

**FACULTY
OF MATHEMATICS
AND PHYSICS**
Charles University

MASTER THESIS

Jan Výkruta

Modular backup system for virtualization

Department of Software Engineering

Supervisor of the master thesis: RNDr. Jakub Yaghob, Ph.D.

Study programme: Computer Science

Study branch: Software and Data Engineering

Prague 2022

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In date
Author's signature

Title: Modular backup system for virtualization

Author: Jan Výkruta

Department: Department of Software Engineering

Supervisor: RNDr. Jakub Yaghob, Ph.D., Department of Software Engineering

Abstract: Backing up of a virtual machine is the act of capturing a static image, that can be subsequently used for restoration in case of the target machine failure. We contribute a pilot implementation of a modular system suitable for virtual machine backup and restoration on the VMware vSphere virtualization stack. The application can be extended with support for other platforms. It supports deduplication and transformation of the backed up data, automatic periodic backups and consolidation of the backups according to a policy. Moreover, the appliance stores the backed up data into a user-defined location. The software can be extended and modified in all of the above cases. Finally, we provide documentation of the system, suggestions for possible extensions and a guide to implement them.

Keywords: virtualization backup deduplication

Contents

Introduction	5
1 Preliminaries	7
1.1 Concepts and terminology	7
1.1.1 Virtualization	7
1.1.2 Virtual Machine	7
1.1.3 Hypervisor	7
1.1.4 Backup actions	8
1.1.5 Hypervisor inventory	8
1.1.6 Delta backup	8
1.1.7 Backup chain	8
1.1.8 Backup tree	8
1.1.9 Disk chain and tree	9
1.1.10 Deduplication techniques	9
1.2 VMware vSphere specifics	9
1.3 Backup techniques	10
1.3.1 Backup	10
1.3.2 Restoration	11
1.3.3 Consolidation	11
2 Analysis	12
2.1 Autonomous operation of the system	12
2.2 System configuration	12
2.3 Transactional behavior	12
2.4 Virtualization stacks	13
2.5 Changed blocks computation during a delta backup	13
2.6 Operation precedence	14
2.7 Operation concurrency	14
2.8 Backup trees	15
2.9 Restoration method	15
3 System design	16
3.1 Architecture overview	16
3.2 Technology stack	16
3.2.1 vCenter communication	17
3.2.2 Accessing disk data	18
3.2.3 Language choice	18
3.2.4 Additional libraries	19
3.3 Data persistence	19
3.4 Daemon process architecture	20
3.4.1 Daemon core	20
3.4.2 Client communication	21
3.4.3 Scheduler	22
3.4.4 State manager	22
3.4.5 Tracked info provider	22

3.4.6	Inventory info provider	22
3.4.7	Worker groups	22
3.5	Concurrency control	22
3.6	Requests	23
3.7	Workers	24
3.8	Responses	24
3.9	Jobs	24
3.10	Data model	25
3.10.1	Inventory	25
3.10.2	Virtual machine	26
3.10.3	Backup	26
3.10.4	Disk	27
3.10.5	Data block	27
3.10.6	Polymorphic virtual machine backup configuration	27
3.10.7	Disk backup configuration	28
3.11	Request processing	28
3.12	Virtualization stack integration	29
3.13	Data modification	29
3.14	Logging	30
3.15	Client communication channel	30
3.16	Configuration	30
3.17	Defects	30
3.17.1	Database limitations	30
4	Implementation	32
4.1	Operation overview	32
4.1.1	Command line arguments	32
4.1.2	Configuration files	32
4.1.3	Database operation	33
4.1.4	Processes and communication fabric	34
4.1.5	Termination	34
4.2	Main operation	35
4.2.1	Client communication	36
4.2.2	Daemon core	36
4.2.3	Worker groups	37
4.2.4	State manager	37
4.2.5	Inventory and Tracked info providers	37
4.2.6	Scheduler	37
4.3	Backup algorithms	38
4.3.1	Backup	38
4.3.2	Restoration	41
4.3.3	Cleanup	42
4.3.4	Consolidation	43
4.4	Requests and Responses	44
4.5	Implementation issues	44
4.5.1	vSphere changed blocks tracking	45
4.5.2	vSphere transfer modes	45
4.5.3	vSphere NVRAM operations	45

4.5.4	vSphere virtual machine configuration	46
4.5.5	vCenter communication	47
4.5.6	Disk chain iterator	47
4.5.7	Installation	47
4.5.8	Bindings	48
4.5.9	Communication between core and client handling process .	49
4.6	Software extensibility	49
4.6.1	Mixin	49
4.6.2	Virtualization stacks	50
4.6.3	Database engine change	51
4.6.4	Scheduler	51
4.6.5	Block and Buffer adaptors	52
4.6.6	Schedule and Retention policies	52
4.6.7	Alternative disk data storage	52
4.6.8	Consolidation algorithms	53
4.6.9	Parallelization at the block level	53
4.6.10	Parallel cleaners	54
4.6.11	Parallelization at the disk level	54
4.6.12	Runtime configuration changes	54
4.7	New vSphere version	54
4.8	Implementation defects	55
4.8.1	pydantic	55
4.8.2	Database commit ordering bug	55
4.8.3	Testing	56
5	Comparison	57
5.1	Deployment options	57
5.2	Supported platforms	58
5.3	Data deduplication	59
5.4	Multiple storage support	59
5.5	Agentless backup	60
5.6	File-based backup	60
5.7	Compression and Encryption	60
5.8	Extensibility	60
	Conclusion	61
	Bibliography	63
	List of Figures	64
	List of Tables	66
	List of Abbreviations	67
A	Attachments	68
A.1	User manual	68
A.1.1	Prerequisites	68
A.1.2	VDDK acquisition	68

A.1.3	VDDK installation	68
A.1.4	Installation	69
A.1.5	Configuration	69
A.1.6	Examples	70
A.1.7	Database preparation	70
A.1.8	Running	71
A.1.9	Monitoring	71
A.1.10	Control	71
A.1.11	Requests	72
A.1.12	Responses	72
A.1.13	vmbackupctl	72
A.1.14	Tests	73

Introduction

The significance of hardware virtualization has grown in recent years, for it offers considerable saving opportunities while providing a scalable computation platform. There are various techniques to prevent data loss when using virtualization. While these techniques provide a solid guarantee a data loss is not going to occur, they do not protect against data corruption caused by critical failures of software ran in the virtualized machines. To mitigate the repercussions of such failures, virtual machine backups can be employed. There are multiple providers of such solutions, albeit the solutions are proprietary and lacking in functionality. Certain virtualization stacks, such as VMware vSphere, used to incorporate virtual machine backup as part of the solution, although, the functionality was discontinued.

Thesis Goal

Our goal is to develop a virtual machine backup software that is modular and open to modifications. Our software should be simple to extend and provide powerful configuration options. In this work, we aim to contribute the following:

- Management of the performed backups.
- Scheduling of automatic operations. The scheduling mechanism should be universal, although, we only intend to automate backups in the pilot implementation.
- Various simple scheduling and backup retention policies.
- Ability to save the backed up data in a user-defined way.
- Ability to transform the backed up data using custom transformers. The intention is to allow implementation of data encryption or compression. There are no plans to implement the concrete transformers in the pilot implementation.
- The backup functionality proper.
- The functionality to restore a backup into a suitable hypervisor inventory.
- Deduplication of the backed up data.
- Consolidation of backups.
- Abstractions allowing the use of distinct virtualization stacks.
- Pilot implementation of the modules needed to run the application with the VMware vSphere stack.

Thesis Outline

The thesis is divided into 5 chapters and an attachment.

In Chapter 1 we provide technical background and terms used throughout the thesis. Moreover, we describe the algorithms a backup system should implement.

Chapter 2 analyses problems arising during the backup, restoration, and consolidation processes. We describe the issues, analyze their possible solutions, state which solution we opted for, and explain our reasons.

Chapter 3 focuses on the design of our system. We guide through the design and interactions of the individual components, and clarify our choices.

In chapter 4 we detail the implementation. We outline significant implementation complications and their solutions, illustrate available extension sites, and point out modifications needed in case of an update to the VMware vSphere stack.

Chapter 5 compares our software with available alternatives.

Attachment A.1 contains a user manual.

1. Preliminaries

In this chapter, we aim to provide technical background and define the terms used throughout the thesis. First, we introduce virtual machines and related technologies. Next, we present VMware-specific concepts and terminology. Finally, we detail what techniques are used during virtual machine backup, restoration, and consolidation.

1.1 Concepts and terminology

This section contains an outline of the used technology, its terminology.

1.1.1 Virtualization

“Virtualization is a process that allows for more efficient utilization of physical computer hardware and is the foundation of cloud computing.

Virtualization uses software to create an abstraction layer over computer hardware that allows the hardware elements of a single computer—processors, memory, storage and more—to be divided into multiple virtual computers, commonly called virtual machines (VMs). Each VM runs its own operating system (OS) and behaves like an independent computer, even though it is running on just a portion of the actual underlying computer hardware.” [1]

1.1.2 Virtual Machine

“A virtual machine (VM) is a virtual environment that functions as a virtual computer system with its own CPU, memory, network interface, and storage, created on a physical hardware system (located off- or on-premises). Software called a hypervisor separates the machine’s resources from the hardware and provisions them appropriately, so they can be used by the VM.” [2]

1.1.3 Hypervisor

“A hypervisor is software that creates and runs virtual machines. A hypervisor, sometimes called a virtual machine monitor (VMM), isolates the hypervisor operating system and resources from the virtual machines and enables the creation and management of those VMs.

The physical hardware, when used as a hypervisor, is called the host, while the many VMs that use its resources are guests.

The hypervisor treats resources—like CPU, memory, and storage—as a pool that can be easily reallocated between existing guests or to new virtual machines.

All hypervisors need some operating system-level components—such as a memory manager, process scheduler, input/output (I/O) stack, device drivers, security manager, a network stack, and more—to run VMs.

The hypervisor gives each virtual machine the resources that have been allocated and manages the scheduling of VM resources against the physical resources. The physical hardware still does the execution, so the CPU is still executing CPU

instructions as requested by the VMs, for example, while the hypervisor manages the schedule.

A type 1 hypervisor, also referred to as a native or bare metal hypervisor, runs directly on the host's hardware to manage guest operating systems. It takes the place of a host operating system, and VM resources are scheduled directly to the hardware by the hypervisor. A type 2 hypervisor is also known as a hosted hypervisor, and is run on a conventional operating system as a software layer or application.

It works by abstracting guest operating systems from the host operating system. VM resources are scheduled against a host operating system, which is then executed against the hardware.” [3]

1.1.4 Backup actions

Virtual machine backup is a process where the data of the machine is saved for future use. Restoration is a process where a virtual machine backup is used to create a new copy of the backed up machine. Finally, backup consolidation is a process where one or more virtual machine backups are merged into a single or multiple different backups. Such a situation usually arises when older backups get deleted.

1.1.5 Hypervisor inventory

An inventory of a hypervisor is all entities managed by the hypervisor that are exposed through an API. An example of such entities are virtual machines, datacenters, virtualization hosts, clusters, data storage and directories organizing them.

1.1.6 Delta backup

A delta backup saves only data that has changed since a certain date, usually when another backup was created. We limit ourselves to occasions when a backup was taken throughout the text.

1.1.7 Backup chain

A delta backup chain is a collection of backups that have a parent-child relationship. A child backup does not contain a whole virtual machine image, instead, it has to supplement the missing data using its parent's chain. A child backup only stores the data that has changed since the data in its parent's chain was captured. In this case, the consolidation operation merges into a single backup.

1.1.8 Backup tree

A backup tree is a generalization of the backup chain. The parent-child relationship is replaced with a parent-children relationship, which allows multiple backups to depend on a single backup. In this case, the consolidation operation merges into a single or multiple backups.

1.1.9 Disk chain and tree

A disk chain is the same relationship among disks as the backup chain is among backups. In fact, when two backups are in a relationship, their corresponding disks are usually—but not necessarily—also in a relationship. A disk tree is analogous to the disk chain.

1.1.10 Deduplication techniques

A disk level deduplication is a technique that only backs up data of the disk that has changed since a certain date.

A backup destination level deduplication checks whether there is data in the destination corresponding to the data it is about to back up. If the same data is present, albeit belonging to a different disk or even machine, it is reused for the current backup.

1.2 VMware vSphere specifics

VMware vSphere is an enterprise proprietary virtualization stack offered by the company VMware. The following components make up the stack:

- ESXi, an enterprise, type-1 hypervisor, and
- vCenter server, a centralized management utility of ESXi hosts, virtual machines running on the hosts, and any supporting components, such as virtual networking.

Hosts running ESXi are registered with the vCenter server. The server provides their management and grouping into resource groups. Moreover, the server administers data storage, networking and virtual machine migrations between hosts called vMotion. The vMotion technology arranges fail tolerance and load balancing migrations.

The vCenter server can be communicated with using a SOAP API. We use the library `pyVmomi`¹ to consume the API.

Virtual storage data can be transferred using the VDDK toolkit². The toolkit is a proprietary software. The repository of the official VMware wrapper of the toolkit [4] claims that the toolkit is free to use for personal and internal use and that redistribution requires a license. The VDDK manual [5] establishes the following:

After you use the VDDK to develop software applications that run on VMware platform products, you might need to repackage library components that are compiled into your software. To qualify for VDDK redistribution, you must be in the VMware TAP program at Standard level or above, and sign a redistribution agreement. Contact your VMware alliance manager to request the VDDK redistribution agreement. VMware would like to know how you use the VDDK, in

¹<https://github.com/vmware/pyvmomi>

²<https://developer.vmware.com/web/sdk/7.0/vddk>

what products you plan to redistribute it, your company name, and your contact information.

The aforementioned redistribution limitation does not affect our software, for we do not distribute the library with our work, and instead expect the user to provide it themselves.

1.3 Backup techniques

In this section, we describe the algorithms employed by a backup system.

1.3.1 Backup

To perform a backup of a virtual machine, the hypervisor has to take a quiesced snapshot of the system. Quiescing instructs the operating system of the guest to flush any important data to its disks so that the snapshot provides a consistent static image of the environment while the virtual machine continues to run. The software runs all the following operations against the state captured by the snapshot. The execution of the virtual machine is completely unaffected.

The appliance saves the configuration of the virtual machine to enable creation of its copy during a subsequent restoration. Any extra needed data is saved, for example, for the vSphere platform, we need to preserve the “NVRAM” (virtual ROM) data of the machine.

We follow with the disk data backup. We save the metadata associated with the disks and move to the data backup proper. There are two situations that can arise:

- We are creating the very first backup.
- There is an already made backup.

In the former case, we perform a full backup of the disk, where all data of the disks is saved.

In the latter case, we attempt to perform a delta backup. A delta backup saves only data that has changed since a certain date. We limit ourselves to the dates when a backup was made. Moreover, we only consider the most recent backup. The delta backup mechanism is responsible for calculating which blocks have changed. The computation procedure can be conducted two ways:

- We compare the data saved during the most recent backup with the current data. This is a platform-agnostic approach that incurs massive data transfer overhead. On the other hand, the mechanism can be reused by the implementation of a new stack, and generalized to compare the current data with the data of all previous backups, which allows it to pick the most suitable ancestor instead of the most recent one.
- We instruct the hypervisor to keep track of the changed blocks. This approach provides the highest performance. However, we are limited to hypervisors that are capable of the tracking, furthermore, we have to implement the mechanism for each new stack. The procedure might also fail under

certain conditions. For example, in vSphere, when a virtual machine is reverted to a snapshot the hypervisor cannot keep track of the changed blocks, and therefore, cannot present the information when queried about the differences with a state captured before the revert occurred.

