Towards Efficient Virtual Appliance Delivery with Minimal Manageable Virtual Appliances

Gabor Kecskemeti, *Member, IEEE*, Gabor Terstyanszky, Peter Kacsuk, and Zsolt Nemeth

Abstract—Infrastructure as a Service systems use virtual appliances to initiate virtual machines. As virtual appliances encapsulate applications and services with their support environment, their delivery is the most expensive task of the virtual machine creation. Virtual appliance delivery is a well-discussed topic in the field of cloud computing. However, for high efficiency, current techniques require the modification of the underlying IaaS systems. To target the wider adoptability of these delivery solutions, this article proposes the concept of minimal manageable virtual appliances (MMVA) that are capable of updating and configuring their virtual machines without the need to modify IaaS systems. To create MMVAs, we propose to reduce manageable virtual appliances until they become MMVAs. This research also reveals a methodology for appliance developers to incorporate MMVAs in their own appliances to enable their efficient delivery and wider adoptability. Finally, the article evaluates the positive effects of MMVAs on an already existing delivery solution: the Automated Virtual appliance creation Service (AVS). Through experimental evaluation, we present that the application of MMVAs not only increases the adoptability of a delivery solution but it also significantly improves its performance in highly dynamic systems.

Index Terms—Cloud computing, manageable, virtualization, deployment, virtual appliance

1 Introduction

Infrastructure as a SERVICE (IaaS) cloud systems [1], [2] offer interfaces to create, destruct and manage virtual machines (VMs—[3], [4], [5]). Even early IaaS systems [6], [7], [8] were marketed as a way to increase elasticity of their user's infrastructure. Such elasticity allows dynamic infrastructure scaling behind the user's services to meet their actual demands. To allow infrastructure scaling, applications and services are hosted in virtual machines deployed in IaaS systems. VMs are created by instantiating virtual appliances (VA—[9], [10]) that encapsulate the required applications or services (the *key functionality*) with their support environment (e.g., an operating system and several software libraries).

When users start to exploit the elasticity of their infrastructure, they frequently create and destruct virtual machines thus reduce the lifetime of average VMs. Consequently, VM creation time becomes more dominant in IaaS usage. As the dynamism increases in the infrastructure, time spent on VM creation becomes comparable to actual VM usage times. Thus, VM creation time becomes a serious obstacle before highly dynamic systems [28]. By analyzing virtual machine creation, researchers revealed [11], [12] that VM creation time is mostly dependent on *virtual appliance delivery* to the physical machine of the future VM.

Therefore, new methods were proposed to *optimize delivery time* by either pre-optimizing VA size or by optimizing the transfer from the VA storage (called repository) to the future execution hosts. First, *size optimization* [13], [14] removes appliance contents not related to the key functionality. This action is time consuming, thus appliance developers only apply size optimization prior publishing the VA if frequent deployments are expected. Second, *transfer optimization* techniques [12], [15], [16] analyze appliances to allow caching common appliance parts in multiple repositories (supporting repository hierarchies, multiple Amazon S3 regions or third party repositories). Hence, appliances are rebuilt from parts downloaded from repositories offering the best available network speeds.

Both delivery time optimization approaches require the *alteration of the virtual appliance* just before the appliance is executed. For efficiency reasons, this alteration should be performed on the future execution host of the appliance. However, current optimization solutions imply the modification of the underlying IaaS system so that third parties could modify the appliance right before execution. Without evident financial gains, commercial providers (e.g., [6], [17], [18]) have no incentive for adopting these solutions. To enable their adoption, this article proposes an approach that *avoids changes in IaaS systems* but still *allows appliance alteration* operations

at ap

capabilities could be utilized to alter their contents. Thus, no

- *G. Kecskemeti is with the Distributed and Parallel Systems group at University of Innsbruck, 6020 Innsbruck, Austria, on leave from MTA SZTAKI, the Computer and Automation Research Institute of the HAS. E-mail: gabor@dps.uibk.ac.at; kecskemeti.gabor@sztaki.mta.hu.*
- *G. Terstyanszky is with the Centre of Parallel Computing, University of Westminster, 309 Regent Street, W1B 2UW London, U.K. E-mail: G.Z.Terstyanszky@westminster.ac.uk.*
- *P. Kacsuk is with the Centre of Parallel Computing, University of Westminster, 309 Regent Street, W1B 2UW London, U.K. and also with the Laboratory of Parallel and Distributed Systems, Computer and Automation Research Institute of the Hungarian Academy of Sciences, 1111 Budapest, Hungary. E-mail: kacsuk.peter@sztaki.mta.hu.*
- *Z. Nemeth is with the Laboratory of Parallel and Distributed Systems, Computer and Automation Research Institute of the Hungarian Academy of Sciences, 1111 Budapest, Hungary. E-mail: nemeth.zsolt@sztaki.mta.hu.*

Manuscript received 6 Oct. 2012; revised 2 Feb. 2013; accepted 7 Feb. 2013.

(Duplicate first page for exhibit labeling, per 37 C.F.R. § 42.63(d)(2)(ii).)

MICROSOFT CORP.
EXHIBIT 1014

Digital Object Identifier no. 10.1109/TSC.2013.12

Authorized licensed use limited to: Klarquist Sparkman LLP. Downloaded on November 20,2025 at 21:18:46 UTC from IEEE Xplore. Restrictions apply.
1939-1374 © 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

Towards Efficient Virtual Appliance Delivery with Minimal Manageable Virtual Appliances

Gabor Kecskemeti, *Member, IEEE*, Gabor Terstyanszky, Peter Kacsuk, and Zsolt Nemeth

Abstract—Infrastructure as a Service systems use virtual appliances to initiate virtual machines. As virtual appliances encapsulate applications and services with their support environment, their delivery is the most expensive task of the virtual machine creation. Virtual appliance delivery is a well-discussed topic in the field of cloud computing. However, for high efficiency, current techniques require the modification of the underlying IaaS systems. To target the wider adoptability of these delivery solutions, this article proposes the concept of minimal manageable virtual appliances (MMVA) that are capable of updating and configuring their virtual machines without the need to modify IaaS systems. To create MMVAs, we propose to reduce manageable virtual appliances until they become MMVAs. This research also reveals a methodology for appliance developers to incorporate MMVAs in their own appliances to enable their efficient delivery and wider adoptability. Finally, the article evaluates the positive effects of MMVAs on an already existing delivery solution: the Automated Virtual appliance creation Service (AVS). Through experimental evaluation, we present that the application of MMVAs not only increases the adoptability of a delivery solution but it also significantly improves its performance in highly dynamic systems.

Index Terms—Cloud computing, manageable, virtualization, deployment, virtual appliance

1 Introduction

Infrastructure as a SERVICE (IaaS) cloud systems [1], [2] offer interfaces to create, destruct and manage virtual machines (VMs—[3], [4], [5]). Even early IaaS systems [6], [7], [8] were marketed as a way to increase elasticity of their user's infrastructure. Such elasticity allows dynamic infrastructure scaling behind the user's services to meet their actual demands. To allow infrastructure scaling, applications and services are hosted in virtual machines deployed in IaaS systems. VMs are created by instantiating virtual appliances (VA—[9], [10]) that encapsulate the required applications or services (the *key functionality*) with their support environment (e.g., an operating system and several software libraries).

When users start to exploit the elasticity of their infrastructure, they frequently create and destruct virtual machines thus reduce the lifetime of average VMs. Consequently, VM creation time becomes more dominant in IaaS usage. As the dynamism increases in the infrastructure, time spent on VM creation becomes comparable to actual VM usage times. Thus, VM creation time becomes a serious obstacle before highly dynamic systems [28]. By analyzing virtual machine creation, researchers revealed [11], [12] that VM creation time is mostly dependent on *virtual appliance delivery* to the physical machine of the future VM.

Therefore, new methods were proposed to *optimize delivery time* by either pre-optimizing VA size or by optimizing the transfer from the VA storage (called repository) to the future execution hosts. First, *size optimization* [13], [14] removes appliance contents not related to the key functionality. This action is time consuming, thus appliance developers only apply size optimization prior publishing the VA if frequent deployments are expected. Second, *transfer optimization* techniques [12], [15], [16] analyze appliances to allow caching common appliance parts in multiple repositories (supporting repository hierarchies, multiple Amazon S3 regions or third party repositories). Hence, appliances are rebuilt from parts downloaded from repositories offering the best available network speeds.

Both delivery time optimization approaches require the *alteration of the virtual appliance* just before the appliance is executed. For efficiency reasons, this alteration should be performed on the future execution host of the appliance. However, current optimization solutions imply the modification of the underlying IaaS system so that third parties could modify the appliance right before execution. Without evident financial gains, commercial providers (e.g., [6], [17], [18]) have no incentive for adopting these solutions. To enable their adoption, this article proposes an approach that *avoids changes in IaaS systems* but still *allows appliance alteration* operations to be performed on the execution host.

Our research reveals that appliances with *management capabilities* could be utilized to alter their contents. Thus, no

- *G. Kecskemeti is with the Distributed and Parallel Systems group at University of Innsbruck, 6020 Innsbruck, Austria, on leave from MTA SZTAKI, the Computer and Automation Research Institute of the HAS. E-mail: gabor@dps.uibk.ac.at; kecskemeti.gabor@sztaki.mta.hu.*
- *G. Terstyanszky is with the Centre of Parallel Computing, University of Westminster, 309 Regent Street, W1B 2UW London, U.K. E-mail: G.Z.Terstyanszky@westminster.ac.uk.*
- *P. Kacsuk is with the Centre of Parallel Computing, University of Westminster, 309 Regent Street, W1B 2UW London, U.K. and also with the Laboratory of Parallel and Distributed Systems, Computer and Automation Research Institute of the Hungarian Academy of Sciences, 1111 Budapest, Hungary. E-mail: kacsuk.peter@sztaki.mta.hu.*
- *Z. Nemeth is with the Laboratory of Parallel and Distributed Systems, Computer and Automation Research Institute of the Hungarian Academy of Sciences, 1111 Budapest, Hungary. E-mail: nemeth.zsolt@sztaki.mta.hu.*

Manuscript received 6 Oct. 2012; revised 2 Feb. 2013; accepted 7 Feb. 2013. Date of publication 24 Feb. 2013; date of current version 13 June 2014.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSC.2013.12

Authorized licensed use limited to: Klarquist Sparkman LLP. Downloaded on November 20,2025 at 21:18:46 UTC from IEEE Xplore. Restrictions apply.
1939-1374 © 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

changes are required in the IaaS systems because the necessary alterations are realized in a virtual machine. We present that delivery time optimization can be supported by a few content management operations (e.g., addition/removal, configuration of software components). However, current appliances do not offer these operations. Therefore, we propose the concept of *minimal manageable virtual appliances* (MMVA) that are special appliances to be embedded in other appliances.

