Proceedings of the Linux Symposium

June 27th–30th, 2007
Ottawa, Ontario
Canada

Manageable Virtual Appliances

David Lutterkort
Red Hat, Inc.
dlutter@redhat.com

Mark McLoughlin
Red Hat, Inc.
markmc@redhat.com

Abstract

Virtual Appliances are a relatively new phenomenon, and expectations vary widely on their use and precise benefits. This paper details some usage models, how requirements vary depending on the environment the appliance runs in, and the challenges connected to them. In particular, appliances for the enterprise have to fit into already existing infrastructure and work with existing sitewide services such as authentication, logging, and backup.

Appliances can be most simply distributed and deployed as binary images; this requires support in existing tools and a metadata format to describe the images. A second approach, basing appliances entirely on a metadata description that includes their setup and configuration, gives users the control they need to manage appliances in their environment.

1 Introduction

Virtual Appliances promise to provide a simple and cost-efficient way to distribute and deploy applications bundled with the underlying operating system as one unit. At its simplest, an appliance consists of disk images and a description how these disk images can be used to create one or more virtual machines that provide a particular function, such as a routing firewall, or a complete webmail system. Distributing a whole system instead of an application that the end user has to install themselves has clear benefits: installing the appliance is much simpler than setting up an application, the appliance supplier tests the appliance as a whole, and tunes all its components for the appliance's needs. In addition, appliances leave the appliance supplier much more latitude in the selection of the underlying operating system; this is particularly attractive for applications that are hard to port even amongst successive versions of the same distribution.

In addition to these installation-related benefits, some of the more general benefits of virtualization make appliances attractive, especially hardware isolation, better utilization of existing systems, and isolation of applications for improved security and reliability.

Because of their novelty, there is little agreement on what kind of appliance is most beneficial in which situation, how different end user requirements influence the tooling around appliances, and how users deploy and maintain their appliances. Examples of applications that are readily available as appliances range from firewalls and asterisk-based VoIP appliances, to complete Wikis and blogs, and database servers and LAMP stacks.

A survey of existing appliances makes one thing clear: there are many open questions around appliances; for them to fulfill their potential, these questions must be answered, and those answers must be backed by working, useful tools.

Comparing how current Linux distributions are built, delivered and managed to what is available for appliances points to many of these shortcomings: installing an appliance is a completely manual process, getting it to run often requires an intimate understanding of the underlying virtualization platform, and the ability to configure the virtual machine for the appliance manually; managing it depends too often on appliance-specific one-off solutions. As an example, for an appliance based on paravirtualized Xen, the host and the appliance must agree on whether to use PAE or not, a build-time configuration. A mismatch here leads to an obscure error message when the appliance is started. As another example, in an effort to minimize the size of the appliance image, appliance builders sometimes strip out vital information such as the package database, making it impossible to assess whether the appliance is affected by a particular security vulnerability, or, more importantly, update it when it is.

In addition, appliances bring some problems into the spotlight that exist for bare-metal deployments, too, but are made more pressing because appliances rely on the distribution of entire systems. The most important of these problems is that of appliance configuration—very few appliances can reasonably be deployed without any customization by the end-user. While appliances will certainly be a factor in pushing the capabilities and use of current discovery and zeroconf mechanisms, it is unlikely that that is enough to address all the needs of users in the near future.

We restrict ourselves to appliances consisting of a single virtual machine. It is clear that there are uses for appliances made up of multiple virtual machines, for example, for three-tier web applications, with separate virtual machines for the presentation logic, the business logic, and the database tier. Such appliances though add considerable complexity to the single-VM case: the virtual machines of the appliance need to be able to discover each other, might want to create a private network for intra-appliance communication, and might want to constrain how the virtual machines of the appliance should be placed on hosts. For the sake of simplicity, and given that handling single-VM appliances well is a necessary stepping stone to multi-VM appliances, we exclude these issues and focus solely on single-VM appliances.

The rest of the paper is organized as follows: Section 2 discusses appliance builders and users, and how their expectations for appliances vary depending on the environment in which they operate, Section 3 introduces images and recipes, two different approaches to appliances, Sections 4 and 5 describe images and recipes in detail, and the final Section 6 discusses these two concepts, and some specific problems around appliances based on the example of a simple web appliance.