The appliance has to keep track of relationships between the backups, for the data of multiple backups comprise a full virtual machine image needed to restore the machine. Finally, the system removes the created snapshot.

1.3.2 Restoration

There are two possible backup restoration approaches:

- Make a snapshot of the corresponding virtual machine and restore the backup into the snapshot.
- Create a new virtual machine and restore the data into the machine.

We assume the latter method is used. For the rationale, see Section 2.9.

To restore a backup, the software needs to create a new virtual machine in a hypervisor inventory using the configuration saved during the backup. Any saved platform-dependent data, for example the vSphere “NVRAM” data, is then uploaded to the created virtual machine. Finally, the system loads the backed up disk data and writes the data to the disks of the machine.

The disk data loading procedure differs depending on whether a delta backup was taken or not. In the latter case, the data is simply loaded and written as is with no further processing. In the former situation, the system has to walk the backup relationship chain to determine which blocks of which backups comprise the requested virtual machine image.

1.3.3 Consolidation

Virtual machine backups are stored according to a retention policy. When the system determines that certain backups should be deleted, it has to amend any retained backups that are dependent on the deleted backups. A backup can become dependent on another backup when both are part of a delta backup chain. If such a situation arises, the appliance has to merge two or more backups into a single target backup (technically, it is possible to merge into multiple backups if backup trees are supported). To accomplish the desired behavior, the software has to walk the backup chain and merge any blocks marked for deletion, which are not covered by the target backup, into the target backup. Finally, any data not used by any retained backup has to be cleaned up.

2. Analysis

In this chapter, we reiterate the goals of our contribution and outline problems we faced during the design of the system, their possible solutions, our choices, their rationale, and ramifications.

2.1 Autonomous operation of the system

The system is expected to autonomously perform operations according to its configuration. One of our goals was the ability of the solution to perform periodical backups. Such functionality requires incorporation of a scheduling system. To this end, we can use an external scheduling tool or develop a custom mechanism.

When using an external tool, our software can have the form of a batch executable that would be executed by the scheduling system whenever a scheduling event arises. The executable would perform its task and terminate, akin to a compiler. This approach would simplify the architecture of most of the appliance considerably. There is a multitude of tools supporting scheduling, for example `cron`, however, they suffer from a shortcoming. Usage of such tools requires we split the application configuration into a part used by the scheduler, and a part used by our system.

Designing a custom scheduling solution comes with the price of the extra work and making the system more complex in overall. The resulting system is a perpetually on application (a daemon) that does all scheduling itself and can be configured with a unified configuration. Such approach provides the most flexibility. Therefore, it is the chosen approach.

Consequently, the solution has to support being started at boot time and terminated when the host system shuts down.

2.2 System configuration

The appliance shall be capable of managing multiple hypervisor inventories and provide a mechanism of assigning a virtual machine to be accessed through a particular inventory.

The system can be configured to perform automated backups of a virtual machine. The configuration allows a specification of backup frequency and retention. Moreover, each disk of a particular virtual machine can have a custom configuration specifying where its data should be backed up to, and whether they should be modified and in what way.

A user shall be allowed to issue all supported operations on-demand. Furthermore, They should be able to observe the system's state and affect its execution, for example terminate it.

2.3 Transactional behavior

The system is expected to pledge certain transactional guarantees. The solution shall not leave any artifacts when an operation fails non-critically. Moreover, it

shall not corrupt any backups or systems it communicates with, like a hypervisor inventory, during such a failure.

The system shall try to minimize the impact of critical failures. Critical failures are the following:

- Sudden host loss of power.
- Immediate termination by the host system.

Critical failures can cause the system to leave artifacts that will not be cleaned up automatically. However, the failures should not corrupt any backup data, that is still tracked by the system (i.e., if the system is suddenly terminated during deletion of a backup, it can leave behind some data of the backup, but it should not corrupt data of any non deleted backup). The artifacts should not affect the functionality of the system. Any data not tracked by the system should be safe to delete manually.

2.4 Virtualization stacks

Our system aims to provide platform-agnostic functionality that can be utilized with any virtualization stack. The software shall allow implementation of additional virtualization stacks without any changes to the platform-agnostic components. The pilot implementation will provide integration with the VMware vSphere stack, which can be used as an example.

2.5 Changed blocks computation during a delta backup

During a delta backup, the solution needs to determine which blocks have changed since another backup was taken. The computation can consider other backups in the following configuration:

- Consider all already taken backups, and perform a search using cryptographic hashing functions to determine which backup shares the most data with the current image of the machine, and should, therefore, become the ancestor to the current backup.
- Only consider the most recent backup and decide whether it should become the current backup's ancestor or not.

The former approach has the consequence of possible creation of backup trees. Backup trees complicate the design of backup management, for the consolidation mechanism would have to merge into multiple backups. However, the main disadvantage is considerable runtime of such a search while providing diminishing returns, for the situation when the most similar backup is not the immediately previous one seems unlikely. Moreover, backup chains are simpler to reason about than trees. Therefore, we do not consider the search option in our implementation.

2.6 Operation precedence

We assume a certain operation precedence during the system design. An operation has a higher precedence than another operation when the former is performed more frequently than the latter. When we are faced with a design problem decision that requires making a compromise, we favor the design benefiting the operation with the higher precedence.

The assumed operation precedence in order is as follows: backup, consolidation, cleanup, and restoration.

Backup is the most common operation, in fact, it is the main purpose of the software, and is part of its routine activity.

Consolidation usually occurs in concert with backups, for it is common to have a synchronized configuration of the two. Its precedence is roughly on the same footing as that of backup.

Cleanup can be user requested, issued when a failure emerges, or when used by consolidation. The former two are rare behavior, while the latter is as frequent as consolidations. The cleanup operation has a lower precedence because it is for the most part much simpler than its predecessors.

Consolidation and cleanup are more generic mechanisms than backup and restoration, for they do not need any interaction with a hypervisor. Therefore, only the most generic and simplest consolidation implementation will be provided, while leaving space for data storage-specific optimizations, whereas, the cleanup mechanism must be implemented fully for every data storage type.

Restoration is the least common action, only issued by a user when a failure in a backed up virtual machine arises.

2.7 Operation concurrency

The system should be capable of concurrent operations. While there is technically no limitation on available concurrency, we choose to disallow concurrent operations on the same virtual machine. The major reason is a technical limitation complicating the design severely, if we allow any modifying operations to run concurrently with other modifying or critical operations. See Section 3.17.1 for details and possible partial mitigation.

We can easily make the decision, for not much is lost by this limitation. A situation in which two or more affected operations would take place concurrently is rare.

Two simultaneous backups can only arise because of a misconfiguration or to service a user request that arrives at a similar point in time, a regular backup is scheduled. There is no loss in functionality, for a single backup is sufficient to capture the current state.

Two concurrent restorations can only originate from user requests. While there is nothing preventing the system from performing multiple restorations at once, the functionality is not allowed to simplify the concurrency protocol of other operations. Our opinion is that restorations are rare, simultaneous restorations of the same machine even more so, hence we do not have to sacrifice any relevant functionality.

Two concurrent consolidations can originate from multiple sources. There is no advantage to simultaneous consolidations, for a single one can do the job of the bunch.

Simultaneous restoration and backup or consolidation can only arise from an ill-timed user request. While there is nothing fundamental preventing such an operation, the aforementioned technical limitation would make the implementation too complex to justify such a rare occurrence.

Finally, concurrent backup and consolidation are fairly rare, for they are usually synchronized to perform one after the other.

To reiterate, the only impactful sacrifice we make is disallowing restorations concurrent with other operations. This limitation can be mitigated at the cost of a more complex setup and a similarly complex implementation, see Section 3.17.1.

2.8 Backup trees

Backup trees, can be produced under several circumstances:

- Concurrent backups.
- Concurrent backup and consolidation.
- If the changed blocks computation searches for the most suitable candidate during a delta backup.

All listed cases cannot arise in our application as established by the previous sections. Consequently, the existence of delta backup trees cannot emerge in the system and the trees are, therefore, disallowed.

2.9 Restoration method

We have mentioned there are two possible methods of backup restoration:

- Restore into a snapshot.
- Restore into a new virtual machine.

Let's discuss the former. The method is extremely error-prone because the configuration of the machine could have changed since a backup was made and the only mitigation is to compare the complex configuration, and possibly try to recover from any inconsistencies. Such recovery can prove troublesome when the number or size of disks change. Furthermore, the approach provides absolutely no advantage, for the machine still has to be powered off when the snapshot is reverted because system memory backup is not supported.

The latter method allows us to simply use the saved configuration to create a copy of the backed up machine, which is guaranteed to accept the saved data without any additional processing.

From this point onward, we only consider the latter option for its simplicity while having no disadvantages.

3. System design

This chapter comprises the design of the system, its rationale and ramifications, description of used technologies, and any known defects of the design.

3.1 Architecture overview

We illustrate the overall design of the system in Figure 3.1. The appliance follows a client-server architecture. We provide an implementation of the client called `vmbackupctl`, however, any API conforming implementation can be used in its place. The client communicates with the daemonized server called `vmbackupd` through a UNIX domain socket. The daemon accesses a database over a connection dependent on the database. Moreover, the server communicates with any configured hypervisors through their control suites in a platform-dependent process.

Each of the aforementioned components is further analyzed in the following sections.

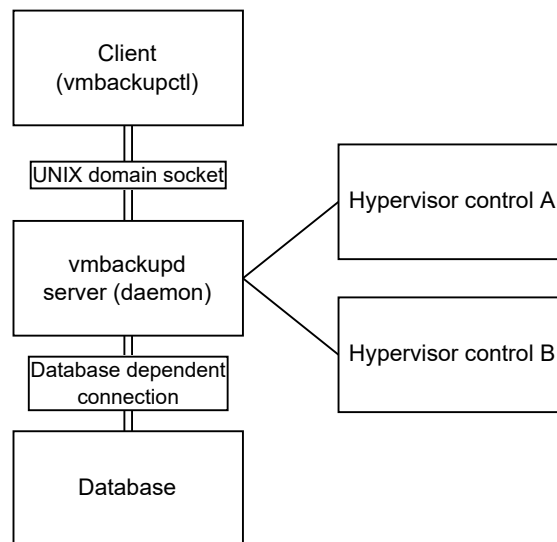


Figure 3.1: The figure illustrates the overall design of the system. It comprises a client (usually `vmbackupctl` contributed by us), the `vmbackupd` daemon, a database for data persistence, and communication channels to any hypervisor control suites.

3.2 Technology stack

While designing the system, we were faced with several decisions on what technology and approaches to use. We provide an overview of the landscape in this section. We immediately establish that our solution targets the GNU/Linux operating system. While it is possible to provide an implementation for Windows, the decision not to was made during the prototyping stage, to limit the number of platforms we have to implement compilation for, and test.

3.2.1 vCenter communication

In this subsection, we discuss communication processes of the vCenter server, see Figure 3.2 for highlight in the overall architecture.

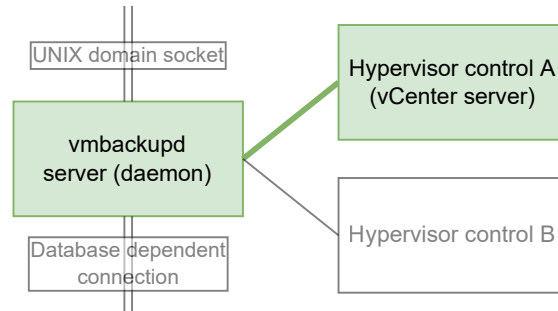


Figure 3.2: The daemon can communicate with the vCenter server, which is one of the supported hypervisor control suites, through a plethora of channels: a SOAP API, a REST API and an unspecified proprietary API.

The vCenter server management suite exposes two APIs:

- “vSphere Automation API”¹ is a REST API that is intended to simplify communication with the suite, as the following SOAP API is considered tedious to use.
- “vSphere Web Services API”² is a SOAP API that is designed to be used by the webpage admin panel shipped with the suite.

The former can be used either directly or through one of the two SDKs:

- “vSphere Automation SDK for Java”³ which is a Java SDK.
- “vSphere Automation SDK for Python”⁴ which is the same SDK only targeting Python.

The API is limited, as it does not provide certain integral functionality, for example creation of snapshots, and thus is not suitable to our use case.

The SOAP API, on the other hand, is very extensive, covering all our needs. Just like the REST API, it can be used either directly or through one of the following SDKs:

- “vSphere Management SDK”⁵ is a Java and .NET targeting SDK.
- goVmomi⁶ is a Go SDK.
- pyVmomi⁷ is a Python SDK.

The pyVmomi SDK is accompanied by a community examples repository⁸.

¹<https://developer.vmware.com/apis/vsphere-automation/latest/>

²<https://developer.vmware.com/apis/1192/vsphere>

³<https://developer.vmware.com/web/sdk/7.0/vsphere-automation-java>

⁴<https://developer.vmware.com/web/sdk/7.0/vsphere-automation-python>

⁵<https://developer.vmware.com/web/sdk/7.0/vsphere-management>

⁶<https://github.com/vmware/govmomi>

⁷<https://github.com/vmware/pyvmomi>

⁸<https://github.com/vmware/pyvmomi-community-samples>

3.2.2 Accessing disk data

None of the APIs mentioned in the previous subsection actually provide access to the data of virtual machine disks. The disk access functionality is implemented by the “Virtual Disk Development Kit” (VDDK) toolkit already expanded upon in Section 1.2. The toolkit is a collection of C libraries targeting usage from C and C++. Of particular interest to us is the component `VixDiskLib` that supplies disk management and access functions.

3.2.3 Language choice

The presented libraries shape our decision options when it comes to the programming language the application is developed in. We have to use the VDDK toolkit, for there is no documented API to access the disks. Moreover, we intend to have all the functionality in the same application to simplify the overall design, limit dependencies and prevent excessive data transfer between components. Therefore, developing two or more components that send the data to each other using inter-process communication is out of question. Furthermore, we would like to utilize one of the aforementioned API SDKs instead of consuming the raw API, as the API is nontrivial, which eliminates the possibility to implement the whole system in C or C++.

To reiterate our options, we can choose from Java, C#, Go and Python. We are faced with the problem of using C calling convention libraries (we will just call them C libraries for brevity, even though the term is not accurate) from one of them. Each of the aforementioned languages supports generation of bindings for C libraries.

The Go language is the least apt at generation of the bindings, requiring considerable manual boilerplate.

The C# language and its .NET runtime is relatively new to the GNU/Linux operating system, and the corresponding SDK does not target any version supported by the system.

The Python language is designed to be extended by C libraries. There are many libraries and frameworks facilitating generation of Python bindings to not only C and C++ libraries, but also C# or Java code. Moreover, Python has a rich ecosystem of third-party packages that might come in useful during the design and throughout the service of the application. Finally, Python has a large user base and most software providers choose to release a Python library interacting with their product. On the other hand, the CPython runtime has considerable technical limitations, for example its general low performance or the Global Interpreter Lock (GIL) preventing parallel execution of threads.

The Java language supports the bindings through the JNI mechanism and its runtime, the JVM, does not suffer from the same limitations as the CPython runtime. However, it is the authors’ least favorite, as its ecosystem is overly complex and its design enforces a certain unfavorable coding style.

Considering the preceding, we choose to implement the system in the Python language and circumvent any of its technical limitations if they prove to be pivotal. Consequently, we use the `pyVmomi` library to communicate with the vCenter server.

3.2.4 Additional libraries

This subsection provides a brief overview of additional libraries utilized by the system.

pybind11

“pybind11 is a lightweight header-only library that exposes C++ types in Python and vice versa, mainly to create Python bindings of existing C++ code. Its goals and syntax are similar to the excellent Boost.Python library by David Abrahams: to minimize boilerplate code in traditional extension modules by inferring type information using compile-time introspection.” [6]

The library targets C++ which proves extremely useful, for the `VixDiskLib` library cannot be directly bound into Python because of certain technical limitations, see Section 4.5.8 for details.