This research focuses on delivery optimization techniques that support virtual machine creation in two phases: 1) initialize a VM using only the MMVA, and 2) automatically alter the new VM according to deployment needs. We also presume that appliances are often instantiated and decommissioned, thus, the initial investment (e.g., in terms of time) on delivery optimization and MMVA embedding can be regained during the lifetime of the VA. Appliance developers decide on the use of these optimization techniques.

The feasibility of the MMVA concept is revealed by presenting its effects and the necessary changes on an existing delivery optimization system: the Automated Virtual appliance creation Service (AVS—detailed in [19]). We reveal that using the management capabilities could significantly increase the performance of the operations of the AVS (e.g., appliance size optimization time can be reduced by 33 percent). We also demonstrate significant increase in deployment efficiency (11 percent delivery time reduction) with a prototype implementation of the MMVA concept on a private cloud infrastructure.

The rest of the article is organized as follows. Section 2 presents the related research results. Then, Section 3 overviews the foundations of appliance delivery through a basic scenario. Later, Section 4 defines the concept of MMVAs and introduces new ways to create or extend virtual appliances based on this concept. Afterwards, Section 5 analyzes the effects of MMVAs on appliance delivery methods of the AVS service. Finally, Section 6 discusses our implementation and its evaluation.

2 Related Works

Nishimura *et al.* [12] identify scalability issues in appliance delivery systems during on-demand cluster deployment. They propose a technique combining pipelined virtual appliance delivery and fine-grained virtual machine customization. To decrease future appliance instantiation latency, this technique automatically caches frequently requested packages in a generic virtual appliance. The efficiency of their solution is demonstrated by installing a virtual cluster of over 100 nodes. However, their system is not applicable to commercial IaaS systems because their components alter IaaS behavior.

Similarly to Nishimura's results, Zhang *et al.* [11] introduces special virtual appliances that store the most commonly used software components in the deployment requests of the users. They refer to these special appliances as Typical Virtual Appliances (or TVAs). They identify possible TVAs in the system by clustering the available software components with the *k-means* algorithm. However, their article remains at the conceptual level and does not consider constraints required for adoptability in non-simulated environments.

Bradshaw *et al.* [15] build on two existing tools (the `bcfg2` [20] and the workspace service [7]) to deliver contextualized virtual appliances promptly before their deployment. Their work discusses a technique to define and construct virtual appliances before delivery and also discusses simple contextualization tasks (e.g., security certificate and IP address propagation) that can be done after appliance instantiation. However, the proposed technique is not widely applicable because it is tailored to the previously mentioned tools.

Jin *et al.* [21] presents the concept of virtualization integrator that allows rapid provisioning, aggregated self-monitoring, simplified service management and automated consolidation of virtual appliance ensembles. These ensembles allow multiple appliances to be deployed and managed as a single system. In contrary to our research, their approach handles virtual appliances as the smallest building blocks of service-based systems.

In [22], Lutterkort and McLoughlin introduce the concept of manageable virtual appliances. These special virtual appliances allow enterprise system administrators to automatically manage virtual machines based on them. They enable enterprise level automation through accompanying the appliance with its (automatically created) recipe that defines the way (and optionally the history) the appliance was constructed and allows its future configuration. Unfortunately, their technique is strongly dependent on several tools (e.g., `libvirt`, `rpm`, `puppet`, `cft`) undermining its universal applicability.

In [23], Wilson discusses the *Conary* virtual appliance constructor tool that provides a versioned appliance repository supporting image creation and update. Conary requires the entire source code of the virtual appliance to be version controlled to allow custom configuration, creation (using rBuilder [13]) and update of virtual appliances. To manage already deployed appliances Conary must be co-located with appliances conforming to the WBEM or CIM standards [24], [25]. However, this last requirement increases the deployment cost of these new manageable VAs on commercial IaaS systems.

Articles [26], [27] investigate the appliance delivery method called *copy-on-read/write*. This method defers delivery by dispatching only those appliance parts that are used by VMs. Thus, a virtual machine starts as if a complete appliance is at hand. Attempting to use not-yet-delivered VA parts blocks the virtual machine while the needed parts arrive. Hence, this approach trades the initial processing power of new virtual machines to significantly reduce their apparent startup time. Unfortunately, this method is not applicable in highly-dynamic systems (our main focus) where VMs have short lifetime.

3 Conceptual Backround

The following section conceptualizes the behavior of current delivery optimization techniques. First, Section 3.1 presents a generic usage scenario of delivery optimization systems from both the viewpoint of the developer and the user. Next, Section 3.2 introduces an abstract model and

Authorized licensed use limited to: Klarquist Sparkman LLP. Downloaded on November 20,2025 at 21:18:46 UTC from IEEE Xplore. Restrictions apply.

notation for delivery optimization systems and their interactions. This model is restricted to focus on the aspects relevant to the main contribution of the article: the use of minimal manageable virtual appliances.

3.1 Deployment with Delivery Optimization

3.1.1 The Perspective of the Appliance Developer

Appliance developers create an *initial virtual appliance* if they expect repeated need for a special purpose virtual machine (e.g., a VM with certain external interfaces). With this appliance, special purpose VMs could be instantiated with minimal effort. In highly-dynamic systems [28], this requires appliances that result in VMs with minimal customization tasks after instantiation. This requirement necessitates an initial virtual appliance that encapsulates the required applications, services and their configurations—the *key functionality*—for the VM's special purpose. In summary, the created appliance should demand less expertise from appliance users while they operate the key functionality.

As an example, let us suppose that general purpose VMs repeatedly need manual customization with a content management system (CMS) that uses an external database. To avoid the repeated need for customization, an appliance developer creates a VA that offers the CMS with the proper configuration. Therefore, the key functionality of this appliance will be the newly offered CMS. Concerning customization, the new appliance should be prepared to accept the contact details of the database on startup—thus allow its contextualization [29].

The just created initial appliance is stored in a *self-contained* package (p_Ω) that is suitable for instantiating a virtual machine in an IaaS system. E.g., these packages are Amazon Machine Images if the developer uses Amazon EC2. For the rest of the article the term self-contained package and virtual appliance is used interchangeably. After the initial appliance is ready, developers estimate its future usage and deployment frequency. Thus, they can determine if the cost of applying delivery time optimization techniques would be regained during future deployments.

When opting for optimization, developers pass the initial appliance as an input for the chosen delivery time optimization technique. In return, they receive one or more packages ready for public use. The received packages still represent the key functionality of the initial virtual appliance, however, they are now constructed to better suit appliance delivery. Finally, developers publish the received packages in repositories or marketplaces (e.g., Amazon S3 [30], VMware marketplace [31] for commercial or Cumulus [32] for private clouds).

3.1.2 The Perspective of the User

Now that appliances are ready, users select one with the key functionality that fits their purpose. Instead of directly contacting an IaaS system to instantiate the selected appliance, users use a *deployment client* that arranges the delivery of the appliance. This deployment client first determines if the appliance is available in multiple packages. If so, then the packages are first composed (or *rebuilt*) to form an appliance ready for execution. Because of otherwise unavoidable performance issues, the rebuilding process should take place on the *deployment host* (the physical machine that will host the future virtual machine). The deployment client proceeds with the rebuilding in three phases. First, it downloads the packages necessary for the appliance to the deployment host. Then, it rebuilds the contents of the downloaded packages so they form a self-contained package again. Finally, the IaaS is instructed to create a VM based on the rebuilt appliance.

Hence, the issue of current delivery time optimization techniques is revealed: deployment clients need direct access to the deployment host. This prevents widespread adoption of these optimization techniques because most IaaS systems do not allow access to physical machines. This article proposes that by adding management capabilities to initial virtual appliances, developers can apply delivery time optimization techniques without intruding IaaS systems. These new capabilities change the way deployment clients should behave in case rebuilding is needed. Instead of the rebuilding procedure, they should start with the creation of the VM. Then proceed to rebuilding within the virtual machine itself by using its management capabilities.

The rest of the article presents the challenges and solutions on how these management capabilities can be added with modest efforts for developers and with minor effects on initial appliances.

3.2 Basic System Definitions

Delivery time optimization is applicable to numerous infrastructures. This article represents a particular infrastructure (φ) with its interacting hosts ($\varphi := \{h_1, h_2, \ldots\}$). Hosts are networked entities and their connections are quantitatively described with network latency and bandwidth. We have identified the following five host types participating in the appliance delivery scenario of Section 3.1:

1. IaaS services,
2. service users,
3. deployment clients,
4. repositories,
5. and deployment hosts.

First, *IaaS services* ($C := \{c_1, c_2, \ldots\}$ where $c_x \in \varphi$ is an individual IaaS service—e.g., the cloud controller service of Eucalyptus [8]) supervise a set of virtualization-enabled hosts. Over the supervised hosts, these services offer virtual machine or even virtual infrastructure management capabilities for their users.

Users host their services in the IaaS provided virtual machines. If the hosted services exhibit under or over provisioning, then users initiate the deployment or decommission of service instances with the *deployment client*. While arranging the delivery of the service, deployment clients look for virtual appliances—encapsulating the service's *key functionality*—in *repositories* ($R := \{r_1, r_2, \ldots\}$ where $R \subset \varphi$ is the set of repositories).

Repositories store virtual appliances in one or more packages ($p \in P$), where P represents all available packages in the system. The set of packages stored in a repository is described with the function $contents : R \rightarrow \wp(P)$. After

Authorized licensed use limited to: Klarquist Sparkman LLP. Downloaded on November 20,2025 at 21:18:46 UTC from IEEE Xplore. Restrictions apply.

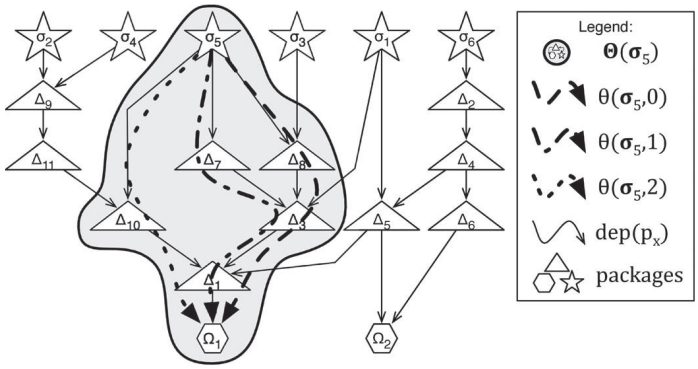

Fig. 1. Hypothetical dependency graph of six services.

identifying the package with the key functionality (p_σ), the deployment client instantiates it in a virtual machine. This is represented with the $initVM : P \rightarrow (\varphi \cup h^*)$ function that returns either a $vm \in \varphi$ or the non-existent host (h^*) if the VM could not be initiated.

Packages that represent key functionalities are accompanied with validator algorithms. These algorithms can semantically and functionally evaluate if a particular host offers the key functionality. This article models these algorithms by the function $valid : P \times \varphi \rightarrow \{true, false\}$. As appliance developers precisely know the key functionality, we assume that they accompany new appliances with appropriate validator algorithms.

