute is read-only and any attempt to set it will be
 * denied by daemon.
 */
# define VIR_CLIENT_INFO_READONLY "readonly"

/**
 * VIR_CLIENT_INFO_SOCKET_ADDR:
 * Macro represents clients network socket address in a standard URI format:
 * (IPv4|[IPv6]):port, as VIR_TYPED_PARAM_STRING.
 * NOTE: This attribute is read-only and any attempt to set it will be
 * denied by daemon.
 */
# define VIR_CLIENT_INFO_SOCKET_ADDR "sock_addr"

/**
 * VIR_CLIENT_INFO_SASL_USER_NAME:
 * Macro represents client's SASL user name, if SASL authentication is
 * enabled on the remote host, as VIR_TYPED_PARAM_STRING.
```

```
 * NOTE: This attribute is read-only and any attempt to set it will be
 * denied by daemon.
 */
# define VIR_CLIENT_INFO_SASL_USER_NAME "sasl_user_name"

/**
 * VIR_CLIENT_INFO_X509_DISTINGUISHED_NAME:
 * Macro represents the 'distinguished name' field in X509 certificate the
 * client used to establish a TLS session with remote host, as
 * VIR_TYPED_PARAM_STRING.
 * NOTE: This attribute is read-only and any attempt to set it will be
 * denied by daemon.
 */
# define VIR_CLIENT_INFO_X509_DISTINGUISHED_NAME "x509_dname"

/**
 * VIR_CLIENT_INFO_UNIX_USER_ID:
 * Macro represents UNIX UID the client process is running with. Only
 * relevant for clients connected locally, i.e. via a UNIX socket,
 * as VIR_TYPED_PARAM_INT.
 * NOTE: This attribute is read-only and any attempt to set it will be
 * denied by daemon.
 */
# define VIR_CLIENT_INFO_UNIX_USER_ID "unix_user_id"

/**
 * VIR_CLIENT_INFO_UNIX_USER_NAME:
 * Macro represents the user name that is bound to the client process's UID
 * it is running with. Only relevant for clients connected locally,
 * i.e. via a UNIX socket, as VIR_TYPED_PARAM_STRING.
 * NOTE: This attribute is read-only and any attempt to set it will be
 * denied by daemon
 */
# define VIR_CLIENT_INFO_UNIX_USER_NAME "unix_user_name"

/**
 * VIR_CLIENT_INFO_UNIX_GROUP_ID:
 * Macro represents UNIX GID the client process is running with. Only
 * relevant for clients connected locally, i.e. via a UNIX socket,
 * as VIR_TYPED_PARAM_INT.
 * NOTE: This attribute is read-only and any attempt to set it will be
 * denied by daemon.
 */
# define VIR_CLIENT_INFO_UNIX_GROUP_ID "unix_group_id"

/**
 * VIR_CLIENT_INFO_UNIX_GROUP_NAME:
 * Macro represents the group name that is bound to the client process's GID
```

```
 * it is running with. Only relevant for clients connected locally,
 * i.e. via a UNIX socket, as VIR_TYPED_PARAM_STRING.
 * NOTE: This attribute is read-only and any attempt to set it will be
 * denied by daemon.
 */
# define VIR_CLIENT_INFO_UNIX_GROUP_NAME "unix_group_name"

/**
 * VIR_CLIENT_INFO_UNIX_PROCESS_ID:
 * Macro represents the client process's pid it is running with. Only
 * relevant for clients connected locally, i.e. via a UNIX socket,
 * as VIR_TYPED_PARAM_INT.
 * NOTE: This attribute is read-only and any attempt to set it will be
 * denied by daemon.
 */
# define VIR_CLIENT_INFO_UNIX_PROCESS_ID "unix_process_id"

/**
 * VIR_CLIENT_INFO_SELINUX_CONTEXT:
 * Macro represents the client's (peer's) SELinux context and this can
 * either be at socket layer or at transport layer, depending on the
 * connection type, as VIR_TYPED_PARAM_STRING.
 * NOTE: This attribute is read-only and any attempt to set it will be
 * denied by daemon.
 */
# define VIR_CLIENT_INFO_SELINUX_CONTEXT "selinux_context"
```

Function Documentation

```
int virAdmServerFree(virAdmServerPtr srv);
```

Release the server object. Libvirt uses reference counting on objects, thus any running instance is kept alive. However the data structure is considered freed and should not be used thereafter.

- *Parameters:*
  - ◇ `srv` - a client-side server object.
- *Return value* - returns 0 or -1 in case of an error.

```
int virAdmServerGetName(virAdmServerPtr srv);
```

Get the public name for the specified server.

- *Parameters:*
  - ◇ `srv` - a valid server object reference.
- *Return value* - returns a pointer to the name or NULL. The string does not need to be deallocated, since its lifetime will be the same as the lifetime of the server object.