SQLAlchemy

“SQLAlchemy is a database toolkit and object-relational mapping (ORM) system for the Python programming language.” [7]

An ORM system abstracts table relations in a database with object hierarchies. Database tables correspond to classes, database rows to instances of the classes, and object references to table joins. However, there are certain ORM constructs not commonly available in relational databases, the most prominent being polymorphic types. When using a polymorphic type, a new table is created for each specialization plus a table for the generic type. When a reference to an instance of a polymorphic type is accessed, the generic table is used to determine which table should be joined depending on the type of the polymorphic instance.

We use the toolkit to completely abstract any database interactions.

NumPy

“NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.” [8]

While we have little use for most of the listed functionality, the library works well with the `pybind11` library. In particular, it facilitates transfer of binary data between the bound library and the CPython runtime without any copying.

3.3 Data persistence

Transactional behavior of data persistence is integral to our design. A well established technology guaranteeing the behavior is the relational database.

There are many relational database engines available of varying complexity of use and performance. Our system is not expected to access the database frequently or modify its state significantly. Therefore, we are free to use SQLite

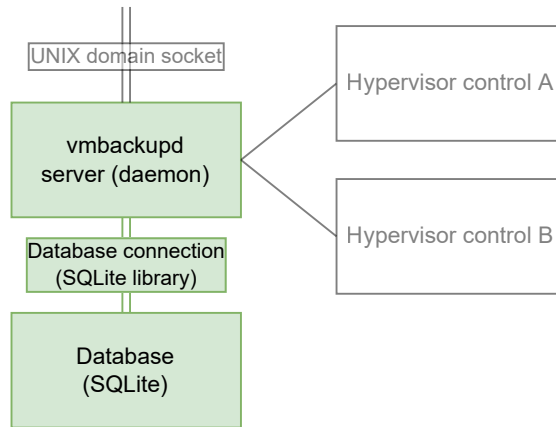


Figure 3.3: The daemon uses the SQLite database engine for data storage and its accompanying library for communication with the engine.

⁹: a simple, resilient albeit less performant —when compared to other industry staples— database system. This choice has its consequences, limiting concurrent restore operations. See Section 3.17.1 for details.

3.4 Daemon process architecture

This section comprises the design of the daemon process architecture. Refer to Figure 3.4 for an overview.

The daemon consists of several processes communicating with each other. There are three groups of processes: synchronous workers, asynchronous workers and management components. All of them are managed by a parent process, the “Daemon core”¹⁰. The core starts, controls, coordinates and terminates the other processes.

The synchronous workers comprise a tracked info provider and an inventory info provider. The asynchronous workers consist of a group of backup, restore, cleanup and consolidation workers. The management components are a state manager, a scheduler, and a user communicating process, where the state manager also partially acts as a synchronous worker.

3.4.1 Daemon core

The daemon core is responsible for bookkeeping of the whole appliance (once the main execution starts). The core terminates the other processes when a corresponding user request is received or when a system interrupt occurs. Moreover, all requests from any part of the application are routed through the core, which enables it to fully control concurrency of requested operations, see Section 3.5 for details.

The main operational responsibility of the core —apart from the aforesaid— is dispatching of client and scheduler requests to their targets. When handling

⁹<https://www.sqlite.org/index.html>

¹⁰We could not resist the temptation to reference the Demon core: https://en.wikipedia.org/wiki/Demon_core.

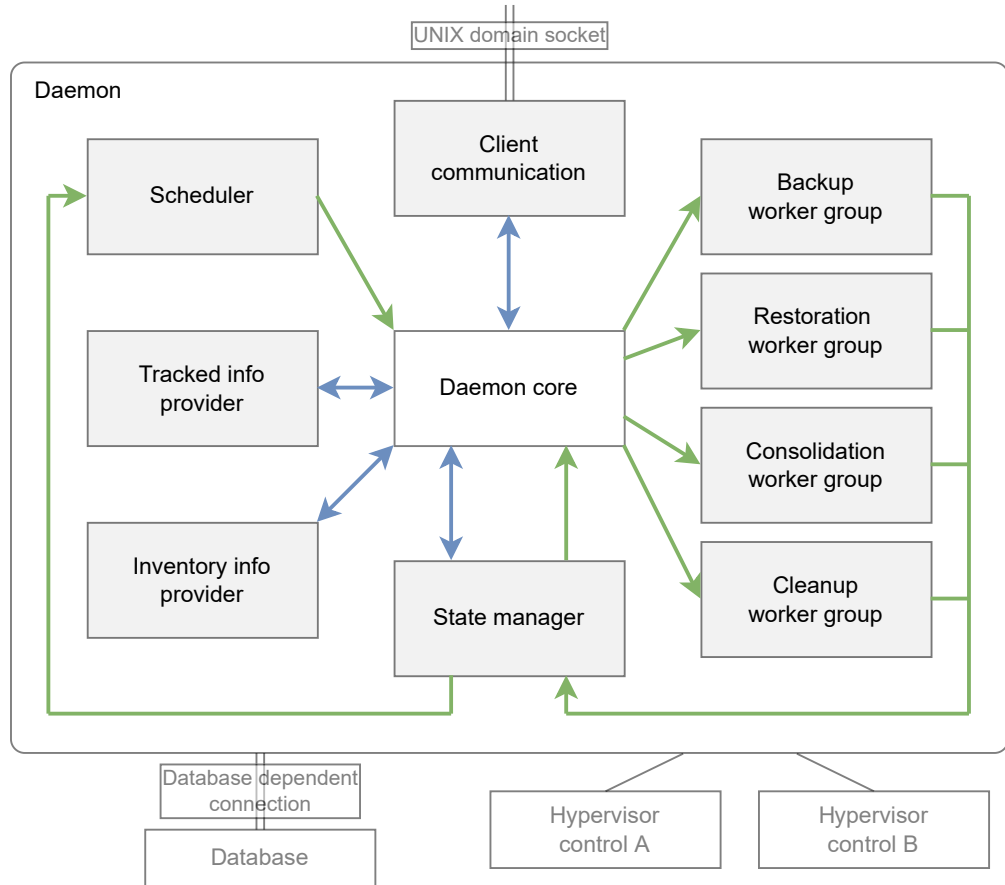


Figure 3.4: The overall architecture of the daemon. The light gray components are individual processes, while the white “Daemon core” component is the parent process. Arrows signify communication between components and its direction, while the blue and green colors of the arrows indicate synchronicity or asynchronicity of the traffic, respectively. The UNIX domain socket is only accessed by the “Client communication” component, while the hypervisor control suites and the database are accessed by most of the segments (the connections are not shown in the Figure to prevent clutter, as are not bookkeeping connections between the core and all other components).

a scheduler request, the request is simply dispatched with no response. On the other hand, when handling a client request, it is dispatched to the correct worker and a response is sent back synchronously.

3.4.2 Client communication

The client communication segment binds to a UNIX domain socket and waits for any client requests received over the socket. When a request arrives, it is sent to the daemon core for processing and a response is always awaited. The type of response depends on the request, see Section 3.11 for requests details.

3.4.3 Scheduler

The scheduler issues requests according to the configured policies of individual virtual machines tracked by the system. The scheduler can also be notified by the state manager (or possibly any other component) of certain events occurring in the system. Currently, the only event reported, is when a virtual machine backup or consolidation fails, however, other events can also be included. The scheduler uses the reported information to amend its schedule, for example, excluding a virtual machine from backups to prevent repeated failures.

3.4.4 State manager

The state manager monitors the state of the system and produces its summary. It notifies other processes when certain events arise in the system. Specially, it reports to the daemon core when a virtual machine blocking operation finishes, for more on concurrency control see Section 3.5. It also serves as a synchronous worker, able to respond to requests for state summary.

3.4.5 Tracked info provider

The tracked info provider is a synchronous worker providing information on the entities tracked by the system. Among the tracked entities are considered virtual machines, their backups and disks, hypervisor inventories and various other supplemental items used to perform backups and their restoration. The information provided is particularly useful during restorations.

3.4.6 Inventory info provider

The inventory info provider is a synchronous worker producing summaries on entities managed by all available hypervisor inventories, for example datacenters, virtual machines or data storage. The summaries are principally useful for configuration of the system and restoration operations.

3.4.7 Worker groups

There are four workers groups: backup, restoration, cleanup and consolidation group. Each group comprises a configured number of workers of the respective type. Any corresponding requests are submitted by the daemon core to the whole group, and the individual workers handle the requests concurrently. If there are more requests than idle workers, the requests are enqueued and processed when a worker becomes available. The workers report to the state manager when they finish processing a request.

3.5 Concurrency control

The system enforces that only at most one of backup, restoration, cleanup and consolidation operations is performed on the same virtual machine at any given time. The daemon core is ideally suited to implement the concurrency control, for every request has to pass through the core.

If any of the blocking operations are requested, the core checks whether the targeted virtual machine is free. The core locks the machine and issues the request if the machine was free. Otherwise, it enqueues the request, and the request is eventually handled when the virtual machine is unlocked (possibly after preceding enqueued requests are handled).

A virtual machine is unlocked when the state manager notifies the core that a blocking operation targeting the machine has finished. If there are any pending requests, the least recent one is issued (resulting in locking of the machine).

3.6 Requests

In this section, we list all requests serviceable by the system. The requests are the same regardless of their origin (there is an implementation detail affecting the client requests, see Section 4.5.9, nevertheless, all requests are the same when they reach the core). The daemon processes the following requests:

- A backup request is a request to back up a particular virtual machine.
- A restoration request is a request to restore a backup of a particular virtual machine into a new virtual machine created in a configured hypervisor inventory.
- A cleanup request is a request to clean up all backups of a particular virtual machine.
- A consolidation request is a request to consolidate the backups of a particular virtual machine according to its retention policy.
- A tracked summary request is a request to produce a summary of queried entities managed by the daemon.
- An inventory summary request is a request to produce a summary of queried entities managed by the available hypervisor inventories.
- A state summary request is a request to summarize the state of the system.
- A blocked summary request is a request to produce a summary of virtual machines that are currently locked, and requests targeting the machines that are blocked.
- A termination request is a request to terminate the appliance.

The requests can be divided into two groups in two distinctive ways. Firstly, by whether the requested operation needs exclusive access to the virtual machine or not, into blocking and non-blocking requests. Secondly, by whether the handler of the request is synchronous or not, into synchronous and asynchronous requests.

Blocking request are: backup, restoration, cleanup and consolidation. Non-blocking are: tracked, inventory and state summary, and termination.

Synchronous request are: tracked, inventory and state summary, and termination. Asynchronous are: backup, restoration, cleanup and consolidation.

While all asynchronous requests are currently blocking, there is nothing preventing the introduction of a non-blocking asynchronous request (although synchronous blocking requests are not supported).

For request processing, see Section 3.11. For technical details and request API, refer to the manual A.1.

3.7 Workers

Workers throughout the system correspond in their type to the request they are handling. To reiterate, there are synchronous and asynchronous workers, and blocking and non-blocking workers. All combinations apart from a synchronous blocking worker are allowed, albeit not used.

Submission of requests to any blocking worker (or a worker group) is protected with the concurrency control mechanism when the request is submitted through the daemon core, which all requests are.

3.8 Responses

The daemon can reply to a client with one of the following responses:

- A success response.
- A partial success response.
- A failure response.
- A critical failure response.
- No response at all.

The former four are further described in request processing in Section 3.11. The latter can only occur under conditions more critical than critical, catastrophic, if you will. These conditions are either sudden termination of the daemon caused by the environment or a bug in a well audited piece of code (in this case the failure is still reported but not to the client).

3.9 Jobs

The asynchronous requests need to be tracked by the system while they are being processed. To that end, they are associated with an identifiable job that also carries their status, before they are submitted to the workers or worker groups. The jobs are used by the state manager to perform its functionality of tracking the system state.

A job can be in multiple states:

- A pending job is a job submitted to a worker group and waiting to be started.
- An in progress job is a job that is being processed by a worker.

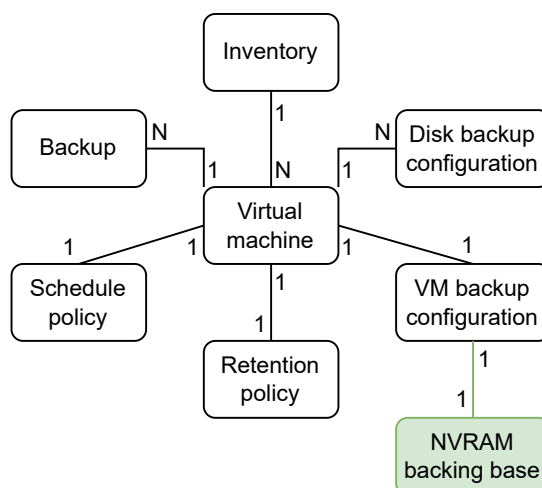


Figure 3.5: The segment of the database model capturing inventory and virtual machine relationships. See Sections 3.10.1, 3.10.2 and 3.10.6 for details.

- A blocked job is a job that cannot currently proceed because the targeted virtual machine is locked.
- A finished job is a job that has finished successfully.
- A failed job is a job that failed for any reason.

3.10 Data model

This section provides an overview of the database model used in the appliance. The model is built using the `SQLAlchemy` ORM system, consequently, there is no traditional model, instead we describe the relationships of entities as modeled by the ORM.

The following description of the model only includes the relationships between individual entities. All fields are excluded. Any entities mentioned in a relationship but not expanded upon are leaf entities with no additional relationships.

We do not include an entity-relationship diagram of the whole model, for it is too unwieldy, see partial (not entity-relationship) diagrams accompanying each subsection instead. The diagrams use the following notation. Individual components denote database entities. Lines with two ones denote one-to-one relationships. Lines with a one and an “N” denote one-to-many relationships. Dashed double lines denote recursive one-to-one relationships. Green components and their lines are specific to vSphere entities.

3.10.1 Inventory

An inventory

- manages several virtual machines.

See Figure 3.5.

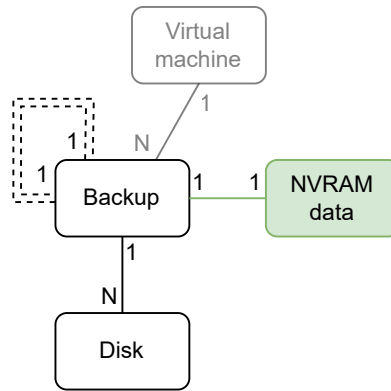


Figure 3.6: The portion of the model describing backups and their relationships. See Section 3.10.3.

3.10.2 Virtual machine

A virtual machine

- is associated with an inventory,
- has a polymorphic schedule policy,
- has a polymorphic retention policy,
- has multiple polymorphic (all the same type) platform-specific backups,
- has a polymorphic platform-specific virtual machine backup configuration, and
- has multiple disk backup configurations.

See Figure 3.5.

3.10.3 Backup

A backup is a polymorphic entity that

- is associated with a virtual machine,
- may be associated with another backup that acts as its predecessor,
- may be associated with another backup that acts as its successor, and
- has multiple polymorphic (all the same type) platform-specific disks.

The vSphere specialization additionally

- has polymorphic NVRAM data.

See Figure 3.6.