Package content is derived from items ($i \in I$ where I symbolizes all possible items in the system). Items are the smallest individually handled entities (e.g., packages are built from files, or even from software packages—like rpm or dpkg) in an appliance. The algorithms introduced later are independent from the actual types of the items. The function $it : P \rightarrow \wp(I)$ defines the set of items that form a particular package. This article depicts items by their hashes and their storage size ($itsize : I \rightarrow \mathbb{N}$).

Finally, we estimate the storage size ($size : \wp(P) \rightarrow \mathbb{N}$) of a package set as the cumulative storage size of all items in the set members:

$$size(P_s) := \sum_{p \in P_s} \sum_{i \in it(p)} itsize(i) \tag{1}$$

where P_s is an arbitrary subset of all packages ($P_s \subset P$). This equation is used to estimate virtual appliance storage size by populating set P_s with packages embodying the complete appliance.

Self-contained packages are optimally sized if no items are removable without the loss of the key functionality:

$$optisize(p) := \begin{cases} true & \text{if } \nexists i \in it(p) : (it(p_x) = it(p) \backslash i \wedge \\ & \wedge valid(p, initVM(p_x) = true) \\ false & \text{otherwise.} \end{cases} \tag{2}$$

3.2.1 Definitions for Transfer Optimization Techniques

Transfer optimization techniques aim to arrange the contents of the repositories so packages can be served with shorter delivery times. However, self-contained packages, which encapsulate complete virtual appliances, do not allow fine grained repository content arrangement. Consequently, these techniques must split virtual appliances into smaller parts (e.g., items or arbitrary groups of items). The items of each part are put into a package and are marked to which particular appliance they belong to. Splitting enables the arrangement operation to focus on the most commonly used packages [11], [19]. E.g., these techniques are likely to form smaller packages using items found in multiple VAs. Consequently, after transfer optimization, the widely used packages are offered by the most repositories. In contrast, rarely used or unique packages usually found in a single repository.

However, split appliances are unusable for VM instantiation because they are stored in multiple packages. Before instantiation, these packages should be rebuilt to form an appliance again. This article proposes the set of direct package dependencies—$dep : P \rightarrow \wp(P)$—as a way to identify packages for an appliance. Through these sets, the package composition rule identifies the packages that can be composed together. Thus, we define the rule as:

$$\begin{aligned} p_3 &:= p_1 + p_2 \text{ where } p_2 \in dep(p_1) \\ &\text{therefore} \\ it(p_3) &:= it(p_1) \cup it(p_2) \\ dep(p_3) &:= dep(p_2). \end{aligned} \tag{3}$$

In this equation, p_1 is the dependent package and p_2 is an arbitrary choice from the direct package dependency set $dep(p_1)$. The equation reveals that we can define the composed package (p_3) through its items ($it(p_3)$) and direct package dependencies ($dep(p_3)$).

The composition rule can be applied repeatedly with the composed packages and their direct package dependencies until the last composed package (p'_3) has no further direct dependencies: $dep(p'_3) = \varnothing$. Later, we refer to this process of repeated composition as rebuilding. Using the example of Fig. 1, we observe that repeating composition of package σ_5 will have the following effect:

$$\begin{aligned} dep(\sigma_5) = \{\Delta_7, \Delta_8, \Delta_{10}\} &\rightarrow \sigma'_5 = \sigma_5 + \Delta_{10} \\ dep(\sigma'_5) = \{\Delta_1\} &\rightarrow \sigma''_5 = \sigma'_5 + \Delta_1 \\ dep(\sigma''_5) = \{\Omega_1\} &\rightarrow \Omega_5 = \sigma''_5 + \Omega_1. \end{aligned}$$

Therefore, rebuilding σ_5 (through packages $\Delta_{10}, \Delta_1, \Omega_1$) will result in the self-contained package of Ω_5.

It can be observed that σ'_5 could have been rebuilt differently because the direct package dependency set of σ_5 contains multiple packages. This article uses the term *construction path* to denote the set of packages that were chosen during the rebuilding process. Thus, a particular construction path of package p_x is denoted with the function ($\theta : P \times \mathbb{N} \rightarrow \wp(P)$). This function returns with a set of packages needed to rebuild an appliance based on p_x. The behavior of transfer optimization techniques ensures that rebuilding two arbitrary construction paths of the same package will result in the exact same VA. We define the set of all possible construction paths ($\Theta : P \rightarrow \wp(\wp(P))$) for a given package so later operations can select a specific construction path.

Authorized licensed use limited to: Klarquist Sparkman LLP. Downloaded on November 20,2025 at 21:18:46 UTC from IEEE Xplore. Restrictions apply.

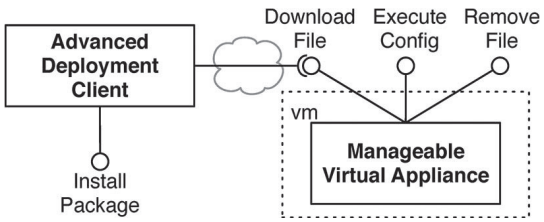

Fig. 2. Basic management interfaces.

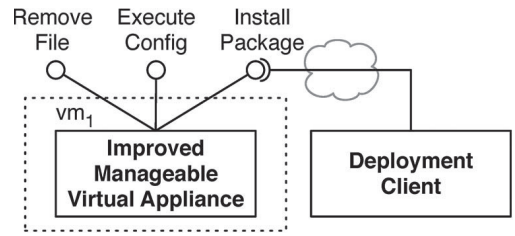

Fig. 3. Improved management interfaces.

Fig. 1 exemplifies the three possible construction paths of package σ_5 that can be formulated as follows:

$$\Theta(\sigma_5) = \left\{ \overbrace{\{\sigma_5, \Delta_8, \Delta_3, \Delta_1, \Omega_1\}}^{\theta(\sigma_5,0)}, \overbrace{\{\sigma_5, \Delta_7, \Delta_3, \Delta_1, \Omega_1\}}^{\theta(\sigma_5,1)}, \underbrace{\{\sigma_5, \Delta_{10}, \Delta_1, \Omega_1\}}_{\theta(\sigma_5,2)} \right\}.$$

4 The Minimal Manageable VA

Section 3.1 has shown that current practices in delivery time optimization need direct access to the deployment host to alter appliances before their instantiation. In contrast, we propose appliances capable to alter themselves at runtime. This approach delivers new appliance contents right to the VM (which also resides on the deployment host). To take advantage of this new approach, delivery time optimization techniques need to change.

To reduce the necessary changes in past solutions, we have analyzed their behavior. A minimal set of operations was isolated to support their operation: 1) installing a package from a repository—enables package composition—, 2) configuring the virtual machine—required after adding a new package—, and 3) erasing items from instantiated virtual machines—allows VM reuse or repurposing. Consequently, we define *manageable virtual appliances* (MVA) as special appliances encapsulating both the key functionality and at least the three management operations listed above.

Appliance instantiation time mostly depends on VA size, therefore, manageable virtual appliances imply longer instantiation times compared to appliances containing the key functionality only. To avoid unnecessary delivery time increase, the introduced management operations should have minimal impact on MVA size. I.e., more complex management interfaces, which support advanced deployment scenarios, can be tolerated on larger appliances (e.g., over a few gigabytes in size). However, smaller appliances would experience significant size (and thus, deployment time) impact. In this article, we pursue a more widely applicable solution: more complex functionalities were given up in exchange for smaller sizes.

Therefore, we first investigate the *Basic management interfaces* that offer the simplest implementation of the minimal operations. As depicted in Fig. 2, these interfaces allow the following operations: 1) *downloading* a single file from a URL to a designated location in the virtual machine, 2) *executing* a configuration script in the VM and 3) *removing* a file from the VM.

Unfortunately, the ''*Download File*'' operation requires files as its inputs although repositories store packages. Thus, repository packages should be converted to a form acceptable by basic management interfaces. We refer to those deployment clients that support this conversion as ''*advanced deployment clients*''. Although, these clients still receive appliance instantiation tasks, they transform the received tasks for basic interfaces. They download and itemize the necessary packages on their host. Then, they upload the individual items to the target VM with its ''*Download File*'' operation. Hence, advanced deployment clients need to transfer the necessary packages twice—first between the repository and the host of the deployment client, then between the client and the execution host of the manageable appliance. Thus, even though basic management interfaces could be implemented with relatively small size, they require changes in existing deployment clients and cause unnecessary network traffic.

To avoid these disadvantages, we introduce the *Improved management interfaces* that are designed to allow the simplest implementation of manageable virtual appliances while still enable using regular deployment clients (see Fig. 3). To avoid the bottleneck of basic management interfaces, the improved ones replace the previously identified ''*Download File*'' operation with the ''*InstallPackage*'' operation. This operation can be called on the VM of an instantiated MVA. As a result, the VM contacts a repository and downloads the requested package. After itemizing the received package, the VM places the items of the package on their intended locations.

Comparing the two interfaces, it is observable that improved management interfaces download appliances only once. Also, improved interfaces do not need advanced clients. However, because of their ''*InstallPackage*'' operation, they notably impact manageable appliance size. In the next section, we propose an approach to overcome this consequence. This approach will let this article to consider only the use of improved management interfaces.

4.1 Definition and Use

To enable the exploitation of MVAs, appliance developers should prepare their appliances with embedded management interfaces. This article suggests that delivery optimization systems could support developers by providing appliance template packages. These templates should include management interfaces only so they could form the base of developer created MVAs. Also, these templates should be minimally sized so appliances based on them will have negligible size increase. Templates that fulfill these requirements will allow MVAs to mainly contain their key functionality. Consequently, we call these

Authorized licensed use limited to: Klarquist Sparkman LLP. Downloaded on November 20,2025 at 21:18:46 UTC from IEEE Xplore. Restrictions apply.

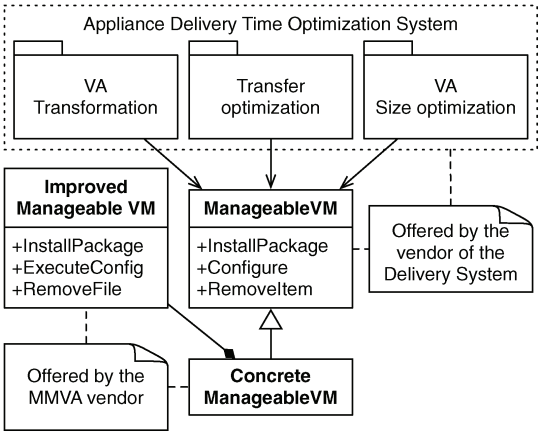

Fig. 4. MMVA interfacing with delivery systems.

templates as ''*Minimal Manageable Virtual Appliances*'' (or MMVAs—p_μ).