```
int virAdmConnectListServers(virAdmConnectPtr conn,
                             virAdmServerPtr **servers,
                             unsigned int flags);
```

Collect list of all servers provided by daemon the client is connected to.

- *Parameters:*
  - ◇ `conn` - a valid daemon connection reference,
  - ◇ `servers` - pointer to a list to store an array containing objects or NULL if the list is not required (only number of servers is required),
  - ◇ `flags` - extra flags which are currently unused and caller should thus pass 0.
- *Return value* - returns the number of servers available on daemon side or -1 in case of a failure, setting `servers` to NULL. There is a guaranteed extra element set to NULL in the `servers` list returned to make iteration easier, excluding this extra element from the final count. Caller is responsible for calling `virAdmServerFree` on each element of the list, followed by freeing `servers`.

```
int
virAdmServerGetThreadPoolParameters(virAdmServerPtr srv,
                                    virTypedParameterPtr *params
                                    int *params,
                                    unsigned int flags);
```

Retrieve threadpool parameters from server `srv`. Upon successfull completion, `params` will be allocated automatically to hold all returned data and `nparams` will be set accordingly. Section *Macros* A describes all supported keys for data extraction from `params`.

- *Parameters:*
  - ◇ `srv` - a valid server object reference,
  - ◇ `params` - pointer to a list of typed parameters which will be allocated to store all returned parameters,
  - ◇ `nparams` - pointer which will hold the number of parameters returned in `params`,
  - ◇ `flags` - extra flags which are currently unused and caller should thus pass 0.
- *Return value* - returns 0 on success or -1 in case of an error.

```
int
virAdmServerSetThreadPoolParameters(virAdmServerPtr srv,
                                    virTypedParameterPtr params,
                                    int params,
                                    unsigned int flags);
```

Change server threadpool parameters according to `params`. Note that some tunables are read-only, thus attempt to set them will result in a failure. Section *Macros* A describes all supported keys to achieve this task.

- *Parameters:*
  - ◇ `srv` - a valid server object reference,
  - ◇ `params` - pointer to threadpool typed parameter objects,
  - ◇ `nparams` - number of parameters in `params`,
  - ◇ `flags` - extra flags which are currently unused and caller should thus pass 0.
- *Return value* - returns 0 on success or -1 in case of an error.

```
int
virAdmConnectGetLoggingLevel(virAdmConnectPtr conn,
                             unsigned int flags);
```

Retrieve current global logging level from daemon. Currently the supported values for logging level are (numbered from the first) `DEBUG`, `INFO`, `WARNING`, and `ERROR`. Inclusion hierarchy is applied, meaning that each level also includes all levels, such that they are ranked with a higher number than the original level.

- *Parameters:*
  - ◇ `conn` - pointer to an active administration connection,
  - ◇ `flags` - extra flags which are currently unused and caller should thus pass 0.
- *Return value* - returns the numeric logging level representation or -1 in case of an error.

```
int
virAdmConnectSetLoggingLevel(virAdmConnectPtr conn,
                             unsigned int level,
                             unsigned int flags);
```

Set daemon's current global logging level to `level`. See `virLogPriority` constants in *Type Documentation* A for supported values.

- *Parameters:*
  - ◇ `conn` - pointer to an active administration connection,
  - ◇ `level` - desired logging level, virLogPriority defines all supported constants,
  - ◇ `flags` - extra flags which are currently unused and caller should thus pass 0.
- *Return value* - returns 0 on success or -1 in case of an error.

```
int
virAdmConnectGetLoggingFilters(virAdmConnectPtr conn,
                               char **logFilters,
                               unsigned int flags);
```

Retrieve a list of currently installed logging filter on daemon. Filters returned are contained within an automatically allocated string and delimited by spaces. The format of each filter conforms to the format described in daemon's configuration file (e.g. `libvird.conf`). To retrieve individual filters, additional parsing needs to be done by the caller. Caller is also responsible for freeing `filters` correctly.

- *Parameters:*
  - ◇ `conn` - pointer to an active administration connection,
  - ◇ `filters` - pointer to a variable to store a string containing all currently defined logging filters on daemon (allocated automatically),
  - ◇ `flags` - extra flags which are currently unused and caller should thus pass 0.
- *Return value* - returns the number of filters returned in `filters`, or -1 in case of an error.

