Tools and Techniques for Managing Virtual Machine Images

Håvard K. F. Bjerke[1], Dimitar Shiyachki[1], Andreas Unterkircher[1], and Irfan Habib[2]

[1] CERN, 1211 Geneva 23, Switzerland
[2] Centre for Complex Cooperative Systems (CCCS), University of the West of England (UWE) Frenchay, Bristol BS16 1QY United Kingdom

Abstract. Virtual machines can have many different deployment scenarios and therefore may require generation of multiple VM images. OS Farm is a service that aims to provide VM images that are tailored and generated on the fly. In order to optimize generation of images, a layered copy-on-write image structure is used, and an image cache ensures that identical images are not regenerated.

Images can be several hundreds of megabytes large and thus can congest the network and delay their transfer. Content-Based Transfer is a technique which transfers only the difference between the source image and existing target client image data. We present an implementation which achieves an observed bandwidth close to the theoretical maximum and a significant reduction in network congestion.

1 Introduction

Virtualization can add agility to datacenters by providing flexible testing environments, failover with live-migration and satisfying different OS flavour requirements with consolidation. In all these scenarios it is important to have an infrastructure that efficiently handles the needed VM images. Sect. 2 presents a real scenario for the application of image management techniques, as a motivation for this work.

Libfsimage is a library and a standalone application, which generates VM images with a rich selection of Linux distributions. It is presented in Sect. 3.

OS Farm is a software application that aims to provide a user interface for generating and managing VM images. For generating images, it uses Libfsimage. It employs some techniques in order to optimize the generation and propagation of images, as described in Sect. 4.

The large sizes of VM images is a hurdle for managing images, particularly in image propagation. An implementation of the Content-Based Transfer technique, which optimizes the propagation of VM images over the network, is presented in Sect. 5.

This paper presents our work as a solution for image management with a good degree of configuration flexibility and performance.

2 Application of Image Management Techniques in the EGEE/WLCG Grid

The Enabling Grids for E-scienceE (EGEE)[1] project, funded by the European Union, provides a seamless Grid infrastructure for e-Science. EGEE produces the gLite[2] middleware for grid computing. Tightly coupled to EGEE is the Worldwide LHC Computing Grid (WLCG)[3]. Its mission is to build and maintain a data storage and analysis infrastructure for the entire high energy physics community that will use the Large Hadron Collider (LHC), which is currently being built at the European Organization for Nuclear Research (CERN)[4].

2.1 Grid Middleware Certification

A section in CERN's IT Department is responsible for the integration, testing and release of the gLite middleware. This activity is carried out in collaboration with several partners all over Europe within EGEE. Testing gLite faces the problem that its components are under active development. To enable progress in certification the turnaround time from feature submission to certified state must be as small as possible.

Bug fixes and new features enter gLite via the concept of a patch. A grid testbed is being operated so that new patches can be applied to the relevant grid nodes. However, certification of several patches at the same time can cause conflicts on the testbed. A non functional patch may spoil the whole testbed. To cope with such problems an infrastructure of virtual machines based on Xen was established so that certifiers can bring up grid nodes with a certain patch independently.

GLite is available on different Linux flavors and architectures: Scientific Linux CERN[5] 3 and 4 on i686 and x86_64 platforms. More Linux flavors (e.g. Debian) are under development. As all these combinations are found in production, interactions of nodetypes with different operating systems and hardware must be tested. To speed up the certification process we need to be able to quickly produce pre-defined images of different gLite nodetypes to use with Xen. We produce such images on a weekly basis in order to reflect the latest updates. The tools libfsimage and OS Farm were developed at CERN to achieve the aforementioned goals.

2.2 Usage of Virtual Machines on the Grid

The EGEE/WLCG infrastructure lets users send jobs for execution to more than 250 sites. Using the information system a user can determine the operating system provided by a site. However as more and more users from different scientific communities join the grid it gets difficult for sites to fullfill all their requirements in terms of operating systems and installed software. One way to deal with this problem is to provide each job on the grid a virtual machine with a dedicated OS setup. With thousands of users on the grid transferring images to sites becomes an issue. The content based image transfer described in Sect. 5 shows how to overcome this problem.