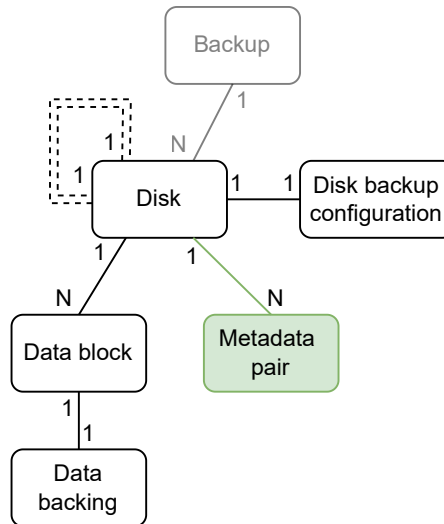


Figure 3.7: The segment of the model focusing on disks and their relationships. See Sections 3.10.4 and 3.10.5.

3.10.4 Disk

A disk is a polymorphic entity that

- is associated with a polymorphic backup,
- may be associated with another disk that acts as its predecessor,
- may be associated with another disk that acts as its successor,
- has a disk backup configuration, and
- has multiple data blocks.

The vSphere specialization additionally

- has multiple vSphere disk metadata pairs.

See Figure 3.7.

3.10.5 Data block

A data block

- is associated with a polymorphic disk, and
- has a polymorphic data backing.

See Figure 3.7.

3.10.6 Polymorphic virtual machine backup configuration

A generic virtual machine backup configuration

- is associated with a virtual machine.

The vSphere specialization additionally

- has a polymorphic NVRAM data backing base.

See Figure 3.5.

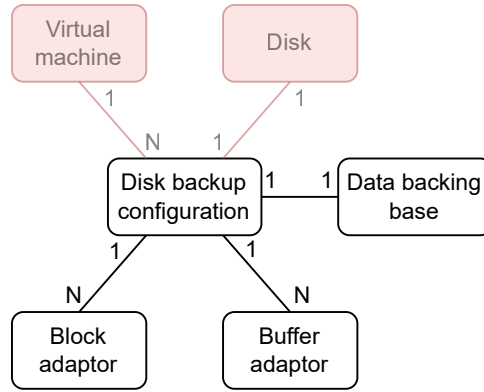


Figure 3.8: The section of the model capturing disk backup configurations and their relationships. See Section 3.10.7. The red segments and their lines denote an exclusive relationship. The disk backup configuration is in a relationship with either a virtual machine or a disk, but not both.

3.10.7 Disk backup configuration

A disk backup configuration

- is either associated with a virtual machine or a polymorphic disk, but not both,
- has multiple polymorphic block adaptors,
- has multiple polymorphic data adaptors, and
- has a polymorphic data backing base.

See Figure 3.8. The relationship of a disk backup configuration with a disk or a virtual machine is exclusive. The relationship is implemented using two many-to-many relationships because of the limitations imposed by the ORM system, nevertheless, the semantics of the relationships are as stated previously and in the diagram. See Section 4.1.3 for an explanation of the implementation.

3.11 Request processing

The discussion about request processing has already been initiated in the daemon core section 3.4.1. Let's quickly reiterate. Scheduler requests are simply dispatched to their destination and no response is produced. Client requests are also dispatched, but a response is sent back synchronously.

The contents of the response depend on the target of the request. When a synchronous process is targeted, the core waits for the response of the worker. On the contrary, when an asynchronous process is tasked, a response with a submission status is produced and the state of the requested operation can be queried using one of the synchronous requests.

Synchronous workers process the request as received, not interacting with the state manager. On the other hand, asynchronous workers receive the request associated with a job, subsequently, they use the job to report the status of the request.

The mechanism of asynchronous request handling is as follows. The core creates a job for an asynchronous request and notifies the state manager of the existence of the job. Subsequently, the job and its request are dispatched to a corresponding worker or worker group. The worker handling the job announces the fact to the state manager and begins to process the associated request. When the computation finishes, the worker notifies the state manager of its result. The job is considered finished by the concurrency control.

Each client request, be it synchronous or not, is replied to with a response. The content of a response depends on the request. The status of a response is determined by the success of the requested operation for synchronous requests, and by the success of submission of the request for asynchronous requests.

When a request is successfully handled or submitted, a success response is produced. When an asynchronous request is blocked because of a locked virtual machine, a partial success response is replied with. When the requested operation or submission fails, a failure response is returned. Finally, when the operation or submission fails fatally—in other words, there is a bug the system managed to handle and remain stable—a critical failure response is produced. It is suggested to terminate the daemon when a critical failure is reported, nevertheless, the system can continue to operate.

3.12 Virtualization stack integration

The appliance implements the algorithms described in Section 1.3 in a platform-agnostic manner. Any implemented platform-specific operations can be plugged in as needed, see Section 4.6.1. The system is especially designed to accommodate various virtualization stacks. While we tried to keep the design as generic as possible to anticipate any virtualization implementation, unforeseen complications can arise, as we made certain limiting assumptions based on our research of the virtualization landscape. We list the limitations in the following paragraphs.

We use UUIDs to identify virtual machines both in our system and in hypervisor inventories because all prominent virtualization technologies expose identification by UUIDs in some way, and the approach simplifies the design significantly.

We assume that the data of a virtual machine disk are organized in disk sectors and accessible in variable sized blocks of sectors.

3.13 Data modification

One of the goals of the system is support for disk data modification. Disk data transformers are supported on a per-disk basis, it is possible to configure custom transformers for each disk and a default configuration for the rest. The transformation modules can be chained. There is practically no limitation on the transformations performed by the modules, in particular, compression and encryption of the data can be easily implemented.

3.14 Logging

The system attempts to log any events relevant to an administrator. Of special notice should be warning and error messages. Error messages are emitted when a bug is encountered and serve as diagnostics. Warning messages are logged whenever some operation fails in a way anticipated and allowed by the system.

3.15 Client communication channel

The daemon communicates with its clients using a UNIX domain socket. The socket was chosen because the clients need to communicate with a perpetually running system.

The appliance does not support any authentication, authorization or traffic encryption, for that reason, we limit the vulnerability surface by supporting only local socket communication. However, the segment responsible for communication, can be easily replaced with any other implementation.

3.16 Configuration

The appliance itself is configured using a JSON file. Inventories that should be available to and virtual machines to be tracked by the system are also specified using JSON files. The JSON file format was chosen because it is relatively simple to edit by humans, while allowing non-trivial structures and providing facilities for automatic format validation.

The configuration of the system cannot be modified while the appliance is running. The system has to be restarted, to reconfigure an entity. This limitation is incurred by the implementation, or rather the lack thereof. Runtime configuration modifications can be easily implemented, see Section 4.6.12.

3.17 Defects

We provide an overview of known design defects and unfortunate technological choices in this section.

3.17.1 Database limitations

The choice of SQLite database engine limits concurrency opportunities throughout the application. While it does not pose any real limitations for backup, cleanup and consolidation, restoration could benefit from concurrency in certain situations, however rare they may be.

The limitation is imposed by the chosen database system, for it only supports a lock-based serializable isolation level which prevents any concurrent operations, not only those on the same virtual machine. Therefore, we are forced to use the only other isolation level it supports to allow concurrency at the virtual machine level. The imposed level is the read uncommitted level, which does not protect against any kind of concurrency. Consequently, we are forced to

implement custom concurrency control locking a virtual machine and with it the whole tree of database entities connected to that particular virtual machine.

It suffices to say that this approach is not ideal, nevertheless, the implementation is extremely lean and reliable, provided all database modifying operations are conducted from blocking workers which are protected by the mechanism.

The simplest mitigation of the whole problem is to replace the SQLite engine with an engine supporting the multiversion concurrency control method for serializable isolation level. The mitigation would eliminate the need for the custom concurrency control and enable concurrent restorations. Database engines supporting the aforementioned method are much more complex to set up compared to the SQLite engine, which is effectively just a file that can be copied if needed. The complexity of alternative engines, the relative simplicity of custom concurrency control, and the fact that concurrent restorations are rare, are the deciding factors to keep using SQLite and do the bookkeeping ourselves, at least in the pilot implementation.

4. Implementation

This chapter serves as a companion guide to the application code. We try to point out all points of interest in the implementation, demonstrate extensibility of the software, as well as, provide a tour of the whole. Please consult the code for any unclear passages.

We are going to refer to the manual distributed with the software throughout the chapter. There is an abbreviated version of the manual in the Attachment A.1, although, the complete version should be consulted for most of the referenced details. See the `README.md` file in the root of the code repository for the complete version.

4.1 Operation overview

This section comprises the general overview of the system. We outline the algorithm of the whole process and delve into individual components in the following sections.

The application operates roughly in the following steps once it is started:

1. Use the command line arguments to determine the location of the configuration files.
2. Read and validate the configuration from the files. First the daemon configuration is read, then the inventory configuration, and finally the virtual machine configuration.
3. Use the configuration to update the database with new entities and modifications of those already present.
4. Create and start all processes required by the application.
5. Let the processes perform their tasks until a termination request is received.
6. Terminate the processes and the application.

4.1.1 Command line arguments

The appliance is primarily configured using JSON files. The location of the files is, however, provided through a command line argument. The argument is the path to the configuration directory, which is the root of all configuration files. There are three more arguments that affect the behavior of the system, however, they should be only used for debugging. See the manual for their details.

4.1.2 Configuration files

The root configuration directory contains the `config.json` file with the daemon configuration, the `platforms` directory with inventory configurations, and the `vms` directory with virtual machine configurations. All files are validated using JSON Schema. We do not provide any examples here because our horizontal space is limited, please see the manual.

Daemon configuration

The daemon configuration file contains a single object with the following fields:

- `dbUrl` is a database URL of the database used by the system. The URL can have any value as long as it has the format: `sqlite+pysqlite:///path/to/database/location.sqlite`.
- `vixDiskLib_libDir` is the path to the root directory of the VDDK distribution. See the manual for VDDK installation.
- `vixDiskLib_configPath` is the path to the VDDK configuration file. The configuration file is described in the VDDK manual¹.
- `daemonSocketPath` is the path to the socket the daemon binds to and listens for requests on.
- `backupWorkerCount`, `restoreWorkerCount`, `cleanupWorkerCount` and `consolidationWorkerCount` specify the number of workers in respective worker groups.

Inventory and virtual machine configuration

Inventory and virtual machine configuration files are used to control individual inventories available to, and machines set to be tracked by the system, respectively. The files can have arbitrary names, and they can be in an additional directory or a directory tree. See the manual for the format of the files.

4.1.3 Database operation

The ORM supports a plethora of database engines through the configuration URL². See Section 4.6.3 for details on database engine migration.

The database is created by application of a database migration auto-generated from the database model. We use the tool `alembic`, which is part of the `SQLAlchemy` project, to generate and apply the migration.

SQLite does not support the UUID type (which is normally supported by engines like PostgreSQL). Therefore, we implement a custom type converting UUID objects to their 16 byte representation and back, and save the bytes instead of the object. See `model.py` for details.

Shared disk backup configuration objects

A “disk backup configuration” is an entity in the database, see Section 3.10. The “virtual machine” and “disk” entities are in an exclusive relationship with the configuration.

The ORM system does not provide any mechanism to establish such a relationship, which in fact is just a one-to-many relation thanks to the exclusivity. Therefore, the relationship is modeled as two many-to-many relations with one-to-many semantics. This is just an implementation detail to bypass the limitations

¹<https://developer.vmware.com/web/sdk/7.0/vddk>

²<https://docs.sqlalchemy.org/en/14/core/engines.html#database-urls>

of the ORM system. All additional operations (like cascading) have the semantics of a one-to-many relationship.

Cascading

All parent-child relationships are cascaded. Parent sides of the relationships use the words “manages” or “has” in the description of the model in Section 3.10. Child sides of the relationship use the compound “is associated with” in the same description. The only relationships that are not cascaded are recursive relations between backups, and disks.

Polymorphic database objects

In the summary of `SQLAlchemy`, we introduce the ability to model polymorphic entities. Polymorphic entities are constructed with inheritance (the only use of inheritance in the entire application, save for inheriting from `enum.Enum`) at the implementation level. Each polymorphic entity has to be tagged with an identifier of the type, so the entities use both subtyping and tagging.

4.1.4 Processes and communication fabric

The appliance is made up of processes and their communication fabric. The processes are described in Section 3.4. Each process is implemented using the `Process` construct from the `multiprocessing` package.

The communication fabric is implemented with a collection of `Pipes` and `Queues` from the same package. Refer to Figure 4.1 for an overview. The communication between the daemon core and scheduler, tracked info provider, inventory info provider, state manager, and client communication processes is facilitated by `Pipes`. The core sends administrative commands to all other processes through bookkeeping `Pipes`. Finally, submission of requests to individual worker groups, reports of job state, and notifications to the scheduler are all sent through `Queues`. Using `Queues` allows multiple provider multiple consumer architecture, which is used inside the worker groups, where multiple workers receive from the same `Queue` and report to a different one.

4.1.5 Termination

Each process is connected to the core with a bookkeeping pipe. When a termination request is received by the core, it issues a termination command to all other processes through the pipes. Moreover, there is a socket pair facilitating termination when the `SIGINT` or `SIGTERM` system interrupts are raised. Before the core is run, an interrupt handler is registered on the interrupts. The handler sends a single byte on its side of the socket, and the core issues a termination command, just like in the previous case, when it receives the byte on the other side.

After the core issues the termination command, it itself terminates and all other processes are joined. If the core terminates because of a bug, it still issues the termination command before it fails, to shut down the system gracefully.

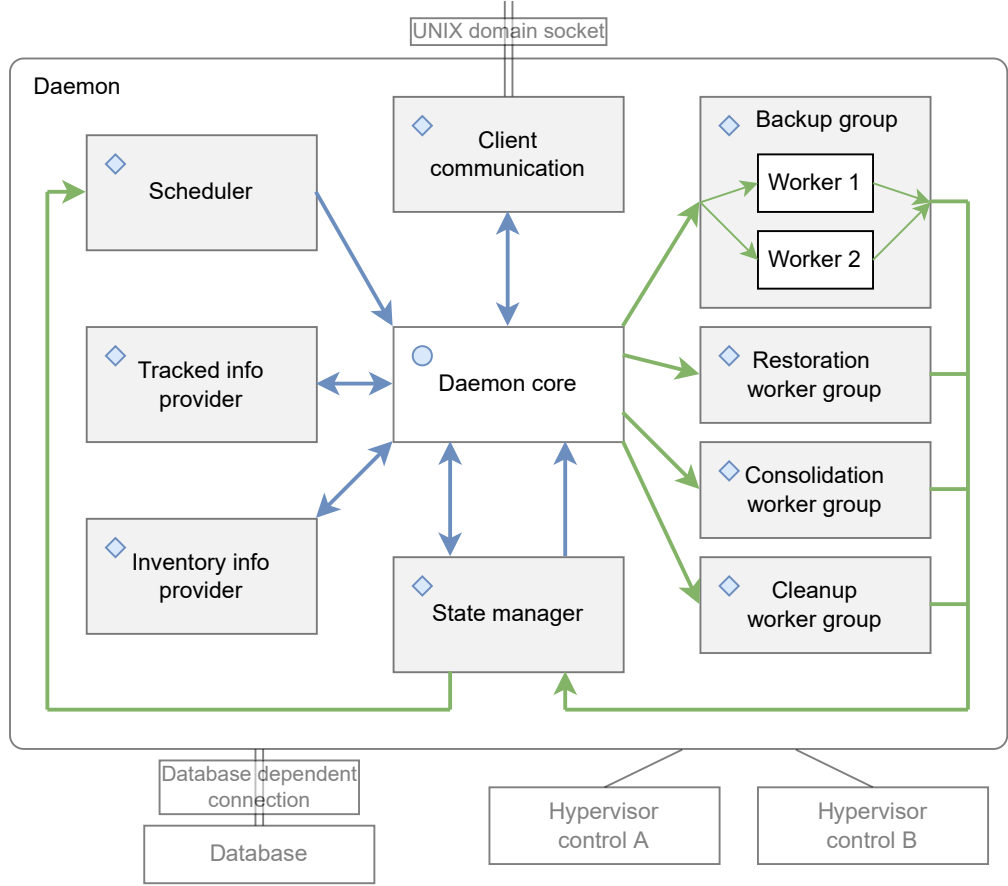


Figure 4.1: This is the same overall architecture of the daemon as shown in Figure 3.4. The arrows still signify communication between components and its direction, but the blue and green colors of the arrows indicate whether a **Pipe** or **Queue** is used, respectively. Moreover, the bookkeeping connections (omitted in the previous figure) are introduced as a blue circle and diamonds. The color blue signifies that they are **Pipes**, the circle is the sending multi-side (there is a sending side for each receiving side) and the diamonds are the receiving side. Additionally, the inside structure of the worker groups is shown (only for the “Backup group” but the other groups follow the same schema). The receiving and sending **Queues** used to communicate with a particular group are shared by the workers in the said group.