We define MMVAs as optimally sized self-contained packages (see (2)) with the management interfaces as their key functionality. Formally:

$$mmva(p) := (manageable(p) = true) \wedge (optisize(p) = true). \quad (4)$$

Where the function $manageable : P \rightarrow \{true, false\}$ evaluates to *true* if the given package represents an MVA and to *false* otherwise. This function allows delivery time optimization techniques to determine if they can exploit management capabilities. Next, we present an approach to algorithmically assess this function.

This algorithm tries to use the improved management interfaces on the appliance under evaluation. If the interfaces operate according to our previous definitions then the algorithm qualifies the appliance as an MVA. Algorithm 1 presents our approach to test manageability and offers an example resolution for the function *manageable*. The following paragraphs detail the behavior of the algorithm.

Algorithm 1 Validator for the management capabilities.

Require: $p_\sigma \in P$ // package to be tested for manageability
1: $vm_e \leftarrow initVM(p_\sigma)$
2: $P_{refs} \leftarrow \{p \in P : (\exists n, m : (\theta(p,n) \setminus \{p\}) \subset \theta(p_\sigma, m)) \wedge (valid(p, initVM(p)) = true) \wedge (valid(p, vm_e) = false)\}$
3: $p_{ref} \leftarrow p \in P_{refs} : (size(\theta(p,0)) = \min_{p_x \in P_{refs}} size(\theta(p_x, 0)))$
4: **for all** $p \in \theta(p_{ref}, 0)$ **do**
5: $\quad installPackage(vm_e, p)$
6: $\quad executeConfig(vm_e, p)$
7: **end for**
8: $before \leftarrow valid(p_{ref}, vm_e) = true$
9: **for all** $i \in it(p_{ref}) : (type(i) = \text{``}file\text{''})$ **do**
10: $\quad removeFile(vm_e, i)$
11: **end for**
12: $after \leftarrow valid(p_{ref}, vm_e) = true$
13: **return** $manageable(p_\sigma) \leftarrow before \wedge \neg after$

First, our algorithm receives a virtual appliance to be evaluated as an input in the form of a package with a key functionality (p_σ). Afterwards, in line 1, the algorithm creates an evaluator VM (vm_e) for the received appliance. Next, line 2 selects those packages that can be added in the appliance of p_σ. These packages mostly share their construction path with p_σ. The selected packages are restricted to those that offer a key functionality not present in the evaluator VM (ensured by two validity checks in line 2). Adding one of the packages of P_{ref} to the evaluator virtual machine will result in a multi functional VM. Afterwards, in line 3, the algorithm picks a reference package (p_{ref}) that is the smallest among the selected packages. By doing so, it minimizes the execution time of the rest of the algorithm.

After the preparation phase, between lines 4 and 7, the algorithm installs and configures p_{ref} in the evaluator VM (vm_e) using the assumed management interfaces of p_σ. If the management interfaces are available, then the vm_e passes the validation in line 8. Next, the items of p_{ref} are removed from vm_e to test its *removeFile* operation (see line 10). Consequently, vm_e should not pass the validation in line 12. Finally, the algorithm accepts the management capabilities of p_σ, if the first validation of vm_e succeeds and the last fails.

We have identified two major uses for this algorithm: 1) to create MMVAs and 2) to guarantee manageability. First, to *create MMVAs* one can use a virtual appliance size optimization technique [13], [14], [33]. These techniques can minimize an MVA until it solely offers management operations. The techniques should use Algorithm 1 to ensure management functionalities remain intact during the optimization process.

Second, future delivery time optimization techniques could depend on MVAs. However, delivery time optimization might imply the modification of the MVA. But, these modifications should not cause the loss of management capabilities. To *guarantee manageability*, these techniques should evaluate their intended changes with our algorithm.

4.1.1 MMVAs in Practice

The beginning of Section 4 has shown the possibility to define multiple management interfaces. Similarly, multiple kinds of minimal manageable virtual appliances are also possible. E.g., a new MMVA is introduced for every new package format to be supported by the ''*InstallPackage*'' function. For faster adoption, delivery optimization systems should offer a common interface with the VMs instantiated from the plethora of MMVAs.

Fig. 4 shows an example for this common interface with the ''*ManageableVM*''. This interface represents the necessary management capabilities and behaves as the intermediary between the delivery system and the various MMVAs. Consequently, the provider of a new MMVA must implement this common interface exemplified by the ''*Concrete ManageableVM*''. This implementation should translate the requests of the delivery optimization system to a form understood by the particular MMVA. In the particular example of Fig. 4, calls to the ''*RemoveItem*'' operation result in ''*RemoveFile*'' calls of the ''*Improved ManageableVM*''. This translation is especially important if the particular delivery optimization system uses different kinds of items than files.

Authorized licensed use limited to: Klarquist Sparkman LLP. Downloaded on November 20,2025 at 21:18:46 UTC from IEEE Xplore. Restrictions apply.

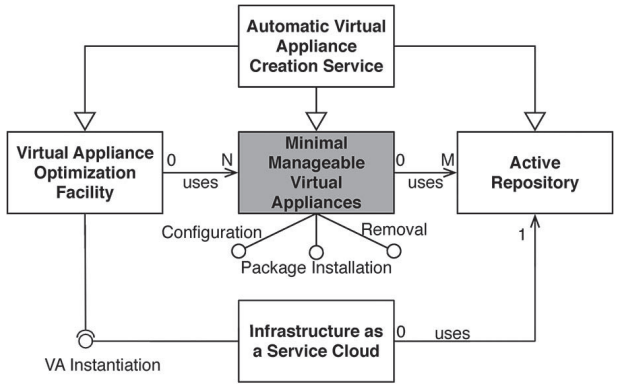

Fig. 5. Automated Virtual appliance creation Service.

4.2 Creating Virtual Appliances Based on MMVAs

Before creating a MMVA based virtual appliance, developers design a self-contained package (p^*) for their hypothetical initial appliance. This package should represent all the items for the intended key functionality and its support environment without management interfaces. If the package design is ready then developers can start creating the MMVA based virtual appliance. We have identified two basic scenarios for appliance creation. First, appliance developers could *extend an MMVA* by utilizing it as the foundation for the key functionality. Alternatively, they could *append* the items of the MMVA to their hypothetical appliance.

When several MMVAs are available, developers should select the most suitable MMVA before appliance creation. To determine their suitability, available MMVAs must be investigated based on their functionality and contents. Developers should collect the MMVAs that best support the delivery optimization tasks to be applied. These MMVAs are equivalent from the developer point of view. We refer to the set of these MMVAs as P_M. This set can be automatically filtered by analyzing their items with the function: $suits : \wp(I) \rightarrow P_M$. To reduce the impact on the size of the future appliance, developers should choose the MMVA that shares the most items with the hypothetical appliance:

$$suits(\mathcal{I}^*) := p_\mu \in P_M : \left(\left| it(p_\mu) \cap \mathcal{I}^* \right| = \max_{p_x \in P_M} \left| it(p_x) \cap \mathcal{I}^* \right| \right) \quad (5)$$

where $\mathcal{I}^*$ depicts the items of the hypothetical initial VA. Using the $suits$ function the developers can now proceed to appliance creation.

First, we analyze the *MMVA extension* scenario. Initially, developers instantiate a virtual machine with the most suitable MMVA (p_μ). During instantiation, they ensure that the new virtual machine can host both the key functionality and the MMVA. E.g., they request a disk for the new VM with a size bigger than $size(\{p^*, p_\mu\})$. Afterwards, they install the key functionality inside the newly created virtual machine just as they would regularly do. Finally, they shut down the VM and export its disk image as the initial virtual appliance (p_Ω).

In the second scenario (referred as ''*append*''), developers prepare their system—with the key functionality—as they prefer. As a first step towards appliance creation, they download and itemize the most suitable MMVA. Hence, they receive the MMVA's items ($it(p_\mu)$) that they append to their prepared system and proceed with the initial appliance creation: $it(p_\Omega) := it(p^*) \cup it(p_\mu)$.

In both scenarios, appliance developers have to choose an MMVA that remains functional alongside the key functionality. To avoid unexpected behavior, the system should not allow publishing appliances with broken management capabilities (where $manageable(p_\Omega)$ evaluates as *false*). In such cases the developer must repeat the MMVA embedding procedure.

5 Inside a Delivery Optimization System

In the following Sections, we demonstrate the effects of Minimal Manageable Virtual appliances on delivery time optimization systems. For the demonstration, we have selected the *Automated Virtual appliance creation Service* (AVS—[19]) that we adopted to utilize the MMVA concept. The AVS service (see Fig. 5) offers basic virtual appliance creation and management capabilities including virtual appliance creation, virtual machine image transformation and initial virtual appliance upload. In [19], the AVS service was introduced with two major components: *active repositories* and the *optimization facility*. Active repositories modify their contents according to the demand patterns of the stored packages to optimize their future delivery. These repositories use the following operations on packages:

1. decompose,
2. merge,
3. replicate,
4. and destruct.

The size optimization facility [33] uses active fault injection to remove virtual appliance contents until only the key functionality and its minimal support environment remains.

The following sections demonstrate, analyze and evaluate how the AVS service exploits minimal manageable virtual appliances. Before utilizing the MMVA concept, the AVS service has had two major bottlenecks: 1) the extensive rebuilding time preceding the deployment of decomposed appliances and 2) the often lengthy virtual appliance optimization time. With the help of MVAs the AVS optimizes the use of active repositories by moving the rebuilding of decomposed virtual appliances to the execution host. Also, the AVS uses the management capabilities of MVAs to significantly decrease virtual appliance optimization time.

5.1 Effects on the Rebuilding Algorithms

As a consequence of package decomposition ($p \rightarrow \{p_1, p_\Omega\}$) in active repositories, the AVS service incorporates techniques for virtual appliance rebuilding to apply the package composition rule on the requested package (p_{req}) until the resulting package becomes self-contained (it is ready to be instantiated as a virtual machine). These techniques can be embedded in three different components of the system: 1) in the *active repository*, 2) in the *IaaS system* and 3) in the *deployment client*. These techniques are generally composed of the following four basic stages:

Authorized licensed use limited to: Klarquist Sparkman LLP. Downloaded on November 20,2025 at 21:18:46 UTC from IEEE Xplore. Restrictions apply.

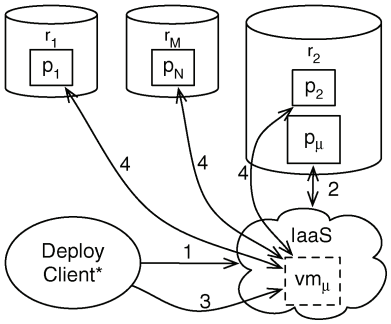

Fig. 6. Deployment client using an MMVA for rebuilding.