```
int
virAdmConnectSetLoggingFilters(virAdmConnectPtr conn,
                               char *filters,
                               unsigned int flags);
```

Redefine the existing filter or a set of filters with a new one specified in `filters`. If multiple filters are specified, they need to be delimited by spaces. The format of each filter must conform to the format described in daemon's configuration file (e.g. `libvirtd.conf`).

- *Parameters:*
  - ◇ `conn` - pointer to an active administration connection,
  - ◇ `filters` - pointer to a string containing a list of filters to be defined,
  - ◇ `flags` - extra flags which are currently unused and caller should thus pass 0.

- *Return value* - returns 0 if the new filter or the set of filters has been defined successfully, or -1 in case of an error.

```
int
virAdmConnectGetLoggingOutputs(virAdmConnectPtr conn,
                               char **outputs,
                               unsigned int flags);
```

Retrieve a list of currently installed logging outputs on daemon. Outputs returned are contained within an automatically allocated string and delimited by spaces. The format of each output conforms to the format described in daemon's configuration file (e.g. `libvird.conf`). To retrieve individual outputs, additional parsing needs to be done by the caller. Caller is also responsible for freeing `outputs` correctly.

- *Parameters:*
  - ◇ `conn` - pointer to an active administration connection,
  - ◇ `filters` - pointer to a variable to store a string containing all currently defined logging outputs on daemon (allocated automatically),
  - ◇ `flags` - extra flags which are currently unused and caller should thus pass 0.
- *Return value* - returns the number of outputs returned in `outputs`, or -1 in case of an error.

```
int
virAdmConnectSetLoggingOutputs(virAdmConnectPtr conn,
                               char *outputs,
                               unsigned int flags);
```

Redefine the existing output or a set of outputs with a new one specified in `outputs`. If multiple outputs are specified, they need to be delimited by spaces. The format of each output must conform to the format described in daemon's configuration file (e.g. `libvirtd.conf`).

- *Parameters:*
  - ◇ `conn` - pointer to an active administration connection,
  - ◇ `filters` - pointer to a string containing a list of outputs to be defined,
  - ◇ `flags` - extra flags which are currently unused and caller should thus pass 0.
- *Return value* - returns 0 if the new output or the set of outputs has been defined successfully, or -1 in case of an error.

```
int
virAdmServerGetClientLimits(virAdmServerPtr srv,
                            virTypedParameterPtr *params,
                            int *nparams,
                            unsigned int flags);
```

Retrieve client limits from server `srv`. These include the current and maximum number of clients connected to server `srv`, as well as the current and maximum number of clients connected to server `srv` still waiting for authentication. Keys to `params` objects are mentioned in Section *Macros* A.

- *Parameters:*
  - ◇ `srv` - a valid server object reference,
  - ◇ `params` - pointer to list of typed parameters which will be allocated to store all returned limits,
  - ◇ `nparams` - pointer which will hold the number of parameters returned in `params`,
  - ◇ `flags` - extra flags which are currently unused and caller should thus pass 0.
- *Return value* - returns 0 on success, allocating `params` to size returned in `nparams`, or -1 in case of an error.

```
int
virAdmServerSetClientLimits(virAdmServerPtr srv,
                            virTypedParameterPtr params,
                            int *nparams,
                            unsigned int flags);
```

Change client limits on server `srv`. Caller is responsible for allocating `params` prior to calling this method. Note that some attributes are read-only, so consult Section *Macros* A to see which keys are supported for `params`.

- *Parameters:*
  - ◇ `srv` - a valid server object reference,
  - ◇ `params` - pointer to client limits object,
  - ◇ `nparams` - number of parameters in `params`,
  - ◇ `flags` - extra flags which are currently unused and caller should thus pass 0.
- *Return value* - returns 0 if the limits have been changed successfully or -1 in case of an error.

```
int virAdmServerLookupClient(virAdmServerPtr srv,
                             unsigned long long id,
                             unsigned int flags);
```

Try to lookup a client on the given server based on an ID. Once the object is no longer needed, `virAdmClientFree` should be used to free the resources.

- *Parameters*
  - ◇ `srv` - a valid server object reference,
  - ◇ `id` - ID of the client to lookup on server `srv`,
  - ◇ `flags` - extra flags which are currently unused and caller should thus pass 0.
- *Return value* - returns the requested client object or NULL in case of a failure. If the client could not be found, `VIR_ERR_NO_CLIENT` error is raised.

```
int virAdmClientFree(virAdmClientPtr client);
```

Release the client object. The running instance is kept alive. The data structure is freed and should not be used thereafter.

- *Parameters:*
  - ◇ `client` - a valid client object reference,
- *Return value* - returns 0 on success, -1 on failure.

```
int virAdmClientGetID(virAdmClientPtr client);
```

Get client's unique numeric ID.

- *Parameters:*
  - ◇ `client` - a valid client object reference,
- *Return value* - returns the numeric value of client's ID or -1 in case of an error.