4.2 Main operation

This section comprises a description of the techniques facilitating the main functionality. This text would better fit after Section 4.1.4, but we would run out of text partitioning elements, so this text structure is a result of technical limitations. We apologize for the inconvenience.

The main operation stage of the system starts when all processes are run and the daemon core starts executing, and ends when one of the termination mechanisms is initiated.

In the following subsections, we outline the operation of each segment described in Section 3.4. The outline should provide a solid understanding of the system as a whole.

4.2.1 Client communication

The client communication segment binds to a UNIX domain socket and listens for client communication. When a connection is established, it

1. reads a request from the socket,
2. parses the request,
3. sends the request to the core,
4. awaits a result from the core,
5. serializes the response, and
6. sends the response to the client.

The communication between a client and the daemon is governed by a simple protocol. The parties exchange only two messages: a request and a response.

The request consists of a length of the payload and the payload. The length is a four-byte big-endian unsigned integer. The payload is a UTF-8 encoded JSON, whose format corresponds to one of the supported requests. See the manual for request API.

The daemon reads exactly the advertised length of bytes from the socket, any extra bytes are ignored, and if there are not enough, the connection timeouts. The request is decoded into a dictionary. If it fails, an error response is sent and the connection is closed.

Once the request is decoded, it is tagged with a UUID that identifies the request in communication with the core. See Section 4.5.9 for explanation. The tagged request is sent to the core and a response is awaited. Once the response arrives, it is encoded the same way the request was encoded (four-byte length, UTF-8 payload) and sent to the client. The connection is, subsequently, closed.

Connections to the daemon are serial, i.e., only one client can communicate with the daemon at the same time, although, the connection itself can be established concurrently.

4.2.2 Daemon core

The core listens on multiple channels: the client communication pipe, the scheduler pipe, the state manager pipe and the interrupt socket. Its operation depends on which pipe or socket is active.

When the interrupt socket is active, the core reads the single byte sent by the interrupt handler, issues the termination command, and terminates itself.

When the state manager pipe is active, the core receives the job state update sent by the state manager. If the job is finished and blocking, it unlocks the virtual machine that was blocked by the job, and if there are any enqueued jobs targeting the machine, it issues the least recent one for processing.

When the scheduler pipe is active, the core receives the request issued by the scheduler and processes it.

When the client communication pipe is active, the core receives the request issued by the client. It strips the request of the tag produced by the client

communicating segment and stores it for sending of a response. The core processes the request. When a response is produced, it is annotated with the stored tag and the response is sent.

The core itself handles the blocked summary request, for only the core keeps track of which virtual machines are locked.

4.2.3 Worker groups

The four worker groups process the issued jobs as detailed in Section 3.11. Individual workers compete for jobs with other workers in their respective group by polling the same queue. The workers reattempt to fulfil their request if the operation fails with a possibly recoverable error, for example a network timeout.

4.2.4 State manager

The segment updates its state with job status notifications received from workers. Furthermore, it serves the state summary requests received from the core. Finally, it notifies the scheduler and core with events relevant to the segments.

The core is informed with finished jobs, be it successfully finished or failed jobs, and the scheduler is notified with failed jobs associated with only serious errors. The failure to perform a backup or consolidation is considered serious, while failed restorations are not severe.

The severity of failures is arbitrary and depends on the use case of the system. For example, currently, only backups are scheduled, therefore, a failed backup or consolidation is relevant to the scheduler, while failed restorations do not affect the backup mechanism. There might be a situation when, although a backup has failed, it would be beneficial to reattempt it. The system should be adjusted to the use case.

4.2.5 Inventory and Tracked info providers

The inventory and tracked info providers handle inventory and tracked summary requests, respectively. The tracked info provider obtains all its data from the database, on the other hand, the inventory info provider relies almost exclusively on the data supplied by the hypervisors.

4.2.6 Scheduler

The scheduler process issues operations configured for each virtual machine. Currently, only the backup operation is implemented.

The scheduler uses a heap to implement a priority queue. The queue is capable of removing entries and changing their priority, which enables implementation of runtime schedule configuration changes without the need to rebuild the heap. See Section 4.6.12 on how to implement runtime configuration changes.

Notifications can be sent to the scheduler to inform it of any relevant events in the system. For example, when a backup fails, the scheduler excludes the virtual machine from its schedule. The functionality can be adjusted and extended to support more events, for example the aforementioned schedule configuration adjustment, see Section 4.6.4.

The scheduler has a limitation of not allowing operation frequencies lower than once every two thousand years. This peculiar limitation is introduced by generalization of the case when no backup of a virtual machine exists yet. We simply pretend a backup was created on the first day of the first month of the first year. Using this generalization results in clean code.

4.3 Backup algorithms

This section provides an overview of the algorithms used in the worker groups. These algorithms are the most technical part of the implementation, and offer a lot of space for improvements and platform specializations.

The behavior of all following algorithms can be modified through a mixin. In fact, all platform-specific behavior is supplied through the mixin. See Section 4.6.1.

4.3.1 Backup

The backup algorithm is divided into multiple stages, modeling fragmentation of the virtual machine architecture. The stages are

1. backup of virtual machine level information, for example its configuration or NVRAM data in the case of a vSphere machine,
2. collective backup of disks of the machine,
3. individual backup of disk level information, for example metadata,
4. collective backup of disk sector blocks, and
5. individual backup of disk sector blocks.

The stages are discussed in the following sections.

Virtual machine level backup

The inventory managing the targeted virtual machine is loaded from the database and used to obtain a platform-specific abstraction of the machine.

A UUID that will be used to refer to the backup is created. A platform-dependent task to create a snapshot is issued through the machine abstraction. The most recent backup of the machine is selected as the parent backup, if there is one.

A platform-specific database object backing the backup is created, and generic data like the UUID or parent backup is stored in the object. Platform-specific backup configuration is loaded from the database (discussed in the model section).

Any platform-specific data that can be saved before the snapshot is created is saved into the database object. We refer to this as pre-snapshot saving. The creation of the snapshot is awaited. The platform-specific snapshot abstraction is used to save any platform-specific data that could not be saved beforehand. We call this post-snapshot saving.

The vSphere implementation saves the virtual machine configuration and NVRAM data. The NVRAM data is saved before the snapshot is finished because it seems to be static. Configuration is saved after the snapshot is created, for the snapshot guarantees that the configuration is immutable.

The next stage of the algorithm is executed. If at any point the backup fails, the snapshot and any backed up data of this stage is deleted. The snapshot is removed when all following stages are completed.

Collective backup of disks

The disks are backed up serially, although the initial intention was to allow choosing of an implementation (possibly parallel) with the mixin. The principal reason is that parallelization at the disk level has to be implemented using processes, for the Python interpreter does not support parallel threads.

The use of processes on their own would not pose much of a problem if the libraries used to communicate with the hypervisor did not have to maintain a network connection (please note that we need the connection before we get to this stage, and cannot, therefore, establish them at this point). This reason and the lesser parallelization balancing options on the disk level led us to the decision not to allow parallel backup of disks to simplify the design. Instead, we intend to provide parallelization options at the disk sector level, which provides better balancing options, see Section 4.6.9 for details.

The snapshot abstraction is used to iterate over available disks, producing disk abstractions. Each disk is backed up serially. The corresponding disk backup configuration is loaded from the database (just like in the virtual machine backup configuration in the previous section), and the configuration is used in the next stage of the algorithm. The individual disks are identified using UUIDs provided by the hypervisor.

If the backup fails at any point, any already backed up data of this stage is deleted.

Individual disk backup

First, the disk corresponding to the disk that is being backed up (i.e., a predecessor disk) is looked up in the previous backup. The predecessor disk is tested whether it can be used for delta backups. The verification is platform-dependent. If the disk can be used for delta backups with the current disk, it becomes its parent disk. The existence of a parent disk enforces a delta backup.

A platform-specific database object backing the disk is created, and generic data is stored in the object. For example, the parent disk or the current disk UUID. Among the generic data are certain objects, that have not been discussed yet. It is the data backing base, and block and buffer adaptors. We will discuss them in the following sections, as they are relevant to the data of the disks.

Any platform-specific disk data is saved into the database object. The vSphere implementation saves the disk metadata, which are key-value pairs, and an ID used to compute which blocks have changed since a certain date.

The sector blocks of the disk that should be backed up are determined in a platform-specific way. The implementation has to consider whether a delta backup is going to be performed or not (which is established by the existence of

a parent disk, and not up to the block computation mechanism to decide). If a delta backup is to be performed, the implementation should return blocks that have changed since the previous backup. Otherwise, it should produce all blocks that are allocated on the disk.

The computed blocks are adapted using the block adaptors. Block adaptors can modify the size of the block, split or even duplicate it, the only limitation being that it cannot extend the blocks beyond their initial size. See Section 4.6.5 on how to implement and use custom adaptors. There is one mandatory block adaptor that limits the size of the blocks to a user-defined value. The principal purpose of the adaptor is to limit the amount of data loaded into memory at once.

The blocks are backed up in the next stage of the algorithm. If at any point the backup fails, any data saved in this stage is deleted.

Collective sector blocks backup

Unlike in the “collective disks backup” stage, the algorithm in this stage is platform-defined. The only available implementation is serial, nevertheless, a parallel approach is possible. The primary reason it is not implemented is the lack of suitable environment for testing and highly heterogeneous data transfer options limiting the contribution of a single implementation. See Section 4.6.9 for a proposed solution and observations of the problem.

The data backing base is used to produce a data saver targeting the disk the blocks belong to. The data backing base should contain generic configuration of the data saving mechanism and specialize the configuration for each disk, although, the specialization is not mandatory. For example, the base targeting the local file system is configured with the root directory the data should be saved into, and it produces disk specialized savers that create a hierarchy in the root directory for their data. The local file system targeting base and its corresponding saver are the only implemented, see Section 4.6.7 on how to contribute an additional base.

The blocks are backed up serially. If at any point the backup fails, any blocks already backed up are deleted.

Individual sector block backup

In the final stage of the algorithm, the sector block is read from the disk abstraction. The data corresponding to the block are transformed using the buffer adaptors. A buffer adaptor can transform the data in any way. For example, it can compress or encrypt the data. The only limitation is that it has to be able to reverse the transformation. See Section 4.6.5 on how to implement additional adaptors.

Finally, the data is saved by the saver produced in the previous stage and a database backing object is created and filled with information needed to read the saved data.

4.3.2 Restoration

Just like the backup, the restoration algorithm is segmented into stages, modeling the fragmentation of the virtual machine architecture. The stages mostly mirror the backup algorithm:

1. preparation of new virtual machine destination,
2. restoration of virtual machine level information,
3. collective restoration of disks of the machine,
4. individual restoration of disk level information, for example metadata,
5. collective restoration of disk sector blocks, and
6. individual restoration of disk sector blocks.

Again, we discuss the stages in their respective sections.

New virtual machine destination preparation

The request used to initiate the restoration is used to build a platform-specific virtual machine builder, which holds the desired destination of the new virtual machine. See the manual on vSphere destination specification and examples. It is not provided here, because the formatting does not allow it.

The builder is passed to the next stage.

Virtual machine level restoration

Any platform-specific data backed up during post-snapshot saving is restored into the virtual machine builder from the previous stage. Just to reiterate, the vSphere implementation saves the configuration of the virtual machine, which is crucial to its creation.

The builder is used to create a new virtual machine.

Any platform-dependent data backed up during pre-snapshot saving is restored into the new virtual machine. The vSphere implementation saves the NVRAM data.

The algorithm continues with the next stage. If at any point the restoration fails, the created virtual machine is deleted along with any restored data.

Collective disks restoration

The disks are restored serially, and the same justification and comments apply as with the backup, see Section 4.3.1.

Each disk of the new virtual machine is paired up with a backed up disk using their UUIDs.

Individual disk restoration

A disk chain of the backed up disk is constructed. The disk chain allows iteration over the disk's parent-child relationship. If the disk has a parent, the relationship is traversed and the iteration continues recursively from the parent (i.e., the grandparent is accessed and so on). If the disk does not have a parent, the iteration stops with the disk. The parent-child relationship arises when a delta backup is performed. Please note, that a disk chain of a disk with no parent is just the disk itself. Therefore, we do not need to consider a separate case for when a normal backup was performed.

An iterator of the blocks and associated data covered by the chain is created. The iterator returns all blocks in the chain with the most recent version of the data, resulting in the most recent image of the disk. See Section 4.5.6 for details.

The iterator is passed to the next stage. When the following stages finish, any platform-specific data saved during the backup is restored. The vSphere implementation only restores the key-value metadata pairs, as the saved ID is useful for delta backups and not restores.

Collective and Individual sector blocks restoration

The sector blocks are restored in a platform-dependent way. The only currently provided implementation is serial. The justification and details are the same as for backup, see Section 4.3.1.

Each block is produced by the iterator from the previous stage. The data corresponding to the block is loaded through a proxy and written to the new disk.

The proxy loading the data serves to reverse the transformation applied by buffer adaptors during the backup. The data cannot be simply processed in reverse with the adaptors because of the mechanism the iterator uses to iterate over blocks and data of the disk chain. The mechanism can create blocks that are subsets of their original blocks, see Section 4.5.6, and thus the data associated with the original blocks cannot be, in certain cases, directly used by the subsets.

The limiting factor is the nature of the transformations applied by the buffer adaptors. There are two types of transformations: “trivial” and “non-trivial”. The data transformed with a trivial operation can be reversed whole or in subsets with no change in semantics. For example, when each byte of the data is transformed in a XOR operation, the operation can be reversed on a subset with no difficulties. On the other hand, a compressed block of data has to be decompressed whole and only afterwards a subset can be taken. Compression is, therefore, a non-trivial operation. The loader proxy hides this detail.

4.3.3 Cleanup

The division of the cleanup algorithm is analogical to the backup and restore algorithms. The algorithm is much simpler than the previous two.

The data of each backup of the targeted virtual machine is deleted, and the associated database backing object along with all its children is also deleted. Any platform-specific pre- and post-snapshot savers are tasked to clean up the data they saved. Then, individual disks are processed serially. For each disk, its

platform-specific data is cleaned up. Finally, the disk data saved in a configuration dependent way (for example, in the local file system) is deleted as well.

There is a single limitation to the cleanup mechanisms. The operation must guarantee that it never fails. The principal reason is that it is invoked during backup and consolidation failure handling to clean up any artifacts, consequently, it has to be extra careful, for some constructs it may expect to exist when cleaning up a successful backup might not exist.

4.3.4 Consolidation

The consolidation is performed only on the disks of the backups, consequently, the extra data saved during pre-, post-snapshot and extra disk saving is ignored and the most recent version is preserved.

The consolidation algorithm has different stages to the previous algorithms. The stages are:

1. resolve which backups are affected by the consolidation according to the retention policy,
2. choose the most suitable consolidation algorithm specialization,
3. execute the chosen algorithm, and
4. delete the consolidated backups.