3 Libfsimage

Libfsimage is a library of Linux file system generation routines implemented in Python. Its primary goals are the simultaneous generation of 32-bit and 64-bit file systems for different Linux distributions in an isolated environment and reuse of the common setup and configuration code between the distributions. At present, the supported Linux flavors are Debian[6], Ubuntu[7], CentOS[8], Scientific Linux CERN and Fedora[9]. The package installation and dependency resolution for the first two are done with Debian package management tools. For the RPM[10] based distributions this is achieved with Yum[11].

Libfsimage uses pre-built environments that reflect the hardware architecture as well as the type and version of the package manager used in the relevant version of the distribution being generated. When possible these pre-built environments are shared between the different distributions. Prior to the generation the appropriate environment is deployed and the library uses the chroot system call to switch to a new root directory. The initial package installations in the generated file system are done from there. Once the latter contains the basic libraries, a package manager and configuration tools, further package installations and image configuration are performed from inside the new file system, again by leveraging the chroot capability.

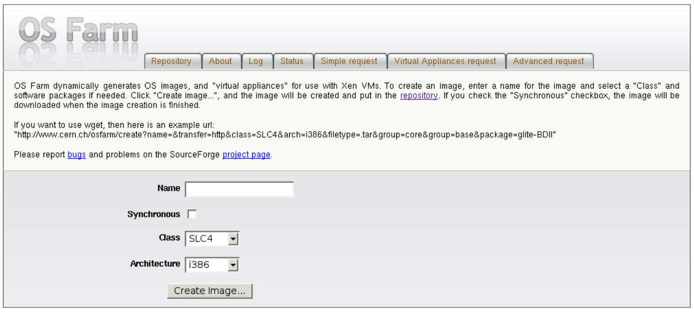
Libfsimage can be used both from the command line and as a python library. In the latter case it manages a Workspace object that keeps track of the deployed environments and their status in order to speed up the file system generation by reusing them.

A consequence of the extensive use of the chroot system call by Libfsimage, and Debian and RPM based package managers is the root privileges indispensability. This is a major drawback in the scenario when Libfsimage is used for simultaneous creation of images for different third parties that need to install custom packages inside the generated file system. A harmful pre- or post-install scriptlet that is run during the installation of a package could escape the chroot jail and run arbitrary code with root privileges, thus compromising the host and interfering with the generation of other parties' images. To address this issue a SELinux policy module is being developed for narrowing the standard root capabilities to the required minimum and confining the concurrently running generation processes in dedicated SELinux domains, so that misuse of the CAP_CHROOT privilege can not lead to a system compromise.

4 OS Farm

OS Farm creates VM images and Virtual Appliances (VA) [12] that can satisfy different needs. It provides a web interface through which users can choose between a range of Linux distributions and yum repositories, with their corresponding yum packages. Images are generated using libfsimage, which provides a rich selection of Linux distributions, which in OS Farm are called *classes*.

The main interface to an OS Farm service is a web interface, which is shown in Fig. 1. It provides several ways for the user to request a VM image. The most



OS Farm dynamically generates OS images, and "virtual appliances" for use with Xen VMs. To create an image, enter a name for the image and select a "Class" and software packages if needed. Click "Create image...", and the image will be created and put in the repository. If you check the "Synchronous" checkbox, the image will be downloaded when the image creation is finished.

If you want to use wget, then here is an example url:
"http://www.cern.ch/osfarm/create?name=&transfer=http&class=SLC4&arch=i386&filetype=.tar&group=core&group=base&package=glite-BDII"

Please report bugs and problems on the SourceForge project page.

Name

Synchronous ☐

Class SLC4

Architecture i386

Fig. 1. OS Farm user interface

```
<image>
    <name>MySlc4Image</name>
    <class>SLC4</class>
    <architecture>i386</architecture>
    <package>emacs</package>
    <package>unzip</package>
    <group>Base</group>
    <group>Core</group>
</image>
```

Fig. 2. OS Farm VM image specification

basic method is a *Simple request*, which allows the user to select image class and architecture.

Advanced request extends *Simple request* and gives the possibility of adding yum packages to the image. A dropdown menu allows the user to choose between a list of predefined yum repositories. Using Asynchronous JavaScript and XML [13], a further list is returned which allows the user to select the corresponding yum packages.

OS Farm also supports image requests by XML descriptions. An image description can be uploaded as an XML file. An example image description is shown in Fig. 2.