```
int virAdmClientGetTimestamp(virAdmClientPtr client);
```

Get client's connection time. A situatiom may happen, that some clients had connected prior to the update to admin API, thus, libvirt assigns these clients epoch time to express that it does not know when the client connected.

- *Parameters:*
  - ◇ `client` - a valid client object reference,
- *Return value* - returns client's connection timestamp (seconds from epoch in UTC) or 0 (epoch time) if libvirt does not have any information about client's connection time, or -1 in case of an error.

```
int virAdmClientGetTransport(virAdmClientPtr client);
```

Get client's connection transport type. This information can be helpful to differentiate between clients connected locally or remotely. An exception to this would be SSH which is one of libvirt's supported transports. Although SSH creates a channel between two (preferably) remote endpoints, the client process which libvirt spawns automatically on the remote side will still connect to a UNIX socket, thus becoming indistinguishable from any other locally connected clients.

- *Parameters:*
  - ◇ `client` - a valid client object reference,
- *Return value* - returns an integer representation of the connection transport used by `client` (this will be one of `virClientTransport`) or -1 in case of an error.

```
int virAdmServerListClients(virAdmServerPtr srv,
                            virAdmClientPtr **clients,
                            unsigned int flags);
```

Collect list of all clients connected to daemon on server `srv`.

- *Parameters:*
  - ◇ `srv` - a valid server object reference,
  - ◇ `clients` - pointer to a list to store an array containing objects or NULL if the list is not required (only number of clients is required),
  - ◇ `flags` - extra flags which are currently unused and caller should thus pass 0.
- *Return value* - returns the number of clients connected to daemon on server `srv` side or -1 in case of a failure, setting `clients` to NULL. There is a guaranteed extra element set to NULL in the `clients` list returned to make iteration easier, excluding this extra element from the final count. Caller is responsible for calling `virAdmClientFree` on each element of the list, followed by freeing `clients`.

```
int virAdmClientGetInfo(virAdmClientPtr client,
                        virAdmClientInfoPtr *info,
                        unsigned int flags);
```

Extract identity information about a client. Attributes returned in `params` are mostly transport-dependent, i.e. some attributes including client process's pid, gid, uid, or remote side's socket address are only available for a specific connection type—local or remote. Other identity attributes like authentication method used (if authentication is enabled on the remote host) or SELinux context are independent of the connection transport.

- *Parameters:*
  - ◇ `client` - a valid client object reference,
  - ◇ `info` - pointer to an info structure to store the returned identity information (allocated automatically),
  - ◇ `flags` - extra flags which are currently unused and caller should thus pass 0.
- *Return value* - returns 0 if the information has been successfully retrieved or -1 in case of an error.

```
int virAdmClientClose(virAdmClientPtr client,
                      unsigned int flags);
```

Close `client`'s connection to daemon forcefully.

- *Parameters:*
  - ◇ `client` - a valid client object reference,
  - ◇ `flags` - extra flags which are currently unused and caller should thus pass 0.
- *Return value* - returns 0 if the daemon's connection with `client` was closed successfully or -1 in case of an error.

Appendix B

Libvirt Administration API Usage Examples

This section consists of practical examples (written in C language) of usage of the key administration interfaces.

B.1 Listing Available Servers on Daemon

```
#include<stdio.h>
#include<stdlib.h>
#include<libvirt/libvirt-admin.h>

int main(void)
{
    int ret = -1;
    virAdmConnectPtr conn = NULL;
    virAdmServerPtr *servers = NULL;    /* where to store the servers */
    virAdmServerPtr *tmp = NULL;
    size_t i = 0;
    int count = 0;

    /* first, open a connection to the daemon */
    if (!(conn = virAdmConnectOpen(NULL, 0)))
        goto cleanup;

    /* get the available servers on the default daemon - libvirtd */
    if ((count = virAdmConnectListServers(conn, &servers, 0)) < 0)
        goto cleanup;

    /* let’s print the available servers, we have 2 options how to interate
     * over the returned list, use count as the boundary or use the fact
     * that servers are guaranteed to contain 1 extra element NULL;
     * this example uses the second option
     */
    printf(" %-15s\n", "Server name");
```