The stages are reviewed in their respective sections.

Consolidation targets and the most suitable algorithm

The retention policy of the virtual machine is used to determine which backups should be consolidated. The policy returns the backups which should be deleted and the least recent backup they should be consolidated into.

All platform-dependent available consolidation algorithms are asked in order whether they are capable of consolidating the targets. The first algorithm that accepts the targets is used. If none of the algorithms is capable of the task, a default generic algorithm is used. Currently, only the generic algorithm is implemented, please see Section 4.6.8 on how to contribute another algorithms. Considering there can be a vast range of algorithms, the following sections describe the generic algorithm and most probably do not apply to a newly contributed one.

Generic consolidation

The generic algorithm can accept backups created with any configuration. In fact, it is so generic it can be used to migrate the backed up data or introduce compression or encryption of the data with no additional steps.

The algorithm does not operate on the backups that should be consolidated, instead it focuses on the backup they should be consolidated into. For each disk of the backup, it checks whether it has a parent disk and thus is part of a backup chain. If the disk does not have a parent, it is ignored, for there is nothing to do, the disk already contains the whole image of the disk.

The data of the parent disk's disk chain is loaded just like in the restoration algorithm (including the loader proxy) and the blocks and associated data is stored along the data of the disk just like in the backup algorithm (with application of block and buffer adaptors and backing base). The transfer of the data is done per block, just like in both of the aforementioned algorithms (i.e., the data is not all loaded into memory at once and then saved). The transfer is sequential, nevertheless, a parallel version can be easily implemented, as there is no limitation by network connection like in the case of backup and restoration. Finally, a simple filtering has to be applied to the blocks of the parent disk chain. Any blocks already covered by the target (not parent) disk should be ignored because the disk already has the most recent version of the data.

In the last stage of the algorithm, the backups that have just been consolidated are cleaned up using the cleanup algorithm. If the consolidation fails at any point, all changes are reverted.

Needless to say, such implementation is not very effective. For example, when the backup configurations of the disks are the same (same block and buffer adaptors, and backing base) there is no need to apply the adaptors or load the data into memory at all. This is where the specializations come in. The specialization for local file system can skip the steps and simply move the files in the file system with very little extra processing (it still needs to compute which blocks are already covered by the disk and which have to be supplemented from the chain, and possibly split the files). Nevertheless, the generic algorithm can gracefully handle any configuration change.

4.4 Requests and Responses

Types of requests and responses are described in Sections 3.6 and 3.8. All requests and responses are in the JSON format and are validated using JSON Schema just like the configuration. Please see the manual for request API, as it is too unwieldy for our page width.

Responses have the following format:

```
{
  "status": "<STATUS>",
  "content": "<CONTENT>"
}
```

Where

- **status** is the status of the requested operation, and
- **content** is the request-dependent content of the response, please see the manual for the API.

4.5 Implementation issues

This section highlights any principal implementation details. The majority of the discussed issues are arduous solutions to vSphere related problems. The list is not

exhaustive, so if there are any unclear passages, consult the source code, where every detail is documented. Many of the presented (and not presented) solutions required a lot of research, taking up a considerable portion of development time.

4.5.1 vSphere changed blocks tracking

When the backup algorithm performs a delta backup, it has to compute which blocks have changed since the last backup. The task is heavily simplified, for the vSphere stack offers the functionality of tracking changes to disk blocks in time. The feature is called “changed block tracking” and is supported since the version 4.0 of ESXi. The functionality has certain limitations. For example, when a machine is restored to a snapshot, the platform cannot track the changes and the system performs a full backup. Moreover, RDM disks in physical compatibility mode are not supported. Please refer to the VDDK manual[5], which provides many details of the mechanism.

4.5.2 vSphere transfer modes

The `VixDiskLib` toolkit supports several data transfer modes. The only mode tested and supported is the “NBD”, which has the lowest backup performance, but solid restore performance. The other modes are “SAN” and “HotAdd”. The “NBD” mode transports all data over the LAN, while the other two bypass the network. Our environment did not allow us to test the other modes, nevertheless, the implementation should be mostly unaffected by the introduction of their support. We strongly suggest reading the VDDK manual[5] which details limitations imposed by the other modes. For example, the “SAN” mode is slower for restores than the “NBD” mode and dictates restoration into a snapshot.

4.5.3 vSphere NVRAM operations

The VDDK manual establishes that the NVRAM data of a virtual machine should be saved during the backup. However, there is no API supporting the operations. The only available approach is to download the corresponding file from the datastore the machine is saved in. The vCenter server supports downloads and uploads of files from the datastore. The mechanism is detailed in the manual which accompanies the Java “vSphere Management SDK” discussed in Section 3.2.1. However, the explanation skips over the method of request authentication. This is where the `pyVmomi` community samples (also mentioned in Section 3.2.1) prove invaluable. The `pyVmomi` library uses a cookie to authenticate the communication, and we simply use the cookie to authenticate our own requests. Considering the authentication method is not documented anywhere (that we know of) the user should vary of possible breakage in future versions.

This, however, is not the only problem plaguing the NVRAM data backup. The location of the file storing the data is easy to obtain during the backup through the `pyVmomi` library. The library can list any files associated with the machine. A problem arises when creating a new virtual machine during a restore operation. The NVRAM file is not created with the machine, but instead when the machine is powered on. However, the machine cannot be powered on before

the restoration is finished. Moreover, the library can only provide information on files that already exist. Consequently, we are forced to reverse engineer the location of the file. We observed that the NVRAM file is always stored alongside the file containing the configuration of the machine (the `.vmx` file). Therefore, we obtain the path of the configuration and store the NVRAM file alongside it. Beware, this mechanism does not rely on any documented behavior, and can, therefore, break at any point. We consider this the lowest point of the implementation.

4.5.4 vSphere virtual machine configuration

The configuration of a virtual machine is accessible through the `pyVmomi` library as a complex graph of objects that is incapable of self serialization and deserialization. We, however, have to store the configuration to facilitate creation of new machines during restore operations.

The object graph corresponds to an instance of `VirtualMachineConfigInfo` in the API reference³. It is infeasible to model the configuration with database constructs, so we implement a custom serialization mechanism that produces a JSON, and we store the JSON as text in the database. During the restore operation, we deserialize the JSON into the original object graph and use it to produce a configuration (which itself is an instance of `VirtualMachineConfigSpec` in the same API reference, an extremely similar object) for the new virtual machine.

The enormous scope of the objects forces the operation to be automatized. To that end, we use the following metaprogramming features of the Python language: `getattr`, `setattr`, `type` and `dir`. See the `vsphere/algorithms.py` module for serialization and deserialization, and the `vsphere/utils.py` module for transformation of the “info” object into the “spec” object. Both modules contain massive dictionaries and lists of field names that should and should not be serialized or copied into the “spec” object. All of these were compiled manually by inspecting the API and guessing which fields to keep and which not. We suggest that whoever is familiar with the matter should inspect the arrangement, for we suspect there are options which can break the system if the configuration is complex.

Each field of a configuration object is serialized by the following algorithm.

- If the field is of a trivial type (for example, string or integer) the value is stored as is, or
- if the field contains binary data, the data is serialized into a base64 encoded string, or
- if the field is a list, the algorithm is applied to each member of the list, otherwise
- the field is another object and is serialized into a JSON object with two fields, a `type` field containing the name of the object type, and the `fields` field containing a mapping of field names to their values.

³<https://developer.vmware.com/apis/1192/vsphere>

An example of the serialization applied to an instance of `VirtualMachineBootOptions` can be seen in the following listing:

```
{
  "type": "vim.vm.BootOptions",
  "fields": {
    "bootDelay": 0,
    "bootOrder": [],
    "bootRetryDelay": 10000,
    "bootRetryEnabled": False,
    "efiSecureBootEnabled": False,
    "enterBIOSSetup": False,
    "networkBootProtocol": "ipv4",
  },
}
```

4.5.5 vCenter communication

Each of the libraries used to communicate with the vCenter server has to maintain its respective network connection with the server. The connections share most of their arguments. The VDDK connections should be limited to a minimum, for the VDDK manual establishes that they consume significant resources. We provide this section as a head up that there are actually two connections used at the same time.

4.5.6 Disk chain iterator

The disk chain iterator can iterate over all blocks contained in the disk chain and pair it with the most recent version of corresponding data. The iterator first returns blocks of the disk belonging to the most recent backup, then it returns blocks of the second most recent backup that have not been returned yet, and so on until the least recent backup. During the operation of the iterator, it inevitably encounters a situation where an older block overlaps with a more recent block. In this case, the iterator simply returns the subset of the older block not covered by the more recent block. See Figure 4.2 for a visualization. For those particularly interested in the implementation details, see the class `DeltaFilter` in the module `adaptors.py`, and the module `algorithms/loaders.py`.

4.5.7 Installation

The installation of the system, or rather the VDDK toolkit the system depends on, is a bit peculiar. The main problem with the toolkit is that it relies on various common custom-built libraries (C++ standard library, `curl`, `openssl`, `sqlite3`) that cannot be installed in the standard location along the VDDK specific libraries. Moreover, they cannot be installed in a local destination (like `/usr/lib/vddk` or `/usr/local/lib`) and registered at the end of the dynamic loader chain using `/etc/ld.so.conf` and `ldconfig` because the main VDDK library does not have any protection against loading of mismatching versions (at least of `openssl`), which results in loading of system libraries instead of the

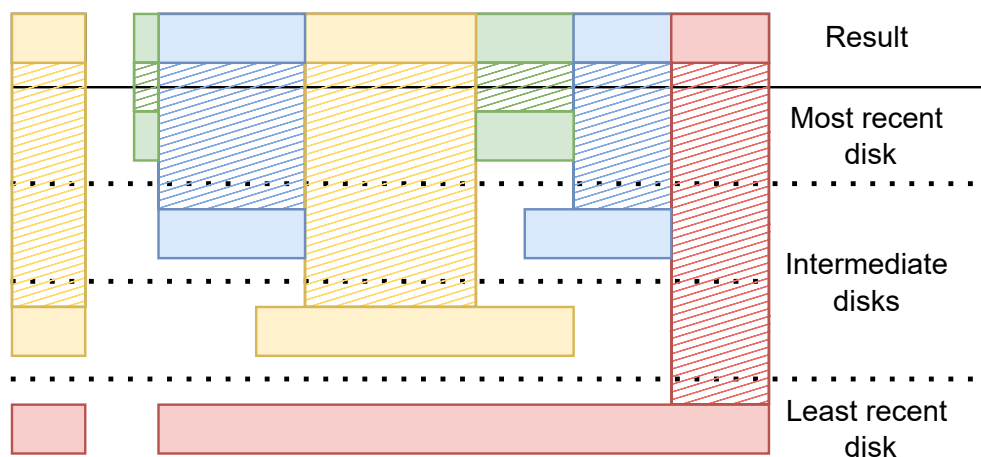


Figure 4.2: Visualization of the disk chain iterator. The iterator first returns the blocks of the green disk, then it returns a subset of the blue disk that is not covered by the green disk, and so on until the red disk.

custom-built ones. Finally, the custom-built libraries could be registered at the beginning of the dynamic loader chain, which results in loading of the libraries by ordinary applications.

Therefore, the simplest (and currently we cannot think of any other) solution is to use the environment variable `LD_LIBRARY_PATH` when running the software dependent on the VDDK toolkit, which includes the installation suite of the application.

4.5.8 Bindings

The VDDK toolkit⁴ or rather its `VixDiskLib` component is a C library that needs bindings to use from Python. We use the `pybind11` library to generate them. While the initial plan was to bind the functions offered by the toolkit without changes, this proved impossible on multiple levels.

Firstly, the library uses pointers to incomplete types (please note, that we are discussing C++ code), which the `pybind11` library cannot process because it uses type introspection. We bypass this limitation by wrapping the pointers in a `struct` and using an instance of the structure in the pointer's place.

Secondly, some functions of the toolkit return values through pointer arguments, which is not possible in Python. Therefore, such arguments are also wrapped in structures and the affected functions have their arguments changed to references to the structures.

Finally, some functions have a very cumbersome API. While it does not prevent binding, we used the occasion to create wrappers. We will not list all the changes we introduced, those interested should consult the `vixdisklib/vix-disklib.cpp` file in the root of the repository, which contains the implementation and an exhaustive list of the modifications.

The final problem is the bindings need to be compiled and linked with the library. The `VixDiskLib` component, is distributed as a shared object file (`.so`)

⁴At this point dreaded toolkit.

that can only link with code compiled with presumably the same version of the compiler. The only version that is confirmed working is the GNU compiler suite versions 10.x. However, there may be other versions or even implementations that work. We can attest, that GNU compiler suite versions 11.x are not among them. We believe that when a new version of the toolkit is released, the compiler version will also advance.

4.5.9 Communication between core and client handling process

When the client communicating process submits a client request to the core, it waits for its answer. If the wait timeouts (the timeout is to prevent hanging of the communicating process when the core terminates, and to signal irregularities in the system if the core does not answer within a reasonable period) the response of the core is not awaited which results in its hanging in the pipe where it is going to pollute the next client session. Therefore, requests are tagged when submitted to the core, and the core uses the same tag for corresponding responses. Any responses whose tag is not the same as that of the request are dropped by the client communicating process because they are artifacts of the previous requests.

4.6 Software extensibility

This section lists extensibility options of the system. We attempt to list all intended sites, although, there might be other possibilities for extensions.

4.6.1 Mixin

There are two methods of customization of backup algorithms. The first is through user-defined configuration. This entails block and buffer adaptors, and backing bases. The other method is through a mixin. The possibility is mentioned in Section 4.3 discussing the algorithms. While the first method is aimed at the user, the mixin is provided for the developer.

All platform-specific functionality is provided by the mixin. The main idea is that the developer only needs to implement the mixin when contributing a new platform, and the algorithms do not need to change. Moreover, the mixin can also customize which implementation is used for certain stages of the algorithms. For example, when parallel backup of disk blocks can be easily implemented for vSphere, but an implementation for Hyper-V requires considerable effort, the mixin can choose a serial algorithm for Hyper-V.

The mixin must implement the `Implementation` protocol from module `interfaces.py` and has to be registered using the class decorator `register_implementation` from module `platforms.py`. Fulfilling both steps makes the mixin available to the algorithms. See the `vsphere/implementation.py` module on vSphere implementation of the mixin, and the following section on other steps needed to contribute a virtualization stack.

4.6.2 Virtualization stacks

To extend the system with support for another virtualization stack, one must do the following (all mentioned protocols are from module `interfaces.py`, unless otherwise said):

- Introduce a new platform by adding a field to `TechnologyType` enumeration in module `enums.py`. Beware, the system is very strict with checking that the platform has all following entities implemented, so this step might be the last, nevertheless, the enumeration field is needed in all the other entities.
- Provide a database model of the new platform. At least the specialization of `Inventory`, `VirtualMachineExtraBackupConfig`, `Backup` and `Disk` objects have to be implemented. To implement a specialization, use subtyping. See the mentioned entities and their vSphere specializations in module `model.py`, and consult SQLAlchemy documentation ⁵.
- Implement all entities used by the algorithms: `Inventory`, `VirtualMachine`, `Snapshot`, `Disk` and `Task` protocols. Consult their vSphere implementation in `vsphere/entity/*.py` and use in `algorithms/*.py`.
- Register the inventory type using `register_inventory_type` class decorator from module `platforms.py`.
- Implement all supporting structures for the entities from the last items: `VirtualMachineBuilder` and `VmExtraConfig` protocols. Examine their vSphere implementation in `vsphere/entity/vm.py` and use in `daemon.py`, `algorithms/*.py` and `handlers.py`.
- Register the machine builder and configuration using `register_vm_builder_type` and `register_vm_extra_config_type` class decorators from module `platforms.py`, respectively.
- Implement any extra pre-, post-snapshot and disk savers the platform might need. The relevant protocols are `ExtraPreSnapshotBackupSaver`, `ExtraPostSnapshotBackupSaver` and `ExtraDiskSaver`, respectively. Consult their vSphere implementation in `vsphere/algorithms.py` and use in `algorithms/*.py`.
- Implement any remaining functionality required by the mixin discussed in the previous section.
- Implement the mixin.