4.1 Layered Generation and Caching

The generation of an image is divided into three layers or stages, *core*, *base* and *image*. *Core* is a small functional image with a minimal set of software required to run the image on a Virtual Machine Monitor or in order to satisfy higher level software dependencies. *Base* is a layer on top of *core*, which provides some additional software needed in order to satisfy requirements for VAs. An *image* is also a layer on top of *core* and provides user defined software in addition to the *core* software. A subclass of *image* is *virtual appliance*, which is an image with an extra set of rules aimed to allow the image to satisfy requirements of

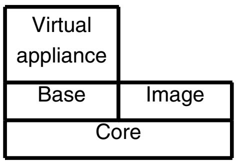


Fig. 3. Layers of a VM image

the deployment scenario. Fig. 3 shows that *core* can be shared between images, while *base* can be shared between VAs.

In order to accelerate the generation of images, *core* and *base* are always cached the first time they are generated. The layers are cached in Logical Volume Manager (LVM) [14] volumes. This allows higher layers to continue instantaneously, using copy-on-write, with the snapshot feature of LVM. *Images* are also cached and tagged such that if an image request matches that of a cached image, the image is returned immediately.

The tag of a cached image is the checksum of its configuration parameters, such as architecture and yum packages. Whenever an image is requested, a checksum is generated from the requested configuration and looked up in the cache. If an image with the exact same configuration parameters already exists in the cache, the image is returned immediately. A timeout value can be set on a cache entry in order to limit the validity of it. If an entry has timed out, a request for that entry results in a regeneration of it.

The cache also serves as a browsable repository for images. Instead of requesting images by parameters, images can be browsed and downloaded directly from the cache.

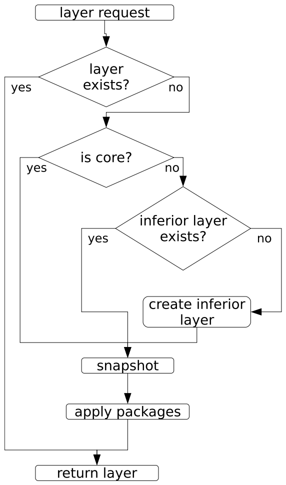


Fig. 4. Flowchart of the recursive image request and generation process

Fig. 4 shows the recursive image generation process. A request for an image is effectively a request for a layer, which in turn requires the presence of its inferior layer *core*, which, if it does not exist, triggers its creation.

The speedup gained from using these techniques varies between image configurations. We have measured the execution times of three simple example scenarios:

- the cache is empty, i.e. the image is generated from scratch: 254 seconds
- *core* is in the cache: 72 seconds
- *image* is in the cache: instantaneous

The results show that a significant speedup can be gained when an image or one or more of its layers are cached.

5 Content-Based Transfer

Content Based Transfer (CBT) is a technique to efficiently transfer VM image data from a source host to a target host. It takes advantage of knowing the structure of the image data to extract common data which need not be transmitted.

Most filesystems are aligned on a fixed boundary. For example, in the Ext2 filesystem, a file is aligned in blocks of size $1024 * 2^n$, where $0 \leq n \leq 2^{32}$ [15]. This means that two identical files on two different ext2 volumes, will be contained in a set of identical blocks on both volumes, even if they are fragmented. Moreover, if block sizes are different on the two volumes, as long as the largest block size of the two volumes is a multiple of the smallest block size, all files in both volumes will be aligned on the smallest block boundary. In our experience, block sizes of 1024 and 4096 are most frequent.

[16] examines the commonality between filesystem volumes. In our experiments we have analysed two different scenarios:

- two computer centre batch systems,
- two major versions of Scientific Linux CERN (SLC) VM images

Our experiments have shown that commonality ranges from moderate to significant. The commonality between some example volumes is shown in Tab. 1.

Before transmitting a volume, A, across the network, a comparison can be done between an existing volume at the destination, B, and A. If any blocks in A already exist in B, then those blocks need not be transmitted. Moreover, any

Table 1. Commonality between example volumes

Scenario	Commonality
Batch system to batch system	84 %
SLC4 to SLC3	48 %
SLC3 to SLC4	22 %

blocks in A which exist in any of the volumes at the destination, need not be transmitted.

Content Adressable Storage [17] exploits commonality in order to reduce storage load. In [18] this is exploited in order to reduce the load on the network and storage. We present an implementation which exploits commonality in order to reduce network load and speed up network transfer to close to the theoretical maximum.

5.1 Implementation

Our implementation of Content-Based Transfer uses Java, as a good compromise between efficiency and convenience. Most notably, it exploits Java's hash digest algorithms and thread-safe hash tables.