```
    printf("---------------\n");
    for (tmp = servers; *tmp; tmp++)
        printf(" %-15s\n", virAdmServerGetName(*tmp));

    ret = 0;
 cleanup:
    /* Once finished, free the list of servers and close the connection
     * properly, @conn will be deallocated automatically
     */
    for (i = 0; i < count; i++)
        virAdmServerFree(servers[i]);
    free(servers);
    virAdmConnectClose(conn);
    return ret;
}
```

examples/list_servers.c

B.2 Listing All Clients Connected To a Given Server

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<libvirt/libvirt-admin.h>

static const char *
exampleTransportToString(int transport)
{
    const char *str = NULL;

    switch ((virClientTransport) transport) {
    case VIR_CLIENT_TRANS_UNIX:
        str = "unix";
        break;
    case VIR_CLIENT_TRANS_TCP:
        str = "tcp";
        break;
    case VIR_CLIENT_TRANS_TLS:
        str = "tls";
        break;
    }

    return str ? str : "unknown";
}

static char *
exampleGetTimeStr(time_t then)
```

```
{
    char *ret = NULL;
    struct tm timeinfo;

    if (!localtime_r(&then, &timeinfo))
        return NULL;

    if (!(ret = calloc(64, sizeof(char))))
        return NULL;

    if (strftime(ret, 64, "%Y-%m-%d %H:%M:%S%z",
                 &timeinfo) == 0) {
        free(ret);
        return NULL;
    }

    return ret;
}

int main(int argc, char **argv)
{
    int ret = -1;
    virAdmConnectPtr conn = NULL;
    virAdmServerPtr srv = NULL;    /* which server list the clients from */
    virAdmClientPtr *clients = NULL;    /* where to store the servers */
    size_t i = 0;
    int count = 0;

    if (argc != 2) {
        fprintf(stderr, "One argument, specifying the server to list "
                "connected clients for, is expected\n");
        return -1;
    }

    /* first, open a connection to the daemon */
    if (!(conn = virAdmConnectOpen(NULL,0)))
        return -1;

    /* first a virAdmServerPtr handle is necessary to obtain, that is done
     * by doing a lookup for specific server, argv[1] holds the server name
     */
    if (!(srv = virAdmConnectLookupServer(conn, argv[1], 0)))
        goto cleanup;

    /* now get the currently connected clients to server srv */
    if ((count = virAdmServerListClients(srv, &clients, 0)) < 0)
        goto cleanup;
```

```
    /* let's print the currently connected clients and some basic info about
     * them, we have 2 options how to interate over the returned list,
     * use count as the boundary or use the fact that clients are guaranteed
     * to contain 1 extra element NULL;
     * this example uses the first option
     */
    printf(" %-5s %-15s %-15s\n%s\n", "Id", "Transport", "Connected since",
           "--------------------------------------------------");

    for (i = 0; i < count; i++) {
        virAdmClientPtr client = clients[i];
        unsigned long long id = virAdmClientGetID(client);
        int transport = virAdmClientGetTransport(client);
        char * timestr = NULL;
        if (!(timestr =
                exampleGetTimeStr(virAdmClientGetTimestamp(client))))
            goto cleanup;

        printf(" %-5llu %-15s %-15s\n", id,
               exampleTransportToString(transport), timestr);
        free(timestr);
    }

    ret = 0;
 cleanup:
    /* Once finished, free the list of clients, free the server handle and
     * close the connection properly, @conn will be deallocated
     * automatically
     */
    for (i = 0; i < count; i++)
        virAdmClientFree(clients[i]);
    free(clients);
    virAdmServerFree(srv);
    virAdmConnectClose(conn);
    return ret;
}
```

examples/list_clients.c

B.3 Getting and Setting Logging Level

```
#include<stdio.h>
#include<libvirt/libvirt-admin.h>

int main(void)
{
    int ret = -1;
```

```
    virAdmConnectPtr conn = NULL;
    int level;
    char *levelstr = NULL;

    /* first, open a connection to the daemon */
    if (!(conn = virAdmConnectOpen(NULL,0)))
        return -1;

    /* get the current global logging level */
    if ((level = virAdmConnectGetLoggingLevel(conn, 0)) < 0)
        goto cleanup;

    /* let's print the level */
    switch ((virLogPriority) level) {
    case VIR_LOG_DEBUG:
        levelstr = "DEBUG";
        break;
    case VIR_LOG_INFO:
        levelstr = "INFO";
        break;
    case VIR_LOG_WARN:
        levelstr = "WARNING";
        break;
    case VIR_LOG_ERROR:
        levelstr = "ERROR";
        break;
    default:
        fprintf(stderr, "Unrecognized logging level '%d'\n", level);
        goto cleanup;
    }

    printf("Current logging level: %d\n", level);

    /* now, change the level to some other value, INFO for instance */
    if (virAdmConnectSetLoggingLevel(conn, VIR_LOG_INFO, 0) < 0)
        goto cleanup;

    ret = 0;
 cleanup:
    /* Once finished close the connection properly, @conn will be deallocated
     * automatically
     */
    virAdmConnectClose(conn);
    return ret;
}
```