The system is almost ready to accommodate the contribution. The very last step is to migrate the database. Use `alembic`, which can auto-generate the migrations. The migration has to be manually checked, however, for the tool might make mistakes. Especially if the new model uses the custom UUID type `CustomUUID`. See `model.py` for details. Once the migration script is created, it has to be applied to the database. Once the migration finishes, the system supports the

⁵<https://docs.sqlalchemy.org/en/14/orm/inheritance.html>

implemented virtualization platform. While this might seem relatively little work, please see the whole vSphere implementation in `vsphere/*`. Beware, it took up the majority of the development time.

4.6.3 Database engine change

The SQLite database engine can be switched for a different one. There are several things to consider when changing the engine. Firstly, whether the engine supports native UUID types. If it does, the implementation of the custom UUID type in `model.py` has to be revised, for it is not tested with such a database.

Secondly, if the engine supports multiversion concurrency control, the isolation level should be changed to serializable and the custom concurrency control implemented in `processes.py` dropped. Moreover, the implementation of request handlers in `handlers.py` has to be modified to support rollbacks forced by the database concurrency control. This will effectively allow concurrent restore operations of the same virtual machine, as discussed in Section 3.17.1.

Finally, the configuration of the database URL used by SQLAlchemy must be extended to accommodate authentication with a server style database engine. The current method would require having plaintext passwords in the configuration files. See `DaemonConfig` in `daemon.py` for details. Please note, that a change to the URL mechanism will require a modification of the `alembic` script `alembic/env.py`.

4.6.4 Scheduler

The scheduler can be extended in two ways. Firstly, support for operations other than backups can be introduced. Secondly, the scheduler can only be notified to cancel scheduling of a particular virtual machine, but other actions are also possible (for example, to update its schedule with a new virtual machine).

The scheduler is actually made up of two components: a scheduling process `processes.ScheduleWorker`, and a scheduling object `schedule.Scheduler`. Both components have to be maintained in concert.

To support operations other than backup, the following has to be accomplished:

- Modification of the virtual machine configuration to accommodate scheduling of other operations. See `vm.VmBackupConfig`.
- Alteration of the scheduling object to load the configuration in its `build` method.
- Modification of the scheduling process to issue the corresponding request in its `_submit` method.

These modifications are limited to the case when the scheduled operation solely depends on the virtual machine UUID, i.e., restorations cannot be implemented this way. To implement scheduled restorations, the scheduling object has to be reimplemented because the underlying priority queue identifies stored tasks by virtual machine UUIDs, and the process has to be adjusted to the changes of the object.

To support notifications of other events, the following steps have to be completed:

- Introduce a new field in the `NoticeType` enumeration in `processes.py`.
- Implement an entity producing the notifications. See Section 4.6.12 for an example.
- Let the entity notify the scheduling process through the scheduler notice queue.
- Modify the `_handle_notices` method of the process to perform the correct action.
- Extend the scheduling object with any functionality required by the action.

4.6.5 Block and Buffer adaptors

To introduce custom block or buffer adaptors, the `BlockAdaptor` or `BufferAdaptor` protocol has to be implemented, respectively. Moreover, a corresponding enumeration field has to be added to `BlockAdaptorType` or `BufferAdaptorType`. Finally, the resulting class has to be registered for use with `register_block_adaptor` or `register_buffer_adaptor` class decorator from module `adaptors.py`. Create a database model of the adaptor by subtyping `BlockAdaptor` or `BufferAdaptor` from module `model.py`, respectively. Migrate the database to accommodate the new model. With the steps completed, the adaptor is ready for use from user configuration.

Please note, that the block adaptor cannot produce a block that is a superset of the original block, and that the buffer adaptor has to be able to reverse its transformation.

4.6.6 Schedule and Retention policies

Schedule and retention policy extension mechanism is identical to the adaptor technique in Section 4.6.5. Simply replace the mentioned entities in order with: `Schedule`, `Retention`, `ScheduleType`, `RetentionType`, `register_schedule`, `register_retention`, `BackupSchedule` and `BackupRetention` (where the last two are from `model.py`). The only difference is that the policies are split into modules `schedule.py` and `retention.py`.

4.6.7 Alternative disk data storage

To provide additional storage capabilities, the following has to be implemented (again, the mechanism is analogous to adaptors and policies):

- A field in the `DataBackingType` enumeration representing the new method.
- A saver, implementing the `DataSaver` protocol and providing the desired functionality.
- A factory implementing the `DataSaverFactory` protocol, returning the saver specialized for each disk.

- A loader implementing the `DataLoader` protocol, capable of loading the data saved by the saver.
- A cleaner implementing the `DataCleaner` protocol, capable of cleaning up the saved data.
- Register the entities using `register_saver_factory`, `register_loader` and `register_cleaner` class adaptors (the saver does not need to be registered) from module `backing.py`.
- Adjust the database model by subtyping `DataBackingBase`, which represents the factory, and `DataBacking` which is used by the saver, loader and cleaner.
- Migrate the database.

The storage is now ready to be used.

4.6.8 Consolidation algorithms

To contribute additional consolidation algorithms, implement the `BackupsConsolidator` protocol, and add it to all platform mixins (field `BackupsConsolidators` of mixin `ImplementationVm`) that should benefit from the addition. Keep in mind, the order of the algorithms specified in the mixin is the same as the one applied during consolidation. There is practically no limitation on the functionality of the contributed algorithm.

4.6.9 Parallelization at the block level

Parallelization at the block level can be achieved using threads, which normally cannot execute in parallel because of the GIL. However, the majority of the work is done by the `VixDiskLib` library that does not touch any objects that are part of the interpreter. Therefore, it is possible to release the GIL when entering the code handled by the library and reacquire it on exit. See the segment binding the `VixDiskLib_ReadAsync_Custom` function in `vixdisklib/vixdisklib.cpp` on how to achieve it.

The toolkit provides the asynchronous read function `VixDiskLib_ReadAsync`. However, the function is not safe to use from the Python interpreter, for it causes segmentation faults.

To introduce parallel saving or loading, implement the `DataBlocksTransfer` or `DataBlocksReverseTransfer` protocols, respectively, and modify the mixins of platforms that should benefit from the contribution, namely fields `DataBlocksTransfer` and `DataBlocksReverseTransfer` of the `ImplementationDisk` mixin.

Finally, a few things to note about the possible implementation. The iterators providing blocks that should be saved or loaded have to be consumed in the same thread that created them because they rely on database objects that are not thread-safe. The `VixDiskLib` connection to the virtual machine can and should be shared among the threads, please see the VDDK manual for details [5]. Consequently, the low-level disk implementation `VixDisk` in `vixdiskops` has to be extended to support locking. Please note, that the operation is heavily

dependent on transfer rate of the network. Therefore, parallelization might not improve the performance if the capacity of the network is already reached. Finally, a process-based approach where each process manages its own connection is possible, although, it is more complex, see Section 4.6.11, and the number of `VixDiskLib` connections should be kept to a minimum, as they are resource heavy.

4.6.10 Parallel cleaners

The cleaning algorithm can also be parallelized, although, the operation of the algorithm is very fast as is, because it does not really do anything. The extension mechanism is similar to backups and restores. See the `ImplementationVm` and `ImplementationDisk` protocols, and the cleaning algorithm in `algorithms/-cleaners.py` on what needs to be implemented.

4.6.11 Parallelization at the disk level

Parallelization at the disk level is limited by the existence of one or more network connections to the hypervisor. Considering the only method of parallelization available at this level are processes and the entities needed for backup (or restore) have to be passed to them in pickled form, they need to implement a custom pickling mechanism because the connections cannot be pickled. To that end, the `__setstate__` and `__getstate__` methods are available, and their implementation can simply ignore the connections on serialization and establish a new connection on deserialization.

4.6.12 Runtime configuration changes

The system can be extended to support runtime configuration changes. To that end, a new request has to be introduced by implementing the `Request` protocol, a corresponding field in the `OperationType` enumeration and registering the request with `register_request_type` in `requests.py`. Moreover, a request dispatcher, which is called by the core to issue a request to the correct worker, has to be added to `blocking_dispatchers`. The request must be blocking, for it significantly modifies the database. Consider using the generic `generic_async_dispatcher` dispatcher, which should fit the purpose. A handler of the request has to be implemented. Beware that the handler should notify the scheduler that the configuration has changed. Apart from this limitation, the handler can do whatever it deems necessary. Finally, a process running the handler and any associated communication channels should be created and executed. Consider adding the communication channels to the `CommunicationFabric` which contains all other channels, and see the call to `build_workers` in `start` on how to add a process. All mentioned entities are in `processes.py`.

4.7 New vSphere version

When a new version of the vSphere stack is released, it might introduce some changes to the API the system depends on. Interaction with the VDDK toolkit

is contained in `vixdisklib/vixdisklib.cpp` and `vsphere/vixdiskops.py`. The API provided by the toolkit is fairly simple, and we do not expect it to change, nevertheless, any change should be easy to handle.

On the other hand, the `pyVmomi` library provides an abstraction of a massive API. The library is used throughout most of the `vsphere/*` modules. We only use a tiny subset of the API, so changes should be rare and easily manageable. Of particular interest are the following entities (their names are those used in the API documentation ⁶, not by the library): `VirtualMachine`, `VirtualMachineConfigInfo`, `VirtualMachineConfigSpec`, `Folder`, `VirtualMachineSnapshot`, and the types of their fields.

4.8 Implementation defects

In this section, we list known implementation defects. We suggest fixing of the defects should be the priority of future development.

4.8.1 `pydantic`

There are many classes following the same structure of providing functions and methods to serialize/deserialize itself to/from JSON, and to/from database backing objects. Exactly this functionality is provided by the `pydantic` package. However, its existence came to our attention after the implementation stage had concluded, and we were unable to refactor the code to use the package in time. For an example of the mentioned classes, see `AutoSchedule` in module `schedule.py`.

4.8.2 Database commit ordering bug

There is a single implementation bug we are aware of. It occurs in the `CleanupHandler` and `ConsolidationHandler` request handlers. The faulty behavior can only arise when the system is suddenly terminated (using `SIGKILL` or because of power loss).

The problem is the same in both sites. The system marks certain backups for cleanup, and the data of the backups is deleted before the associated database entities are removed and the change committed. This can result in corruption of a backup or backup chain if the system manages to delete some or all of the data, but is terminated before it can commit the change into the database.

The initial solution to this was to query any relevant information, commit the removal of the backups and use the information to delete the data. In this case, a sudden termination would leave some artifacts in the system (like files on a local disk) but the backups would not get corrupted. However, the ORM system used to query the database employs lazy loading of entities and once the entities are removed from the database and the change committed, they cannot be used anymore.

The behavior can be mitigated by using eager loading in the two mentioned handlers, which will leave the entities usable even though they are no longer part

⁶<https://developer.vmware.com/apis/1192/vsphere>

of the database. This change was not implemented for the lack of time. See the `handlers.py` module for more details and an outline of the solution.

4.8.3 Testing

The system has automated tests covering most functionality, not relying on any connections to a hypervisor. The functionality depending on the connections requires a mocking framework, which was not, for the lack of time, implemented, as it requires considerable amount of work. However, the code uses defensive programming and checks all constraints imposed on function arguments as well as throughout the rest of the code. Moreover, the ability of the systems running in restored virtual machines to check the consistency of their file system using checksums proved to be the most valuable testing utility.

5. Comparison

In this chapter, we provide a brief overview of software overlapping in goals with our solution and its comparison to our system.

Some comparable solutions are (the list is by no means exhaustive):

- Veeam ¹,
- Vembu ²,
- Altaro ³,
- NAKIVO ⁴, and
- Microsoft DPM ⁵.

We provide a comparison of our **vmbackupd**, Veeam and Vembu solutions in the following sections. The sources of the following information ⁶ are

- the Veeam Backup & Replication user guide [9],
- the Veeam Backup & Replication datasheet [10],
- the Vembu BDR Suite user guide [11], and
- the Vembu product web page [12].

5.1 Deployment options

An overview of platforms the solutions can be deployed to.

vmbackupd

Tested on

- Debian 11.

Should work with no modifications on

- any recent Linux distribution.

Veeam

The Veeam backup server can be deployed to

- Microsoft Windows Server 2008-2022, and
- Microsoft Windows 7-11.

¹<https://www.veeam.com/vmware-esx-backup.html>

²<https://www.vembu.com/vmware-backup/>

³<https://www.altaro.com/vmware-backup/>

⁴<https://www.nakivo.com/vmware-backup/>

⁵<https://docs.microsoft.com/en-us/system-center/dpm/back-up-vmware?view=sc-dpm-2022>

⁶Please note, that the information might be incomplete because the documentation of the products is enormous.

Vembu

The Vembu BDR backup server can be deployed to

- Microsoft Windows Server 2008-2019,
- Microsoft Windows 10, and
- Ubuntu LTS 12.04-18.04.

5.2 Supported platforms

Platforms that can be backed up using the solution.

vmbackupd

Tested with

- VMware vSphere 7.0.

According to the published API, the solution should work with

- VMware vSphere 5.5 or newer,

with no modifications to the system.

Veeam

Targeted platforms are

- AWS,
- Microsoft Azure,
- Google Cloud Platform,
- VMware vSphere 5.5 or newer,
- Microsoft Hyper-V, and
- Nutanix AHV.

Furthermore, physical machines running any mainstream operating system, and certain applications, for example SAP or database engines, are also supported.

Vembu

Targeted platforms are

- VMware vSphere 4-6.5,
- Microsoft Hyper-V, and
- AWS.

Furthermore, physical machines running Windows or Mac operating systems, and certain applications, for example database engines, are also supported.

5.3 Data deduplication

All mentioned systems support deduplication of the backed up data to a certain level.

vmbackupd

The **vmbackupd** solution supports deduplication at the level of virtual machine disks.

Veeam

The Veeam solution supports deduplication at the level of virtual machines and at the level of the backup destination.

Vembu

The Vembu solution supports deduplication at the level of virtual machine disks.

5.4 Multiple storage support

Support for multiple data storage technologies.

vmbackupd

The **vmbackupd** solution supports implementation of any data storage system, however, only local file system storage is currently implemented.

Veeam

Supported data storage approaches are

- a directly attached backup proxy,
- the ExaGrid storage appliance,
- the Quantum DXi storage appliance, and
- a cloud object storage (AWS S3, S3 compatible, Azure storage, IBM Cloud and Google Cloud).

A directly attached backup proxy is a Windows or Linux server that arranges the saving of the data proper, using local storage, NFS or tape devices.

Vembu

Supported data storage are

- local drives,
- NAS storage, and
- SAN storage.

5.5 Agentless backup

All solutions support agentless backup.

5.6 File-based backup

The `vmbackupd` solution only supports virtual machine-based backup and restoration. The Veeam and Vembu solutions support both virtual machine-based and file-based approaches.

5.7 Compression and Encryption

Support for data compression and encryption.

vmbackupd

Compression and encryption are not implemented, nevertheless, the functionality can be easily supplied by the User. See Section 4.6.5.

Veeam

The Veeam solution supports several levels of compression, compromising between compression ratio and CPU usage. It also supports AES-256 based encryption of the data.

Vembu

The Vembu solution enables compression and encryption, using AES-256, automatically.

5.8 Extensibility

This section provides an overview of user extension options.