Identical blocks are identified with checksums, which are calculated using the available hash digest algorithms in Java. In spite of discovered collisions in the MD5 algorithm[1] [19], for the results in this paper, we have used it because it is the fastest available algorithm in the Java library. The choice of hashing algorithm, however, is open to the user.

The implementation is split into several threads that pass checksums and blocks among each other in a pipelined fashion. For example, one thread scans the source image and generates a checksum for the current block. Immediately, before continuing to the next block, the checksum is passed to the next thread. Concurrently, another thread, at the destination node, receives a checksum and looks it up in its hash table. If the block is not already at the destination, then it is requested from the source, in a similar pipelined fashion. The key to achieving good efficiency in this implementation is to allow the blocks that are already at the destination be written simultaneously and at the same pace as they are read from the source (which should be the disk read speed), and the other blocks be transmitted at the pace which the network allows.

5.2 Hypothetical Maximal Observed Bandwidth

The information needed to be transmitted from the source to the target consists of a *differential*, s_D, and an *identical*, s_I, part. In the differential part, all data is different between the source and the target, so all source data must be transmitted, and thus the transfer speed for the differential part is bound by the network transmission bandwidth, v_n, given that the disk speed is higher than the network bandwidth. In the identical part, data is identical between the source and the target, so data must only be identified and only identification information needs to be transmitted, and thus the transfer speed for the identical part is bound by the disk read speed, v_d.

Given an I/O bound only program (CPU speed is infinite, CPU time is 0), the hypothetical best transfer time is

[1] MD5 is not recommended for security application, since a collision can actively be created. This concern is not that relevant for our application because the user is not expected to actively create blocks that will collide with his or her own blocks.

$$t = \frac{s_I}{v_d} + \frac{s_D}{v_n} \ , \tag{1}$$

iff

$$v_d > v_n \ . \tag{2}$$

The total transfer time of an image, using this technique, is different from a non-optimized technique. With the optimized transfer time, we calculate an *observed bandwidth*. The hypothetical best observed bandwidth is, given (2),

$$v = \frac{s}{\frac{s_I}{v_d} + \frac{s_D}{v_n}} \ . \tag{3}$$

In general,

$$v = \frac{1}{\frac{\Delta_I}{v_d} + \frac{\Delta_D}{v_n}} \ , \text{ where } \Delta_j = \frac{s_j}{s} \in [0, 1] \ . \tag{4}$$

The intent of the hypothetical best observed bandwidth is to determine the performance of a hypothetical optimal CBT implementation to use as a benchmark for our CBT implementation. "Observed bandwidth" serves as a measure of performance that is independent of image sizes and indicates the speedup given by the CBT technique compared to a non-optimized technique.

On our test system we measured, using the Unix command "dd," a disk read speed of 35.6 MB/s. The test systems were also equipped with a 100 Mb/s network interface card, and the theoretical max bandwidth of a 100 Mb/s line is 11.9 MB/s. Using these v_d and v_n bandwidths, we calculated the *hypothetical* observed bandwidths which are given in Fig. 5.

5.3 Experimental Analysis

Running our implementation of CBT on our test systems, we measured the observed bandwidths which are given in Fig. 5. The results show that our CBT

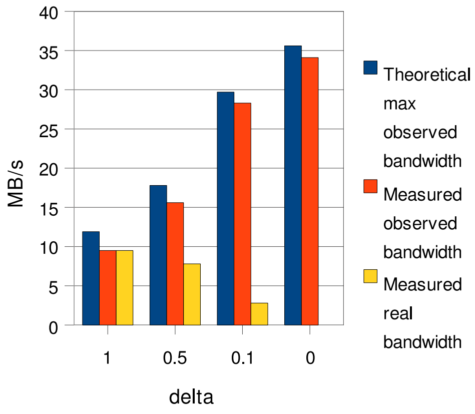


Fig. 5. CBT bandwidth

implementation achieves an observed bandwidth which closely follows the hypothetical maximum bandwidth. Also, the *real bandwidth*, which indicates the network load produced by the actual amount of data transmitted, is reduced, meaning a reduced load on the network.

It is worth noting that the observed bandwidth at the two extremes of Δ_D, at 0 and 1, are close to v_d and v_n, respectively. In our example, $v_d > v_n$. If $v_n > v_d$, as would be, for us, the case with a gigabit network, the disk read speed is the limitation, and CBT does not give any speedup. CBT in this case still has the advantage of reducing the load on the network.