examples/log_level.c

B.4 Getting and Setting Logging Filters

```
#include<stdio.h>
#include<stdlib.h>
#include<libvirt/libvirt.h>
#include<libvirt/libvirt-admin.h>
#include<libvirt/virterror.h>

int main(int argc, char **argv)
{
    int ret = -1;
    virAdmConnectPtr conn = NULL;
    char *filters = NULL;

    if (argc != 2) {
        fprintf(stderr, "One argument specifying filters is required\n");
        return -1;
    }

    /* first, open a connection to the daemon */
    if (!(conn = virAdmConnectOpen(NULL,0)))
        return -1;

    /* get the current logging filters */
    if (virAdmConnectGetLoggingFilters(conn, &filters, 0) < 0)
        goto cleanup;

    printf("Current logging filters: %s\n", filters);

    /* now, change the filters to some other value */
    if (virAdmConnectSetLoggingFilters(conn, argv[1], 0) < 0)
        goto cleanup;

    ret = 0;
 cleanup:
    /* Once finished close the connection properly, @conn will be deallocated
     * automatically
     */
    virAdmConnectClose(conn);
    free(filters);
    return ret;
}
```

examples/log_filters.c

B.5 Getting and Setting Logging Outputs

```
#include<stdio.h>
#include<stdlib.h>
#include<libvirt/libvirt.h>
#include<libvirt/libvirt-admin.h>
#include<libvirt/virterror.h>

int main(int argc, char **argv)
{
    int ret = -1;
    virAdmConnectPtr conn = NULL;
    char *outputs = NULL;

    if (argc != 2) {
        fprintf(stderr, "One argument specifying outputs is required\n");
        return -1;
    }

    /* first, open a connection to the daemon */
    if (!(conn = virAdmConnectOpen(NULL,0)))
        return -1;

    /* get the current logging filters */
    if (virAdmConnectGetLoggingOutputs(conn, &outputs, 0) < 0)
        goto cleanup;

    printf("Current logging outputs: %s\n", outputs);

    /* now, change the filters to some other value */
    if (virAdmConnectSetLoggingOutputs(conn, argv[1], 0) < 0)
        goto cleanup;

    ret = 0;
 cleanup:
    /* Once finished close the connection properly, @conn will be deallocated
     * automatically
     */
    virAdmConnectClose(conn);

    /* we're responsible for freeing the result once we don't need it */
    free(outputs);
    return ret;
}
```

examples/log_outputs.c

B.6 Getting and Setting Server Threadpool Parameters

```
#include<stdio.h>
#include<stdlib.h>
#include<libvirt/libvirt-admin.h>

int main(int argc, char **argv)
{
    int ret = -1;
    virAdmConnectPtr conn = NULL;
    virAdmServerPtr srv = NULL;     /* which server to work with */
    virTypedParameterPtr params = NULL;
    int nparams = 0;
    size_t i;

    if (argc != 2) {
        fprintf(stderr, "One argument specifying the server which to work "
                "with is expected\n");
        return -1;
    }

    /* first, open a connection to the daemon */
    if (!(conn = virAdmConnectOpen(NULL,0)))
        goto cleanup;

    /* a server handle is necessary before any API regarding threadpool
     * parameters can be issued
     */
    if (!(srv = virAdmConnectLookupServer(conn, argv[1], 0)))
        goto cleanup;

    /* get the current threadpool parameters */
    if (virAdmServerGetThreadPoolParameters(srv, &params, &nparams, 0) < 0)
        goto cleanup;

    for (i = 0; i < nparams; i++)
        printf("%-15s: %d\n", params[i].field, params[i].value.ui);

    virTypedParamsFree(params, nparams);
    params = NULL;
    nparams = 0;

    /* set minWorkers to 10, maxWorkers to 15 and prioWorkers to 10 */
    int maxparams = 0;
    if (virTypedParamsAddUInt(&params, &nparams, &maxparams,
                              VIR_THREADPOOL_WORKERS_MIN, 10) < 0 ||
        virTypedParamsAddUInt(&params, &nparams, &maxparams,
                              VIR_THREADPOOL_WORKERS_MAX, 15) < 0 ||
```

```
        virTypedParamsAddUInt(&params, &nparams, &maxparams,
                              VIR_THREADPOOL_WORKERS_PRIORITY, 10) < 0)
        goto cleanup;

    /* now, change the threadpool settings to some different values */
    if (virAdmServerSetThreadPoolParameters(srv, params, nparams, 0) < 0)
        goto cleanup;

    ret = 0;
 cleanup:
    virTypedParamsFree(params, nparams);

    /* Once finished deallocate the server handle and close the connection
     * properly, @conn will be deallocated automatically
     */
    virAdmServerFree(srv);
    virAdmConnectClose(conn);
    return ret;
}
```