The `vmbackupd` solution is designed to be extensible by the User. Most of the lacking features mentioned in this chapter can be added to the software with no or only minor design changes, albeit certain functionality would require considerable effort.

The Veeam and Vembu are proprietary solutions that do not seem to have any methods of extension.

Conclusion

This chapter contains a reiteration of our goals, an overview of their accomplishment, and possible extensions.

Goals and accomplishments

Our goal was to implement a virtual machine backup system. The implementation provided by us is capable of backing up virtual machines running on the vSphere virtualization stack, while keeping the system open to extensions of other virtualization technologies. The system can also restore the backed up machines into a user-defined destination. The software performs delta backups when possible to minimize the size of backups, it can also delete old backups according to a configured policy, which includes consolidation of delta backups. The user can configure the location the machines are backed up into. Moreover, They can apply transformations to the data of the machines, and the transformations can be chained, which enables future compression and encryption of the data. The system is configured by the user to periodically perform operations such as backups.

The main goal of the system was modularity. The software can be extended in many ways. We have already mentioned implementation of another virtualization stack. However, there are many more extension sites, most of which require only minutes to extend. For example, more scheduling and retention policies can be introduced, novel methods of data storage are trivial to implement, and disk data transformers can be added with little modification to the system.

The JSON files used to configure the system allow infinite expressiveness of intent, and the system expects the files to contain heterogeneous data. Moreover, all data ingested from the user is verified, and the system diagnoses issues with the configuration immediately.

The application informs the administrator of all important internal events occurring. The application provides diagnostics of any unexpected state arising in the system. The software is particularly reliable, for its critical components are thoroughly tested, and the system is developed according to the principles of defensive programming.

Finally, all the aforementioned features are implemented in as simple a manner as possible, which complements extensibility and reliability.

Possible extensions

This section comprises our ideas and suggestions for extensions.

Our first suggestion is to implement compression and encryption of the backed up data. See Section 4.6.5.

Another very helpful contribution would be introduction of a consolidation algorithm targeting the local file system data storage. The generic algorithm tends to fragment the data, and much of the data could be simply moved instead of copied, which would improve the performance a great deal. See Section 4.6.8.

A very useful feature would be a testing framework which would enable automatic testing of the whole system. Beware, however, that this contribution requires considerable effort.

An interesting feature would be implementation of another disk storage. However, the provided file system storage is fairly capable, as many technologies provide file system mounting facilities. See Section 4.6.7.

A useful contribution would be runtime support for reconfiguration. The current method of restarting is limiting. See Section 4.6.12.

On the other hand, we do not suggest that implementation of parallelization should be prioritized, for we are skeptic of the ratio of invested work to provided value. A much better time investment would be checking whether the solution works with advanced transport modes and possibly improving the code to accommodate them. See Sections 4.6.9 and 4.5.2.

Finally, the largest possible extension is implementation of support for another virtualization stack. We believe this endeavor should be postponed until all the previous suggestions are implemented, for it requires many times more the effort needed to implement the suggestions. See Section 4.6.2.

Bibliography

- [1] IBM. What is virtualization?, 2019. <https://www.ibm.com/cloud/learn/virtualization-a-complete-guide>.
- [2] Red Hat. What is a virtual machine (vm)?, 2022. <https://www.redhat.com/en/topics/virtualization/what-is-a-virtual-machine>.
- [3] Red Hat. What is a hypervisor?, 2020. <https://www.redhat.com/en/topics/virtualization/what-is-a-hypervisor>.
- [4] VMware. virtual-disks. <https://github.com/vmware/virtual-disks>, 2022.
- [5] VMware. *Virtual Disk Development Kit Programming Guide*. VMware, 2022. <https://developer.vmware.com/docs/15498/virtual-disk-development-kit-programming-guide--7-0-3-->.
- [6] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. pybind11 – seamless operability between c++11 and python, 2017. <https://github.com/pybind/pybind11>.
- [7] Michael Bayer. Sqlalchemy. In Amy Brown and Greg Wilson, editors, *The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks*. aosabook.org, 2012.
- [8] NumPy Developers. Numpy documentation.
- [9] Veeam. User guide for vmware vsphere, 2022. <https://helpcenter.veeam.com/docs/backup/vsphere/overview.html>.
- [10] Veeam. *Veeam Backup & Replication Product Overview*. Veeam, 2021. https://www.veeam.com/veeam_backup_datasheet_ds.pdf.
- [11] Vembu. *Vembu BDR Suite User Guide*. Vembu, 2022. <https://www.vembu.com/pdf/guide/vembu-bdr-suite-user-guide.pdf>.
- [12] Vembu. Vembu backup & disaster recovery, 2022. <https://www.vembu.com/>.

List of Figures

3.1	The figure illustrates the overall design of the system. It comprises a client (usually <code>vmbackupctl</code> contributed by us), the <code>vmbackupd</code> daemon, a database for data persistence, and communication channels to any hypervisor control suites.	16
3.2	The daemon can communicate with the vCenter server, which is one of the supported hypervisor control suites, through a plethora of channels: a SOAP API, a REST API and an unspecified proprietary API.	17
3.3	The daemon uses the SQLite database engine for data storage and its accompanying library for communication with the engine. . . .	20
3.4	The overall architecture of the daemon. The light gray components are individual processes, while the white “Daemon core” component is the parent process. Arrows signify communication between components and its direction, while the blue and green colors of the arrows indicate synchronicity or asynchronicity of the traffic, respectively. The UNIX domain socket is only accessed by the “Client communication” component, while the hypervisor control suites and the database are accessed by most of the segments (the connections are not shown in the Figure to prevent clutter, as are not bookkeeping connections between the core and all other components).	21
3.5	The segment of the database model capturing inventory and virtual machine relationships. See Sections 3.10.1, 3.10.2 and 3.10.6 for details.	25
3.6	The portion of the model describing backups and their relationships. See Section 3.10.3.	26
3.7	The segment of the model focusing on disks and their relationships. See Sections 3.10.4 and 3.10.5.	27
3.8	The section of the model capturing disk backup configurations and their relationships. See Section 3.10.7. The red segments and their lines denote an exclusive relationship. The disk backup configuration is in a relationship with either a virtual machine or a disk, but not both.	28

4.1	This is the same overall architecture of the daemon as shown in Figure 3.4. The arrows still signify communication between components and its direction, but the blue and green colors of the arrows indicate whether a Pipe or Queue is used, respectively. Moreover, the bookkeeping connections (omitted in the previous figure) are introduced as a blue circle and diamonds. The color blue signifies that they are Pipes , the circle is the sending multi-side (there is a sending side for each receiving side) and the diamonds are the receiving side. Additionally, the inside structure of the worker groups is shown (only for the “Backup group” but the other groups follow the same schema). The receiving and sending Queues used to communicate with a particular group are shared by the workers in the said group.	35
4.2	Visualization of the disk chain iterator. The iterator first returns the blocks of the green disk, then it returns a subset of the blue disk that is not covered by the green disk, and so on until the red disk.	48

List of Tables

List of Abbreviations

A. Attachments

A.1 User manual

This section contains an excerpt from the manual distributed with the software. See the `README.md` file in the root of the repository for the complete version. We suggest using the `README.md` file because of its superior code listings, examples and completeness.

A.1.1 Prerequisites

The system can run on any recent Linux distribution, although, it's only been tested on Debian 11.

The system has the following prerequisites:

- An interpreter of Python 3.8 or newer; the only supported implementation is `CPython`.
- Any recent `pip`.
- The C++ frontend of the GNU toolchain (`g++`) version 10.x (only tested with 10.2). The reason is that the `VixDiskLib` library the system depends on is distributed as a shared object file that cannot link with its bindings compiled from `pybind11` if the version of the used compilers mismatch. See Section 4.5.8.

A vCenter server installation, appropriate permissions, and network access to the vCenter server is also needed.

There is practically no limitation on hardware, any reasonably recent machine should fare fine. Keep in mind, however, that the disk data of a backed up VM will be stored on the machine running this software. In other words, the machine requires enough free space.

Finally, VDDK, a proprietary toolkit, is required to run this software. See the following section on how to obtain the toolkit.

A.1.2 VDDK acquisition

The VDDK toolkit can be downloaded free of charge from the VMware website¹, although, a VMware account is needed. There are two download options, one is a Windows distribution, and the other a GNU/Linux distribution. The GNU/Linux distribution is needed.

A.1.3 VDDK installation

After acquisition of the VDDK toolkit in the previous section, it has to be installed. The main problem with the toolkit is that it relies on various common custom-built libraries (C++ standard library, `curl`, `openssl`, `sqlite3`) that

¹<https://developer.vmware.com/web/sdk/7.0/vddk>

cannot be installed in the standard location along the VDDK specific libraries. Moreover, they cannot be installed in a local destination (like `/usr/lib/vddk` or `/usr/local/lib`) and registered at the end of the dynamic loader chain using `/etc/ld.so.conf` and `ldconfig` because the main VDDK library does not have any protection against loading of mismatching versions, which results in loading of system libraries instead of the custom-built ones. Finally, the custom-built libraries could be registered at the beginning of the dynamic loader chain, which results in loading of the custom-built libraries by ordinary applications.

Therefore, the simplest (and currently we cannot think of any other) solution is to use the environment variable `LD_LIBRARY_PATH` when running the software dependent on the VDDK toolkit. Consequently, the VDDK distribution can be installed wherever.

A.1.4 Installation

The installation instructions assume a copy of this software has already been acquired, either by cloning of its repository or by downloading of the official version uploaded to SIS, and the VDDK toolkit has been downloaded and installed.

A Python virtual environment should be created (using `venv`) and activated. The manual expects the environment to be activated from this point on.

To install the software, run the following in the root of the repository:

```
VDDK_CUSTOM_PATH=/path/to/vddk/toolkit/root/directory pip install .
```

For example, if the VDDK distribution is installed in `/home/EVA/vddk` the command should be:

```
VDDK_CUSTOM_PATH=/home/EVA/vddk pip install .
```

The command should install the `vmbackupd` daemon and `vmbackupctl` control utility. The variable `VDDK_CUSTOM_PATH` is used to pass the VDDK installation path to the setup script because using a custom argument with `pip install` breaks wheel installations.

A.1.5 Configuration

The system is primarily configured through configuration files and secondarily through command line arguments.

Configuration directory structure

The configuration files can be located anywhere, as long as the structure of the directory containing them is as follows:

- The daemon configuration file is named `config.json` and is in the root of the configuration directory.
- All inventory configuration files are in the `platforms` directory, which in turn is in the root of the main directory. Each inventory configuration file name ends with `.json`, the name can be arbitrary, and the file can be in an additional directory or a directory tree.

- All virtual machine configuration files are in the **vms** directory, which in turn is in the root of the main directory. Each VM configuration file name ends with **.json**, the name can be arbitrary, and the file can be in an additional directory or a directory tree.

The directory holding the described structure is the root configuration directory (and as such should be passed to the **vmbackupd** daemon, see the command line arguments section).

Daemon, VM, and inventory configuration files each have a different format. They are all JSON files and their format is as follows.

Daemon configuration file

The daemon configuration file is used to configure the daemon. See the full manual for its format.

Inventory configuration file

Inventory configuration files are used to configure individual inventories available to the system. See the full manual for their format.

Virtual machine configuration file

Virtual machine configuration files are used to configure individual virtual machines set to be tracked by the system. See the full manual for their format.

Command line arguments

The **vmbackupd** daemon has the positional argument **config_path** and three optional arguments **--skip-config**, **--log-debug** and **--debug**.

The **config_path** argument specifies the path to the root configuration directory (see the “Configuration directory structure” section).

The **--skip-config** argument instructs the daemon to skip the reading of inventory and virtual machine configuration files.

The **--log-debug** argument enables logging of “DEBUG” level messages.

The **--debug** argument enables the debug mode, which executes a different path and does not start the system as a daemon. It implies the **--log-debug** argument. Used for debugging.

A.1.6 Examples

See the **examples** directory in the root of the repository for configuration examples that can be easily adapted for any environment.

A.1.7 Database preparation

To create the database, the “Daemon configuration file” has to exist and contain at least the **dbUrl** field.

To create the database (at the URL specified in **dbUrl**) the following should be run in the root of the repository:

```
LD_LIBRARY_PATH=/path/to/vddk/toolkit/root/directory/lib64 \  
VMBACKUPD_CONFIG_PATH=/path/to/daemon/config/file \  
alembic upgrade head
```

Which will create an SQLite library using the application's model.

For example, if the VDDK toolkit is installed into `/home/EVA/vddk` and the daemon configuration is located at `/home/EVA/vmbakupd_cfg/config.json` the command would be:

```
LD_LIBRARY_PATH=/home/EVA/vddk/lib64 \  
VMBACKUPD_CONFIG_PATH=/home/EVA/vmbakupd_cfg/config.json \  
alembic upgrade head
```

Please note the `lib64` at the end of the `LD_LIBRARY_PATH` value.

A.1.8 Running

Now the daemon can be started. All previous steps have to be completed for the daemon to function properly.

At this point, the daemon config and inventory configuration files should be verified to have the correct format and intended contents. No virtual machine configuration files should exist at this point (if this is the very first time the system is run), for there is no way to determine the virtual machine UUIDs that have to be specified in the configuration files.

The following can be run to start the daemon:

```
LD_LIBRARY_PATH=/path/to/vddk/toolkit/root/directory/lib64 \  
vmbakupd /path/to/daemon/config/root/directory
```

This should start the daemon and its logs should start appearing in the console.

The next step is to use the functionality of the daemon to determine the UUIDs of the virtual machines that are intended to be backed up, write configuration files for the machines, and restart the daemon to update its configuration, which will result in a fully operational system. See the following 'Control' and 'Requests' sections for more information on how to obtain the UUIDs and operate the system.

A.1.9 Monitoring

The daemon logs all its messages to the standard output. `INFO` messages are safe to ignore, `WARNING` messages signalize that something out of the ordinary, but expected, has happened, and `ERROR` messages signalize bugs.

A.1.10 Control

The daemon can be controlled in two ways: through system interrupts and over the socket configured in the daemon configuration file with the `daemonSocketPath` option.

System interrupts

The `SIGINT` and `SIGTERM` system interrupts can be sent to the daemon to terminate it.

Socket communication

The daemon can be communicated with using the exposed socket. There are several requests that can be sent to the daemon, which are detailed in the 'Requests' section. This section focuses on the communication protocol. Consider using the `vmbackupctl` utility described in its respective section, which already implements the protocol and requests.

The protocol is as follows:

- The size of the JSON request encoded in UTF-8 is sent as a four-byte big-endian unsigned integer.
- The encoded request is sent.
- The size of a JSON response encoded in UTF-8 is received as a four-byte big-endian unsigned integer.
- The encoded response is received.

A.1.11 Requests

There are several supported requests. See the full manual for their API, and Section 3.6 for more details.

A.1.12 Responses

There are several possible responses to the requests. See the full manual for their API, and Section 3.8 for more details.

A.1.13 `vmbackupctl`

The `vmbackupctl` utility can be used to send requests and receive responses as described above. It has the following command line arguments:

- `address` is the path to the `vmbackupd` socket, which will be used for communication.
- An action which exactly corresponds to one of the described requests, and is one of the following: `backup`, `restore`, `cleanup`, `consolidation`, `inventory_summary`, `tracked_summary`, `state_summary`, `blocked_summary` and `terminate`.
- Any action-dependent arguments which are identical to those of the corresponding request. Please see the 'Requests' section in the full manual for details on each request's arguments.

A.1.14 Tests

The tests can only be run if the `vmbackupd` package is installed.

The tests can be executed by running the following command in the `test` directory:

```
LD_LIBRARY_PATH=/path/to/vddk/toolkit/root/directory/lib64 pytest
```