1. identification of the possible construction paths—$\Theta(p_{req})$—,
2. selection of the optimal construction path identifier—$optp : P \times \varphi \rightarrow \mathbb{N}$—,
3. downloading the required packages,
4. and rebuilding of the virtual appliance on the deployment host.

This article considers only deployment client based solutions because others would require significant changes in IaaS systems. When the client embeds the rebuilding algorithm, first, it downloads all required packages to its host to rebuild them. Then, it publishes the rebuilt appliance (p_{reb}) in the repository with the highest bandwidth connection towards the deployment host (h_{dpl}). Afterwards, it requests the deployment system to initiate a VM with the rebuilt appliance. Finally, the IaaS system downloads the rebuilt VA and instantiates the VM. The overhead of several extra transfers would make this rebuilding algorithm ineffective. However, virtual appliances based on MMVAs ($\exists n < |\Theta(p_{req})| : (\exists p_x \in \theta(p_{req}, n) : (mmva(p_x) = true))$) allow deployment clients to instantiate them with a new approach.

5.1.1 New Rebuilding Technique

As shown in Fig. 6, the deployment client requests the IaaS system to initiate the MMVA in a virtual machine suitable for the virtual appliance of p_{req} (see *step 1*). In *step 2*, the IaaS system downloads the MMVA from repository r_2 and creates a virtual machine using the MMVA ($vm_\mu \leftarrow initVM(p_\mu)$). In *step 3*, the deployment client uses the management interfaces on the vm_μ to download the packages according to the optimal construction path of the requested package. In *step 4*, the MMVA installs and configures the requested package as commanded by the deployment client. Finally, the deployment client restarts the VM to activate the rebuilt service. The final step ensures the application of the newly added packages and configuration.

5.1.2 Identifying the Optimal Construction Path

Steps 3-4 construct virtual appliances by selecting and composing the construction path with the lowest rebuilding time. To identify this construction path, first, we define the individual localized package rebuilding time ($t_{reb} : P \times R \times \varphi \rightarrow \mathbb{R}$) based on two basic components: 1) the *transfer time*—$t_{tr} : P \times R \times \varphi \rightarrow \mathbb{R}$—of the package to the location of rebuilding, and 2) the *package composition time*—$t_{comp} : P \times \varphi \rightarrow \mathbb{R}$—of the transferred package and its dependencies. Formally:

$$t_{tr}(p, r, h) := \begin{cases} l(r, h) + size(\{p\})/BW(r, h) & \text{if } r \neq h \\ 0 & \text{otherwise} \end{cases}$$

$$t_{comp}(p, h) := size(\{p\})/BW(h, h)$$

$$t_{reb}(p, r, h) := t_{tr}(p, r, h) + t_{comp}(p, h). \quad (6)$$

Thus, we model *transfer time* so it mainly depends on the network latency ($l : \varphi^2 \rightarrow \mathbb{R}$) and bandwidth ($BW : \varphi^2 \rightarrow \mathbb{R}$) between the repository and the transfer target host. Also, we model the *composition time* as making a local copy of a package on the target host. The measure $BW(h, h)$ represents the local storage bandwidth measurement for in-host bandwidth estimation.

The repository with the highest bandwidth is picked to calculate the optimal rebuilding time ($t_{oir} : P \times \varphi \rightarrow \mathbb{R}$):

$$t_{oir}(p, h_{dpl}) := \min_{r \in R} t_{reb}(p, r, h_{dpl}). \quad (7)$$

The optimal rebuilding time is considered from the viewpoint of the deployment host (h_{dpl}). Using the values of the $t_{oir}(p, h_{dpl})$ function, the total rebuilding time on the deployment host is calculated with the following method ($T_{totreb} : P \times \mathbb{N} \times R \rightarrow \mathbb{R}$):

$$T_{totreb}(p, n, h_{dpl}) := \sum_{p_x \in \theta(p,n)} t_{oir}(p, h_{dpl}). \quad (8)$$

Finally, on a given host, we define the optimal construction path identifier of a package as follows:

$$optp(p, h) := \nu < |\Theta(p)| : (\forall m < |\Theta(p)| : (T_{totreb}(p, \nu, h) \leq T_{totreb}(p, m, h))). \quad (9)$$

Therefore $optp(p, h)$ will point towards the construction path with the smallest total rebuilding time: $\theta(p, \nu)$.

5.1.3 Estimating Connectivity for New Virtual Machines

The active repository and IaaS system based rebuilding techniques store the connectivity history (previous latency and bandwidth values between repositories, IaaS systems and various hosts in the system). Those techniques exploit historical values to determine the optimal construction path. However, in case of deployment clients, this approach cannot be followed, because the connectivity history is unknown for new VMs.

Therefore, the deployment client estimates the connectivity data using the management interfaces of the new VM (see Algorithm 2). The behavior of the client is customizable in the algorithm with the following predefined constants:

1. the maximum acceptable latency (*timeout*),
2. the maximum amount of data used for bandwidth measurements (expressed in the ratio of total appliance size, later referred as *pre-transfer measurement threshold*—τ),

Authorized licensed use limited to: Klarquist Sparkman LLP. Downloaded on November 20,2025 at 21:18:46 UTC from IEEE Xplore. Restrictions apply.

TABLE 1
Comparison of the Introduced Rebuilding Scenarios

	IaaS System	Active Repository	Deployment client no MMVA	Deployment client MMVA
Measurements	prior	prior	during	during
No IaaS change	–	√	√	√
No repo. change	√	–	√	√
Arbitrary VAs	√	√	√	–
Transfer size ($\cdot size(\theta(p_{req}, n))$)	1	$1 + E$	$3 + \tau$	$1 + \tau + M$

Algorithm 2 Estimating connectivity details between the repositories and the virtual machine of an MMVA.

Require: *timeout*, $0 < \tau < 1$, *awaitedbw*, *mxtsize*
Require: $vm \in \varphi$
Require: $p_{req} \in P$
Require: $p_\mu \in P : (mmva(p_\mu) = true \wedge (\exists n < |\Theta(p_{req})| : (p_\mu \in \theta(p_{req}, n))))$ // An MMVA on which p_{req} depends
1: **for all** $r \in R : (\exists \theta(p_{req}, x) \in \Theta(p_{req}) : (contents(r) \cap \theta(p_{req}, x) \neq \emptyset))$ **do**
2: $l(r, vm) \leftarrow measure(l(r, vm), timeout)$
3: $BW(r, vm) \leftarrow 0$
4: **end for**
5: $transfersize \leftarrow size(\theta(p_{req}, 0)) - size(\{p_\mu\})$
6: $uptimeout \leftarrow transfersize/awaitedbw$
7: $R' \leftarrow R$
8: $bwmeasures \leftarrow \tau \frac{transfersize}{mxtsize}$
9: **for** $i = 0$ to *bwmeasures* **do**
10: $r_{ml} \leftarrow r \in R' : (l(r, vm) = \min_{r_y \in R'} l(r_y, vm))$
11: $R' \leftarrow R' \setminus r_{ml}$
12: $P_m \leftarrow \{\forall p \in contents(r_{ml}) : (size(\{p\}) < mxtsize)\}$
13: $p_{dummy} \leftarrow p \in P_m : (size(\{p\}) = \max_{p_x \in P_m} size(\{p_x\}))$
14: $BW(r_{ml}, vm) \leftarrow measure(BW(p_{dummy}, r_{ml}), uptimeout)$
15: **end for**

3. the minimal bandwidth between the repository and the virtual machine (*awaitedbw*),
4. and finally, the maximum size of the package used for bandwidth estimation (*mxtsize*).

In its first line, this algorithm identifies those repositories that contain a package from one of the construction paths of package p_{req}. Then, the deployment client measures the latencies of the repositories—$l(r, vm)$, see line 2. Afterwards, in line 8, it estimates the number of bandwidth measurements (*bwmeasures*) the algorithm can make before reaching the pre-transfer measurement threshold. Then, in lines 10-14, selects the repositories ($r_{ml} \in (R \setminus R')$) with the lowest latencies to measure their bandwidth towards the host of the MMVA. By taking these measurements, the deployment client can estimate the optimal construction path.

5.1.4 Comparison of the Rebuilding Techniques

Table 1 compares the previously detailed rebuilding scenarios from five points of view:

1. ''*Measurements*'' denote if the system takes the measurements independently from the rebuilding process;
2. ''*No IaaS change*'' depicts the approaches that do not require changes in the available IaaS systems for their operation; similarly
3. ''*No repository change*'' describes the solutions independent of the repository implementation;
4. ''*Arbitrary VAs*'' present if an algorithm is not dependent on a specific type of virtual appliance—e.g., they are not dependent on the management capabilities;
5. finally, ''*Transfer size*'' represent the amount of data (expressed relative to appliance size: $reltsize := tsize/size(\theta(p_{req}, 0))$) required to reconstruct the VA on the execution host.

Analyzing the table reveals that the IaaS and active repository based solutions can take *measurements* prior to the actual rebuilding process and do not require extra transfers to determine the latency and bandwidth values. These systems can measure and monitor the connection properties of previous transfers between the different hosts of the system. As an opposite, deployment client based solutions use extra transfers to estimate the connection properties just before the rebuilding takes place. However, Algorithm 2 limits these extra transfers using the pre-transfer measurement threshold value (τ).

The last row of Table 1 compares the total size of transfers. The row unveils the superiority of the IaaS based solution: appliances can be transferred directly to the execution host. Active repositories are less efficient because first they transfer external packages (E) to the repository of the requested package then, after rebuilding, they send the rebuilt appliance (p_{reb}) to the IaaS.

First, deployment clients estimate connection properties (as the pre-transfer measurement threshold—τ—allows). Then they download all packages for rebuilding on the client's host (the first time the entire appliance is transferred). Later, the rebuilt appliance is published (second transfer from the client to a repository). Finally, the IaaS is tasked with the instantiation of the published VA, resulting the final transfer during deployment. Thus, we calculate the required transfer size for deployment clients ($tsize_{DC}, reltsize_{DC}$) as:

$$tsize_{DC} := \overbrace{\tau \cdot size\big(\theta(p_{req}, n)\big)}^{\text{measure}} + \overbrace{\sum_{p \in \theta(p_{req}, n)} size(\{p\})}^{\text{rebuild}} + \underbrace{size\big(\theta(p_{req}, n)\big)}_{\text{publish}} + \underbrace{size\big(\theta(p_{req}, n)\big)}_{\text{deploy}} \tag{10}$$

$$reltsize_{DC} := \tau + 3. \tag{11}$$

In contrast, when deployment clients utilize MMVAs, first, they instantiate the MMVA (the entire MMVA is transferred). Then they estimate connection properties using the MMVAs management interfaces (see τ in the equation). Finally, after the client evaluates the optimal construction path, the MMVA is requested to install the packages along the path—$\theta(p_{req}, optp(p_{req}, vm))$. The size of the required transfers ($tsize_{VM}, reltsize_{VM}$) by the MMVA based rebuilding solution is defined as:

Authorized licensed use limited to: Klarquist Sparkman LLP. Downloaded on November 20,2025 at 21:18:46 UTC from IEEE Xplore. Restrictions apply.

$$tsize_{VM} := \overbrace{size(\{p_\mu\})}^{\text{init}} + \overbrace{\tau \cdot size(\theta(p_{req}, n))}^{\text{measure}} \underbrace{+ size(\theta(p_{req}, n)) - \sum_{i \in (it(p_\mu) \cap it(p^*))} itsize(i)}_{\text{reconstruct}} \tag{12}$$

$$reltsize_{VM} := 1 + \tau + M$$

where :

$$M := \frac{size(\{p_\mu\}) - \sum_{i \in (it(p_\mu) \cap it(p^*))} itsize(i)}{size(\theta(p_{req}, n))} \tag{13}$$

where M embodies the size of extra items added to the hypothetical initial virtual appliance (p^*) to offer management capabilities (see Section 4.2). If the MMVA builds on items only from p^* then M turns 0.