examples/threadpool_params.c

B.7 Getting and Setting Client Limits on Server

```
#include<stdio.h>
#include<stdlib.h>
#include<libvirt/libvirt-admin.h>

int main(int argc, char **argv)
{
    int ret = -1;
    virAdmConnectPtr conn = NULL;
    virAdmServerPtr srv = NULL;     /* which server to work with */
    virTypedParameterPtr params = NULL;
    int nparams = 0;
    size_t i;

    if (argc != 2) {
        fprintf(stderr, "One argument specifying the server which to work "
                "with is expected\n");
        return -1;
    }

    /* first, open a connection to the daemon */
    if (!(conn = virAdmConnectOpen(NULL,0)))
        goto cleanup;
```

```
    /* a server handle is necessary before any API regarding threadpool
     * parameters can be issued
     */
    if (!(srv = virAdmConnectLookupServer(conn, argv[1], 0)))
        goto cleanup;

    /* get the current client limits */
    if (virAdmServerGetClientLimits(srv, &params, &nparams, 0) < 0)
        goto cleanup;

    for (i = 0; i < nparams; i++)
        printf("%-15s: %d\n", params[i].field, params[i].value.ui);

    virTypedParamsFree(params, nparams);
    params = NULL;
    nparams = 0;

    /* set nclients_max to 100 and nclients_unauth_max to 20 */
    int maxparams = 0;
    if (virTypedParamsAddUInt(&params, &nparams, &maxparams,
                              VIR_SERVER_CLIENTS_MAX, 100) < 0 ||
        virTypedParamsAddUInt(&params, &nparams, &maxparams,
                              VIR_SERVER_CLIENTS_UNAUTH_MAX, 20) < 0)
        goto cleanup;

    /* now, change the client limits on the server */
    if (virAdmServerSetClientLimits(srv, params, nparams, 0) < 0)
        goto cleanup;

    ret = 0;
 cleanup:
    virTypedParamsFree(params, nparams);

    /* Once finished deallocate the server handle and close the connection
     * properly, @conn will be deallocated automatically
     */
    virAdmServerFree(srv);
    virAdmConnectClose(conn);
    return ret;
}
```

examples/client_limits.c

B.8 Retrieving a Client's Identity Information

```
#define _GNU_SOURCE
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<string.h>
#include<libvirt/libvirt-admin.h>

static const char *
exampleTransportToString(int transport)
{
    const char *str = NULL;

    switch ((virClientTransport) transport) {
    case VIR_CLIENT_TRANS_UNIX:
        str = "unix";
        break;
    case VIR_CLIENT_TRANS_TCP:
        str = "tcp";
        break;
    case VIR_CLIENT_TRANS_TLS:
        str = "tls";
        break;
    }

    return str ? str : "unknown";
}

static char *
exampleGetTimeStr(time_t then)
{
    char *ret = NULL;
    struct tm timeinfo;

    if (!localtime_r(&then, &timeinfo))
        return NULL;

    if (!(ret = calloc(64, sizeof(char))))
        return NULL;

    if (strftime(ret, 64, "%Y-%m-%d %H:%M:%S%z",
                 &timeinfo) == 0) {
        free(ret);
        return NULL;
    }

    return ret;
```

```
}

static char *
exampleGetTypedParamValue(virTypedParameterPtr item)
{
    int ret = 0;
    char *str = NULL;

    switch (item->type) {
    case VIR_TYPED_PARAM_INT:
        ret = asprintf(&str, "%d", item->value.i);
        break;

    case VIR_TYPED_PARAM_UINT:
        ret = asprintf(&str, "%u", item->value.ui);
        break;

    case VIR_TYPED_PARAM_LLONG:
        ret = asprintf(&str, "%lld", item->value.l);
        break;

    case VIR_TYPED_PARAM_ULLONG:
        ret = asprintf(&str, "%llu", item->value.ul);
        break;

    case VIR_TYPED_PARAM_DOUBLE:
        ret = asprintf(&str, "%f", item->value.d);
        break;

    case VIR_TYPED_PARAM_BOOLEAN:
        str = strdup(item->value.b ? "yes" : "no");
        break;

    case VIR_TYPED_PARAM_STRING:
        str = strdup(item->value.s);
        break;

    default:
        fprintf(stderr, "unimplemented parameter type %d\n", item->type);
        return NULL;
    }

    return str;
}

int main(int argc, char **argv)
{
    int ret = -1;
```

```
virAdmConnectPtr conn = NULL;
virAdmServerPtr srv = NULL; /* server the client is connected to */
virAdmClientPtr clnt = NULL;  /* which client get identity info for */
virTypedParameterPtr params = NULL;  /* where to store identity info */
int nparams = 0;
size_t i = 0;
char *timestr = NULL;

if (argc != 3) {
    fprintf(stderr, "Two arguments, first specifying the server the "
            "client is connected to and second, specifying the "
            "client's  ID for which identity information should be "
            "retrieved, are expected\n");
    return -1;
}

/* first, open a connection to the daemon */
if (!(conn = virAdmConnectOpen(NULL, 0)))
    return -1;

/* first a virAdmServerPtr handle is necessary to obtain, that is done "
 * by doing a lookup for specific server, argv[1] holds the server name
 */
if (!(srv = virAdmConnectLookupServer(conn, argv[1], 0)))
    goto cleanup;

/* next, virAdmClientPtr handle is necessary to obtain, that is done by
 * doing a lookup on a specific server, argv[2] holds the client's ID
 */
if (!(clnt = virAdmServerLookupClient(srv,
                                      strtoll(argv[2], NULL, 10), 0)))
    goto cleanup;

/* finally, retrieve clnt's identity information */
if (virAdmClientGetInfo(clnt, &params, &nparams, 0) < 0)
    goto cleanup;

/* this information is provided by the client object itself, not by
 * typed params container; it is unnecessary to call virAdmClientGetInfo
 * if only ID, transport method, and timestamp are requested
 */
if (!(timestr = exampleGetTimeStr(virAdmClientGetTimestamp(clnt))))
    goto cleanup;

printf("%-15s: %llu\n", "id", virAdmClientGetID(clnt));
printf("%-15s: %s\n", "connection_time", timestr);
printf("%-15s: %s\n", "transport",
         exampleTransportToString(virAdmClientGetTransport(clnt)));
```

```
    /* this is the actual identity information retrieved in typed params
     * container
     */
    for (i = 0; i < nparams; i++) {
        char *str = NULL;
        if (!(str = exampleGetTypedParamValue(&params[i])))
            goto cleanup;
        printf("%-15s: %s\n", params[i].field, str);
        free(str);
    }

    ret = 0;
 cleanup:
    /* Once finished, free the typed params container, server and client
     * handles and close the connection properly, @conn will be deallocated
     * automatically
     */
    virTypedParamsFree(params, nparams);
    virAdmClientFree(clnt);
    virAdmServerFree(srv);
    virAdmConnectClose(conn);
    free(timestr);
    return ret;
}
```