6 Related Work

RPath[20] is a service that offers VAs, and provides a service similar to OS Farm. RPath's "recipe" approach to constructing a VA is a powerful method and gives great opportunity for reuse of packages between VAs. The approach encourages a VA developer, who develops recipes, and a VA user, who downloads the VAs, as two separate roles. The user can choose between a set of predefined VAs, but does not actively change the VA. In OS Farm, the user would normally also be the author of an image, since it is a trivial exercise.

If VM images are to be deployed on a large scale, they need to be adapted to their deployment context. Libfsimage allows paremeterized configuration of the images, but a future goal is to allow for contextualization[21].

Rsync[22] is an application that uses commonality in order to speed up the transmission of data. It uses SSH for authentication, which adds some overhead. The observed bandwidth given by Rsync, as calculated from the time reported by Rsync itself, which is lower than the total execution time including authentication, is not as high as CBT. For example, for $\Delta_D = 0.1$, Rsync gives a 30 MB/s observed bandwidth, and $\Delta_D = 0.5$ gives a 13 MB/s observed bandwidth. Another advantage CBT has is that it takes a set of images as a source for commonality, as opposed to Rsync, which uses only one target file.

7 Conclusion

We have presented tools and techniques for managing images, which help to overcome some of the problems that present themselves when managing an infrastructure of VMs. Libfsimage provides the different flavors and architectures that are needed in our use case and has a basis which allows extension for further flavors of Linux. It also lends itself as a library for external application, as in the example of OS Farm.

OS Farm uses Libfsimage and provides a graphical user interface for generation of VM images. It also provides a repository for VM images, which also serves to cache and optimize image generation through sharing layers of images.

CBT exploits commonality between images to optimize the transfer of images across the network. It achieves an observed bandwidth close to the theoretical maximal observed bandwidth. It can also help to avoid network congestion when transferring large images.

Acknowledgements

This work makes use of results produced by the Enabling Grids for E-sciencE project, a project co-funded by the European Commission (under contract number INFSO-RI-031688) through the Sixth Framework Programme. EGEE brings together 91 partners in 32 countries to provide a seamless Grid infrastructure available to the European research community 24 hours a day. Full information is available at http://www.eu-egee.org.

References

1. About EGEE, `http://public.eu-egee.org/intro/`
2. gLite, `http://glite.web.cern.ch/glite/`
3. Worldwide LHC Computing Grid, `http://lcg.web.cern.ch/LCG/`
4. European Organization for Nuclear Research, `http://public.web.cern.ch/public/`
5. Scientific Linux CERN, `http://linux.web.cern.ch/linux/`
6. Debian home page, `http://www.debian.org/`
7. Ubuntu home page, `http://www.ubuntu.com/`
8. CentOS home page, `http://www.centos.org/`
9. Fedora home page, `http://fedoraproject.org/`
10. RPM home page, `http://www.rpm.org/`
11. Yum home page, `http://linux.duke.edu/projects/yum/`
12. Saputzakis, C., et al.: Virtual Appliances for Deploying and Maintaining Software. In: Proceedings of LISA 2003 (2003)
13. Wikipedia article on AJAX, `http://en.wikipedia.org/wiki/AJAX`
14. LVM2 Resource Page, `http://sources.redhat.com/lvm2/`
15. Bovet, D.P., Cesati, M.: Understanding the Linux Kernel, pp. 574–607. O'Reilly, Sebastopol (2003)
16. Tolia, N., et al.: Using Content Addressing to Transfer Virtual Machine State (2002), `http://www.intel-research.net/Publications/Pittsburgh/050520030704_127.pdf`
17. Tolia, N., et al.: Opportunistic Use of Content Addressable Storage for Distributed File Systems. In: Proceedings of USENIX 2003 (2003)
18. Nath, P., et al.: Design Tradeoffs in Applying Content Addressable Storage to Enterprise-scale Systems Based on Virtual Machines. In: Proceedings of USENIX 2006 (2006)
19. Mikle, O.: Practical Attacks on Digital Signatures Using MD5 Message Digest, Cryptology ePrint Archive: Report 2004/356 (2004)
20. rPath home page, `http://www.rpath.com/`
21. Bradshaw, R., et al.: A Scalable Approach To Deploying And Managing Appliances. In: TeraGrid 2007, Madison, WI (June 2007)
22. Rsync home page, `http://samba.anu.edu.au/rsync/`