5.2 MMVA and the Optimization Facility

The AVS service offers a size optimization facility that implements the $optimize : P \rightarrow P$ function. This function processes the initial virtual appliance ($p_{orig} \in P$) and removes those items from its self-contained package that are not necessary for its key functionality. The result is an minimally sized self-contained package (p_{opt}). The properties and behavior of the optimization facility were detailed in [33]. This section focuses only on those parts that highlight the advantages and validate the use of MMVAs.

Algorithm 3 Simplified size optimization.

Require: $p_{orig} \in P$ // initial package
Require: $r \in R : (p_{orig} \in contents(r))$ // repository
1: $p_{subopt} \leftarrow \sum_{p \in \theta(p_{orig}, 1)} p$
2: $p_{opt} \leftarrow \varnothing$
3: **while** $it(p_{subopt}) \setminus it(p_{opt}) \neq \varnothing$ **do**
4: $i \in it(p_{subopt}) \setminus it(p_{opt})$ // Arbitrary item selection
5: $it(p_{cand}) \leftarrow it(p_{subopt}) \setminus \{i\}$
6: $contents(r) \leftarrow contents(r) \cup \{p_{cand}\}$
7: $vm \leftarrow initVM(p_{cand})$
8: **if** $(vm \in \varphi) \wedge (valid(p_{orig}, vm) = true)$ **then**
9: $p_{subopt} \leftarrow p_{cand}$
10: **else**
11: $it(p_{opt}) \leftarrow it(p_{opt}) \cup \{i\}$
12: **end if**
13: $contents(r) \leftarrow contents(r) \setminus \{p_{cand}\}$
14: **end while**
15: **return** $optimize_{base}(p_{orig}) \leftarrow p_{opt}$ // optimized package

5.2.1 The Simplified Algorithm

Algorithm 3 describes a simplified version of the optimization facility. This algorithm focuses on the repository and virtual machine handling operations, because these are the areas that will be changed after utilizing the MMVA functionality in the appliances. The algorithm starts with the creation of the candidate package (p_{cand}, see lines 4-5) by removing an arbitrary (but not yet evaluated) item from the suboptimal package. Then, to allow the creation of a VM based on the candidate package, it is uploaded to the repository of the target IaaS system (see line 6). Next, in line 7, the candidate appliance is instantiated in a virtual machine on the target IaaS system. In case of successful instantiation, a valid host ($vm \in \varphi$) is returned by *initVM*. This new virtual machine is evaluated with the validator algorithm of the initial appliance (see line 8). On successful validation, we consider the candidate as the new suboptimal package (p_{subopt} in line 9). Otherwise, item i must be part of the optimal package (see line 11). Finally, the algorithm reduces storage costs by cleaning up the temporarily published candidate package in line 13.

5.2.2 The Improved Algorithm

The analysis of Algorithm 3 reveals that its most expensive operations are 1) the publication of the candidate package—due to implicit transfer and storage costs—and 2) the creation of the suboptimal virtual machine—due to the involved time consuming steps. Algorithm 3 publishes and creates a VM for each item in the initial appliance. To improve the performance of the optimization facility, we introduce Algorithm 4 that is specifically aimed at reducing the average number of VM creations and package publications. The two key elements of the advanced algorithm are: 1) virtual machine reuse and 2) intermediate appliances. First, the algorithm reuses every VM, unless the candidate appliance inside is dysfunctional. This is achieved by using the management interfaces to create the candidate VA inside a VM. Second, only successfully evaluated candidate appliances are published (as *intermediate VAs*). To further reduce publication costs, the algorithm uses an intermediate appliance upload threshold ratio (ut—see line 11 in Algorithm 4) to publish only relevant candidate appliances: those that are significantly smaller than the latest intermediate appliance.

A detailed look at Algorithm 4 reveals that it keeps a virtual machine running with the actual suboptimal VA (see lines 5 and 17). Most IaaS systems do not allow VM creation from unpublished appliances (e.g., the suboptimal appliance). Thus, the algorithm uses the intermediate VA for the basis of the current VM. During the runtime of the algorithm, the suboptimal and intermediate appliances are rarely identical. Line 19 synchronizes the contents of the intermediate VA by removing those items that are not present in the suboptimal VA.

The MMVA enabled algorithm simplifies the way to reach the VM of the candidate VA: it utilizes the *removeFile* operation of current VM. As a result, the algorithm creates the candidate package only inside the current VM (see line 8). The now prepared VM is validated in line 9. Based on the validation results, the algorithm either creates a new suboptimal package (see line 10) or initiates a new VM instead of the unsuccessfully validated one (see line 17). The technique for VM creation was already discussed, thus we can now focus on the handling of the suboptimal package. When creating a new suboptimal package, the algorithm optionally updates the repository (see lines 11-14). First, the update procedure removes the previous intermediate appliance, then it publishes the current suboptimal package as the new intermediate virtual appliance (p_{im}). If the validation

Authorized licensed use limited to: Klarquist Sparkman LLP. Downloaded on November 20,2025 at 21:18:46 UTC from IEEE Xplore. Restrictions apply.

TABLE 2
Properties of the Experimental Appliances

	Compressed Size	Files	Deployment Time
MMVA (p_μ)	6.6 MiB	197	4.65 seconds
Initial VA (p_{Ap})	165 MiB	14050	36.4 seconds
Apache (p_σ)	13 MiB	236	7.78 seconds

fails in line 9, then intermediate appliances show their second advantage: compared to the initial appliance used by the simplified algorithm, intermediate VAs are instantiated considerably faster because of their size.

Algorithm 4 MMVA enabled size optimization.

Require: $p_{orig} \in P : (manageable(p_{orig}) = true)$
Require: $r \in R : (p_{orig} \in contents(r))$
Require: $ut \in \mathbb{R}$ // upload threshold
1: $p_{subopt} \leftarrow \sum_{p \in \theta(p_{orig},1)} p$
2: $contents(r) \leftarrow contents(r) \cup \{p_{subopt}\}$
3: $p_{opt} \leftarrow \emptyset$
4: $p_{im} \leftarrow p_{subopt}$
5: $vm \leftarrow initVM(p_{im})$
6: **while** $it(p_{subopt}) \setminus it(p_{opt}) \neq \emptyset$ **do**
7: $i \in it(p_{subopt}) \setminus it(p_{opt})$ // Arbitrary item selection
8: $removeFile(vm, i)$
9: **if** $valid(p_{orig}, vm) = true$ **then**
10: $it(p_{subopt}) \leftarrow it(p_{subopt}) \setminus \{i\}$
11: **if** $ut < \frac{(size(\{p_{im}\})}{size(\{p_{subopt}\}))}$ **then**
12: $contents(r) \leftarrow contents(r) \setminus \{p_{im}\}$
13: $p_{im} \leftarrow p_{subopt}$
14: $contents(r) \leftarrow contents(r) \cup \{p_{im}\}$
15: **end if**
16: **else**
17: $vm \leftarrow initVM(p_{im})$
18: **for all** $i_x \in it(p_{subopt}) \setminus it(p_{im})$ **do**
19: $removeFile(vm, i_x)$
20: **end for**
21: $it(p_{opt}) \leftarrow it(p_{opt}) \cup \{i\}$
22: **end if**
23: **end while**
24: **return** $optimize_{mmva}(p_{orig}) \leftarrow p_{opt}$

5.2.3 Comparison of the Two Algorithms

Section 5.2.2 identified the two main factors of potential performance degradation in Algorithm 3. To show the intrinsic differences between the simplified and the improved algorithms, we analyze their behavior during an optimization operation focusing on the two main factors only. First, we provide an estimate of the number of virtual machine creations during the optimization. Then, we give an upper bound for the total transferred data towards the repositories during the same period.

First, the simplified algorithm creates a virtual machine for all items in the initial appliance ($|it(p_{orig}|$). In contrast, the MMVA enabled algorithm reduces the number of VM creations to the number of items impossible to remove from the appliance ($|it(p_{opt}|$). Consequently, the MMVA enabled algorithm always performs better as $|it(p_{opt})| \leq |it(p_{orig})|$.

Next, we compare the cost (in terms of data volume transferred) of package publications in the repository. In order to support the widest range of IaaS systems, the simplified algorithm publishes a candidate package for every item in the appliance (see line 7 in Algorithm 3). Thus, the simplified algorithm transfers the following amount of data: $TotTransfer_S = size(\{p_{orig}\}) \cdot |it(p_{orig})|$). In comparison, the improved algorithm decreases the transfers towards the repository because it publishes appliances only if the following conditions are met: 1) the current appliance offers the key functionality—see line 9 of Algorithm 4—and 2) there is a significant reduction in virtual machine instantiation time—see line 14 of Algorithm 4. As a result, the transfer requirements of the improved algorithm decrease significantly: $TotTransfer_I < size(\{p_{orig}\}) \cdot |it(p_{orig}) \setminus it(p_{opt})|$.

6 Implementation and Results

We have implemented the MMVA extensions on the AVS service. This extended service could utilize multiple IaaS systems. We have conducted a series of experiments to analyze the effects of MMVAs on the AVS.

6.1 Implementation and Experimental Setup

First, we created a simple MVA based on a small *Debian Squeeze* installation extended with `ssh` and `rsync`. Through `ssh`, we allow running shell scripts generated by the AVS service or the deployment client. `Rsync` is used to provide reliable file transfer and removal operations over `ssh`. The *installPackage* operation of the *improved management interface* (see Section 4) is accomplished by uploading a package to the virtual machine of the MVA then running a pre-configuration script that parses package contents and places them to their designated locations. To create an MMVA, we minimized the size of our simple MVA with AVS's optimization facility. This size optimized MVA (p_μ) is detailed in Table 2.