examples/client_info.c

B.9 Closing a Client's Connection Forcefully

```
#include<stdio.h>
#include<stdlib.h>
#include<libvirt/libvirt.h>
#include<libvirt/libvirt-admin.h>

int main(void)
{
    int ret = -1;
    virAdmConnectPtr conn1 = NULL; /* admin connection */
    virConnectPtr conn2 = NULL;  /* libvirt standard connection */
    virAdmServerPtr srv = NULL;  /* server the client connected to */
    virAdmClientPtr clnt = NULL; /* which client to disconnect */

    /* first, open a standard libvirt connection to the daemon */
    if (!(conn2 = virConnectOpen(NULL)))
        return -1;
```

```
    /* next, open an admin connection that will be used to disconnect the
     * standard libvirt client
     */
    if (!(conn1 = virAdmConnectOpen(NULL,0)))
        goto cleanup;

    /* a virAdmServerPtr handle is needed, therefore server lookup is
     * performed
     */
    if (!(srv = virAdmConnectLookupServer(conn1, "libvirtd", 0)))
        goto cleanup;

    /* a virAdmClientPtr handle is also necessary, so lookup for client is
     * performed as well
     */
    if (!(clnt = virAdmServerLookupClient(srv, 1, 0)))
        goto cleanup;

    /* finally, use the client handle to disconnect the standard libvirt
     * client from libvirtd daemon
     */
    if (virAdmClientClose(clnt, 0) < 0)
        goto cleanup;

    ret = 0;
 cleanup:
    /* Once finished, both server and client handles need to be freed and
     * both connections @conn1 and @conn2 should be closed to free the
     * memory.
     * NOTE: Although @conn2 has been disconnected, unlike disconnecting by
     * calling virConnectClose which closes the connection voluntarily and
     * frees the object automatically, virAdmClientClose is a forceful
     * disconnect of another client (client can use it on itself as well).
     * Therefore no automatic deallocation of the object takes place and is
     * the callers responsibility to do so.
     */
    virAdmClientFree(clnt);
    virAdmServerFree(srv);
    virAdmConnectClose(conn1);
    virConnectClose(conn2);
    return ret;
}
```

examples/client_close.c

Appendix C

Contents of the CD

- `/tests` - testing toolset including the test suite itself,
- `/commits` - all the important commits that were made, including the application interfaces themselves, some necessary refactors, as well as minor bug fixes,
- `/examples` - C language examples demonstrating the usage of all the implemented APIs,
- `/src` - cloned github repository including the 'logging' branch which has the logging interfaces applied, and
- `/doc` - electronic version of the thesis.