Next, we have selected a widely used application (the Apache http server) as the key functionality of our experimental virtual appliance. The Apache http server is the base for several widely used applications thus, the effects of our system on its delivery also gives a solid estimation for other applications. To prepare the Apache appliance, we have *extended* our simple MVA (similarly to the MMVA extension scenario in Section 4.2). The added apache http server was configured to only offer static webpages to its clients. The just created Apache appliance (p_{Ap}) shares common roots with our MMVA (p_μ) because they are both based on our simple MVA. Consequently, the new appliance can be used to demonstrate the effects of the embedded MMVAs on the AVS (thus generally on delivery time optimization systems).

As a last step to prepare the experiments, we have deployed three IaaS systems (namely Nimbus, Eucalyptus and a proprietary solution) on the infrastructure of the University of Westminster. These IaaS systems did not coexist. We have executed an initial batch of experiments using each IaaS system. These experiments were all

Authorized licensed use limited to: Klarquist Sparkman LLP. Downloaded on November 20,2025 at 21:18:46 UTC from IEEE Xplore. Restrictions apply.

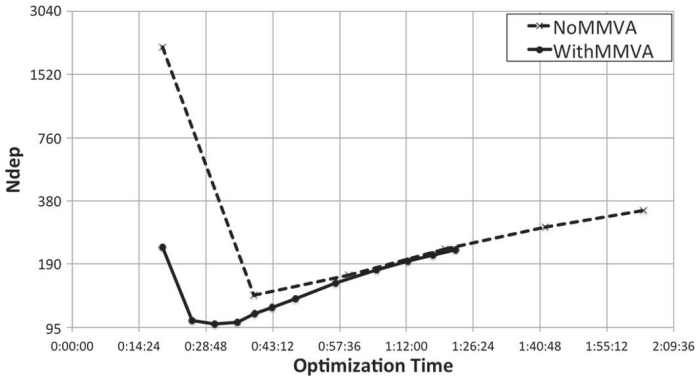

Fig. 7. Number of future deployments required.

focusing on the way the AVS behaved with and without utilizing the management capability embedded in our Apache appliance. The behavior of the AVS was independent from the actual IaaS system that managed the infrastructure of the university. Therefore, we have selected an IaaS system that was used in the later evaluations of the extended AVS service. The selected IaaS system was a proprietary solution—similar to Amazon EC2—that was designed to resemble the properties and behavior of larger scale IaaS systems. Thus, it ensures the applicability of our conclusions on large—even commercial—IaaS systems. This proprietary IaaS used a round-robin VM placement algorithm for virtual machine scheduling and provided active repositories (of the AVS system) on all hosts. This repository installation ensured a two level hierarchy: local and remote with package access latencies of 22 ms and 53 ms respectively). The underlying infrastructure of the university consists of eight hosts each configured with 4 CPUs, 4 GiBs of RAM and 80 GiBs of HDD (40 MB/s write throughput). All hosts were interconnected with gigabit Ethernet.

6.2 Experimental Results

First, we investigated how MMVAs influence the behavior of the optimization facility. We ran the size optimization of our Apache appliance $(p_\sigma = optimize(p_{Ap}))$ 20 times—10 with and 10 without using the management capabilities. The properties of the optimized Apache appliance are summarized in Table 2 (these properties are independent from the use of the management capabilities). We monitored the resource (e.g., network and CPU) usage costs throughout each optimization run. We also measured the deployment time of the suboptimal appliances created throughout the optimization runs. Using these measurements, we calculated the deployment time advantage of each suboptimal appliance compared to the original one. Using this time advantage value, we estimated the minimum number of future deployments (N_{dep}) necessary to overcome the optimization time spent for creating a particular suboptimal appliance. This estimate can be used as a guide for appliance developers when they determine on further appliance optimization (as mentioned in Section 3.1). Finally, we calculated the average of these estimates for every optimization run.

These averaged estimates are presented in Fig. 7. The results show that by reusing virtual machines, the MMVA based Algorithm significantly reduces the optimization time (from 2 hours 3 minutes to 1 hours 22 minutes—a 33 percent decrease) and it also reduces the initial optimization cost (90 percent-25 percent decrease for more than half of the optimization time). When we utilize management interfaces, the overall optimization cost is notably cut. With the simplified algorithm, 343 deployments were necessary to regain the optimization time. Whereas the MMVA based optimization required only 223 deployments to regain costs (35 percent decrease in overall cost).

As several IaaS providers charge for consumed bandwidth, we also measured the data transfers caused by our algorithms. On each optimization run, whenever the MMVA based algorithm would create an intermediate virtual appliance, we measured the total data transfers since the last evaluation. The results are presented in Fig. 8 and summarized in Table 3. The results show that the MMVA based algorithm excels independently from the use of intermediate appliance creation (by using an upload threshold of 1.1). The MMVA based approach reduced the necessary transfers for the optimization operation by over 85 percent compared to the original algorithm. To show the potential of MMVAs, we also added intermediate VA creation to the simplified algorithm. The results of this solution are presented in the third line of the Table and in Fig. 8b as the baseline. Compared to this solution, the MMVA based approach still offers almost 75 percent reduction in transfers.

Afterwards, we prepared an experiment for the evaluation of the new rebuilding technique. First, our MMVA (p_μ)

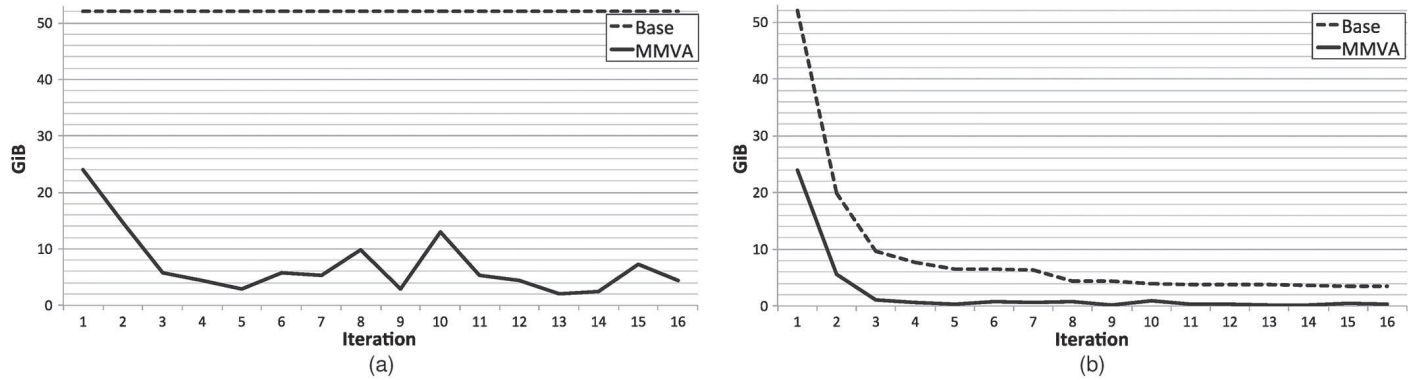

Fig. 8. Transferred data for VM instantiations during size optimization. (a) Without creating intermediate appliances: $ut = \infty$. (b) While creating intermediate appliances: $ut = 1.1$

Authorized licensed use limited to: Klarquist Sparkman LLP. Downloaded on November 20,2025 at 21:18:46 UTC from IEEE Xplore. Restrictions apply.

TABLE 3
Summary of Transferred Data During Size Optimization

	Transferred	Used BW	MMVA effect
Original	832.6 GiB	157.1 MB/s	Baseline
MMVA only	114.2 GiB	21.6 MB/s	7.3x
Int. VAs only	143.4 GiB	27.1 MB/s	Baseline
MMVA + Int. VAs	36.9 GiB	7 MB/s	3.9x

was added to all repositories: $\forall r \in R : (p_\mu \in contents(r))$. Then, the optimized Apache appliance (p_σ) was published in a single dedicated repository: $\exists r_x \in R : (p_\sigma \in contents(r_x) \wedge (\forall r \in (R \setminus \{r_x\}) : (p_\sigma \notin contents(r)))$. That repository automatically decomposed the Apache appliance and identified the 39 new files that are not common with the our MMVA: $p_\sigma \rightarrow \{p_\mu, p'_\sigma\}$. After decomposition, the repository held the following packages: $contents(r_x) = \{p_\mu, p_\sigma, p'_\sigma\}$. The decomposition defined the new Apache package with the following properties: $dep(p'_\sigma) = \{p_\mu\}$ and $size(p'_\sigma) = 6.4MiB$. This new Apache package directly represents the key functionality of the initial p_σ.

After these preparations, we deployed p'_σ 100 times on a host that did not store p'_σ so the system needed to acquire the package with the key functionality from repository r_x. During the deployments we measured Apache webserver's VM instantiation time (composed of the instnatiation time of our MMVA and the rebuilding time of the Apache appliance). For the MMVA based rebuilding (see Section 5.1), we used the *pre-transfer measurement threshold* value of $\tau = 0.1$. The second row of Fig. 9 shows the average measurement results and compares them to the scenario when the IaaS system downloads and instantiates the entire optimized Apache appliance from the remote r_x. The deployment time decreased by 14 percent that also positively affects the optimization by further reducing the number of future deployments necessary to regain the optimization cost. In the current case it reached 216 (a 37 percent reduction in costs compared to the simplified approach).

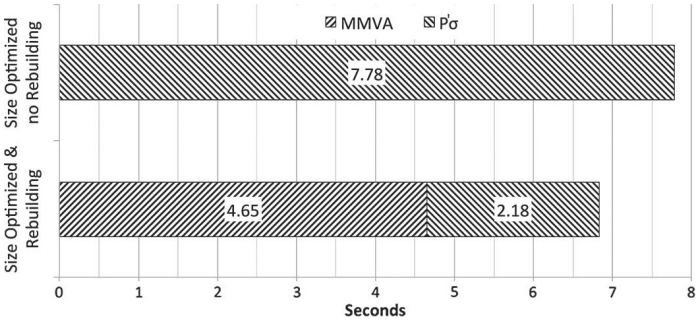

Fig. 9. Comparing the delivery of the Apache appliance.

7 Conclusion

This article analyzed and identified the adoptability issue of current virtual appliance delivery optimization systems. They are either dependent on changes of the current IaaS systems or they require change in virtual appliance repositories. To overcome this problem we introduced the concept of the *minimal manageable virtual appliance*. This appliance can be used as a template by appliance developers to construct their own appliances with embedded management functionalities required for delivery optimization systems. Then, we offer a methodology to embed MMVAs in custom appliances and analyse the impact of MMVAs on delivery systems.

The article evaluates the use of MMVAs in the Automated Virtual appliance creation Service (AVS). Throughout the evaluation, we discuss the new ways the AVS can interact with virtual appliances and with their instances. The article reveals and analyzes how the MMVA based solutions reduce the impact of delivery optimization systems (like the AVS) on the current IaaS systems and repositories. Finally, the measurements confirm the presumed effects on the experimental IaaS system: the MMVA enabled delivery optimization system does not need any changes in IaaS systems or in repositories but maintains the improved delivery times.

We have identified several future research goals. First, a developer friendly toolset needs to be designed for manageable appliance creation utilizing the basic methodology for embedding MMVAs. This toolset should also consider scenarios that allow developers to debug the multi phased virtual appliance instantiation process. Next, the current testbed was set up for short running experiments, however the use of MMVAs could also affect delivery costs on the long term. Research needs to focus on the long term effects of using MMVAs compared to non-MMVA enabled delivery systems. Rebuilding and size optimization represent just the initial use cases where the application of MMVAs are beneficial. Further investigations are needed to identify possible new use cases (e.g., focusing on maintenance cost reduction options or new VM image transformation approaches). Also, to improve the performance of instantiating multi-package virtual appliances, new more complex kinds of MMVAs and new algorithms for measuring connectivity should be investigated. Finally, for rebuilding scenarios where initial virtual machine startup is priorized, we are also aiming at combining the benefits of MMVAs with novel copy-on-read/write methods.

Acknowledgment

The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

References

[1] L.M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, ''A Break in the Clouds: Towards a Cloud Definition,'' *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 1, pp. 50-55, Jan. 2009.

[2] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R.H. Katz, A. Konwinski, G. Lee, D.A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, ''Above the Clouds: A Berkeley View of Cloud Computing,'' Univ. California Berkley, Berkley, CA, USA, Tech. Rep. UCB/EECS-2009-28, Feb. 2009.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, ''Xen and the Art of Virtualization,'' in *Proc. 19th ACM SOSP*, Oct. 2003, pp. 164-177.

[4] VMWare Inc., Palo Alto, CA, USA. [Online]. Available: http://www.vmware.com

[5] N. Susanta and C. Tzi-Cker, ''A Survey on Virtualization Technologies,'' Stony Brook Univ., Stony Brook, NY, USA, Tech. Rep. ECSL-TR-179. [Online]. Available: http://www.ecsl.cs.sunysb.edu/tr/TR179.pdf

Authorized licensed use limited to: Klarquist Sparkman LLP. Downloaded on November 20,2025 at 21:18:46 UTC from IEEE Xplore. Restrictions apply.

[6] Amazon Web Services LLCAmazon Elastic Compute Cloud. 2012. [Online]. Available: http://aws.amazon.com/ec2/
[7] K. Keahey, I. Foster, T. Freeman, X. Zhang, and D. Galron, ''Virtual Workspaces in the Grid,'' Math. Comput. Sci., Argonne Nat. Lab., Argonne, IL, USA, ANL/MCS-P1231-0205, 2005.
[8] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, ''The Eucalyptus Open-Source Cloud-Computing System,'' in *Proc. IEEE/ACM CCGRID*, 2009, pp. 124-131.
[9] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M.S. Lam, and M. Rosenblum, ''Virtual Appliances for Deploying and Maintaining Software,'' in *Proc. 17th USENIX Conf. LISA Syst. Admin.*, 2003, pp. 181-194.
[10] D. Geer, ''The OS Faces a Brave New World,'' *Computer*, vol. 42, no. 10, pp. 15-17, Oct. 2009.
[11] T. Zhanga, Z. Dua, Y. Chenb, X. Jic, and X. Wang, ''Typical Virtual Appliances: An Optimized Mechanism for Virtual Appliances Provisioning and Management,'' *J. Syst. Softw.*, vol. 84, no. 3, pp. 377-387, Mar. 2011.
[12] H. Nishimura, N. Maruyama, and S. Matsuoka, ''Virtual Clusters on the Fly—Fast, Scalable, Flexible Installation,'' in *Proc. 7th IEEE Int. Symp. CCGRID*, 2007, pp. 549-556.
[13] rPath—rBuilder, Raleigh, CA, USA. [Online]. Available: http://www.rpath.com/rbuilder/
[14] *Voptimizer, Optimization of Virtual Machine Size and Performance*, Vizioncore Inc., Hoffman Estates, IL, USA, 2008.
[15] R. Bradshaw, N. Desai, T. Freeman, and K. Keahey, ''A Scalable Approach to Deploying and Managing Appliances,'' presented at the TeraGrid Conference, Madison, WI, USA, June 2007.
[16] C. Peng, M. Kim, Z. Zhang, and H. Lei, ''VDN: Virtual Machine Image Distribution Network for Cloud Data Centers,'' in *Proc. IEEE INFOCOM*, 2012, pp. 181-189.
[17] *Rackspace Cloud Hosting Solutions*, Rackspace US Inc., San Antonio, TX, USA, Sept. 2012. [Online]. Available: http://www.rackspacecloud.com/
[18] *Windows Azure Services Platform*, Microsoft Inc., Redmond, WA, USA, Sept. 2012. [Online]. Available: http://www.windowsazure.com/
[19] G. Kecskemeti, G. Terstyanszky, P. Kacsuk, and Z. Nemeth, ''An Approach for Virtual Appliance Distribution for Service Deployment,'' *Future Gen. Comput. Syst.*, vol. 27, no. 3, pp. 280-289, Mar. 2011.
[20] N. Desai, A. Lusk, R. Bradshaw, and R. Evard, ''BCFG: A Configuration Management Tool for Heterogeneous Environments,'' in *Proc. IEEE Int. Conf. CLUSTER Comput.*, Dec. 2003, pp. 500-503.
[21] X. Jin, R. Willenborg, Y. Zhao, C. Sun, L. He, Z. Chen, Y. Chen, and Q. Wang, ''Reinventing Virtual Appliances,'' *IBM J. Res. Develop.*, vol. 53, no. 4, pp. 7:1-7:15, July 2009.
[22] D. Lutterkort and M. McLoughlin, ''Manageable Virtual Appliances,'' in *Proc. Ottawa Linux Symp.*, Ottawa, ON, Canada, June 2007, pp. 293-302.
[23] M.S. Wilson, ''Constructing and Managing Appliances for Cloud Deployments from Repositories of Reusable Components,'' in *Proc. Conf. Hot Topics Cloud Comput.*, 2009, pp. 1-5.
[24] S.-M. Yoo, J.W.-K. Hong, J.-G. Park, C.-W. Ahn, and S.-W. Kim, ''Performance Evaluation of WBEM Implementations,'' *KNOM Rev.*, vol. 8, no. 2, pp. 7-13, Feb. 2006.
[25] *CIM Standard Specifications*, Distributed Management Task Force, Portland, OR, USA, 2009. [Online]. Available: http://www.dmtf.org/standards/cim/
[26] C. Tang, ''FVD: A High-Performance Virtual Machine Image Format for Cloud,'' in *Proc. USENIX Annu. Tech. Conf.*, 2011, p. 18.
[27] M. Hines and K. Gopalan, ''Post-Copy Based Live Virtual Machine Migration Using Adaptive Pre-Paging and Dynamic Self-Ballooning,'' in *Proc. ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.*, 2009, pp. 51-60.
[28] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl, ''A Journey to Highly Dynamic, Self-Adaptive Service-Based Applications,'' *Autom. Softw. Eng.*, vol. 15, no. 3, pp. 313-341, Dec. 2008.
[29] K. Keahey and T. Freeman, ''Contextualization: Providing One-Click Virtual Clusters,'' in *Proc. 4th IEEE Int. Conf. ESCIENCE*, 2008, pp. 301-308.
[30] Public EC2 Amazon Machine Images. 2012. [Online]. Available: https://aws.amazon.com/amis
[31] VMWare Public Virtual Appliances. 2012. [Online]. Available: https://solutionexchange.vmware.com/store/category_groups/virtual-appliances
[32] J. Bresnahan, K. Keahey, D. LaBissoniere, and T. Freeman, ''Cumulus: An Open Source Storage Cloud for Science,'' in *Proc. 2nd Int. Workshop ScienceCloud Comput.*, 2011, pp. 25-32.
[33] G. Kecskemeti, G. Terstyanszky, and P. Kacsuk, ''Virtual Appliance Size Optimization with Active Fault Injection,'' *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 10, pp. 1983-1995, Oct. 2012.

Gabor Kecskemeti received the MSc degree from the University of Miskolc, Miskolc, Hungary, in 2004, and the PhD degree from the University of Westminster, London, UK, in 2011. He is a postdoctoral fellow at the Distributed and Parallel Systems group at University of Innsbruck, Innsbruck, Austria. He has got involved in several successful grid and service-oriented research projects (e.g., ePerSpace, EDGeS, S-Cube). He was coordinating the Cloud Computing research group of the MTA SZTAKI LPDS in 2011-2012. He has published over 30 scientific papers (incl. four journal articles) about cloud and grid computing, particularly from the field of virtual appliance delivery in IaaS clouds. He is a member of the IEEE.

Gabor Terstyanszky received the MSc degree from the Electrotechnical University, St. Petersburg, Russia, in 1977, the MPhil and PhD degrees from the University of Miskolc, Miskolc, Hungary, in 1990 and 1997, respectively. He is a Professor in distributed systems at the University of Westminster, London, U.K. He is the Co-leader of the Centre for Parallel Computing at the University of Westminster. His research interests cover distributed and parallel computing systems including clouds, desktop, and service grids. He was involved in more than 20 research projects either as a Local Coordinator or a Principal Investigator. He has published more than 100 papers in periodicals and conference proceedings on distributed and parallel computing systems. He was a Member of Program Committee of several European and world conferences. He was also a Guest Editor of several special issues published in periodicals on distributed computing infrastructures.

Peter Kacsuk received the MSc and University doctorate degrees from the Technical University of Budapest, in 1976 and 1984, respectively, and the kandidat degree (equivalent to PhD) and the Doctor of Academy degree (DSc) from Hungarian Academy of Sciences, Budapest, Hungary, in 1989 and 2001, respectively. He habilitated at the University of Vienna in 1997. He is the Director of the Laboratory of the Parallel and Distributed Systems in MTA SZTAKI, Budapest, Hungary. He received his professor title from the Hungarian President in 1999. He is a co-editor-in-chief of the Journal of Grid Computing published by Springer. He coordinated four EU Grid projects: EDGeS, EDGI, SCI-BUS, and SHIWA.

Zsolt Nemeth received the MSc degree from the Technical University of Budapest, Budapest, Hungary, in 1994, and the PhD degree from the Budapest University of Technology and Economics, Budapest, Hungary, in 2001. He is a Senior Researcher at the Laboratory of Parallel and Distributed Systems, MTA SZTAKI, Budapest, Hungary. His research work covered various aspects of parallel and distributed computing, formal modeling, grid computing, and unconventional programming paradigms. His work on semantic modeling of grid computing met a reception. He has been a visiting researcher at Emory University and an ERCIM Fellow at IRISA-INRIA and FORTH.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.

Authorized licensed use limited to: Klarquist Sparkman LLP. Downloaded on November 20,2025 at 21:18:46 UTC from IEEE Xplore. Restrictions apply.