2 Builders and Users

Appliances available today cover a wide variety of applications and therefore target users: from consumer software to applications aimed at SMB's, to enterprise software. It is worthwhile to list in more detail the expectations and needs of different applications and target audiences—it is unlikely that the differences that exist today between a casual home user and a sysadmin in a large enterprise will disappear with appliances.

Two separate groups of people are involved with appliances: *appliance builders*, who create and distribute appliances, and *appliance users* who deploy and run them. We call the first group builders rather than developers for two reasons: while appliance creation will often involve a heavy dose of software development, it will, to a large extent, be very similar to traditional development for bare-metal systems and the need for appliance-specific development tools will be minimal; besides creating appliances through very specific development, they can also be created by assembling existing components with no or minimal custom development. The value of these appliances comes entirely from them being approved, pretested and tuned configurations of software. In that sense, a sysadmin in an enterprise who produces "golden images" is an appliance builder.

What all appliance builders have in common is that they need to go through repeated cycles of building the appliance from scratch, and running, testing and improving it. They therefore need ways to make these steps repeatable and automatable. Of course, they also share the goal of distributing the appliance, externally or internally, which means that they need to describe the appliance in as much detail as is necessary for users to deploy them easily and reliably.

Users and their expectations vary as much as the environments in which they operate, and expectations are closely coupled to the environment in which the appliance will run. Because of the differences in users, the requirements on the appliances themselves should naturally vary, too: whereas for consumer software ease of installation and configuration is crucial, appliances meant for an enterprise setting need to focus on being manageable in bulk and fitting into an environment with preexisting infrastructure and policies. As an example, a typical hardware appliance, a DSL router with a simple web configuration interface is a great consumer device, since it lets consumers perform complicated tasks with ease, while it is completely unsuited for a data center, where ongoing, automated management of a large number of routers is more important than the ease of first-time setup.

These differences have their repercussions for virtual appliances, too: an appliance developer for a consumer appliance will put more emphasis on an easy-to-use interface, whereas for enterprise appliances the emphasis will be on reliability, manageability and the problems that only appear in large-scale installations. Building the user interface for a consumer appliance is tightly coupled to the appliance's function, and we consider it part of the custom development for the appliance; addressing the concerns of the enterprise user, the sysadmin, requires tools that go beyond the individual appliance, and therefore should be handled independently of the function of the appliance.

While for consumer appliances it is entirely reasonable to predefine what can and can not be configured about the appliance, this is not feasible for data center appliances: the infrastructures and policies at different sites are too varied to allow an appliance developer to anticipate all the changes that are required to make the appliance fit in. Often, these changes will have little bearing on the functioning of the appliance, and are concerned with issues such as logging to a central logging server, auditing shell access, monitoring the appliance's performance and resource usage etc. At the same time, enterprise users will demand that they can intervene in *any* aspect of the appliance's functioning should that become necessary, especially for security reasons.

Consumers and enterprise users also differ sharply in their expectations for a deployment tool: for a consumer, a simple graphical tool such as `virt-manager` is ideal, and adding features for downloading and running an appliance to it will provide consumers with a good basis for appliance use. For enterprise users, who will usually be sysadmins, it is important that appliances can be deployed, configured and managed in a way as automated as possible.

3 Images and Recipes

This paper details two different approaches to building and distributing appliances: *Appliance Images*, appliances distributed as one or more disk images and a description of how to create a virtual machine from them, and *Appliance Recipes*, appliances completely described in metadata.

Appliances can be built for any number of virtualization technologies and hypervisors; deployment tools for appliances should be able to understand appliance descriptions regardless of the virtualization technology they are built on, and should be able to at least let the user know ahead of time if a given appliance can be deployed on a given host. The appliance description therefore needs to carry information on the expected platform. Rather than encode this knowledge in appliance-specific deployment tools, we base the appliance description on libvirt and its XML metadata format for virtual machines, since libvirt can already deal with a number of virtualization platforms such as paravirtualized and fully-virtualized Xen hosts, qemu, and kvm and abstracts differences between them away. This approach also makes it possible for related tools such as `virt-manager` [1] and `virt-factory` [2] to integrate appliance deployment and management seamlessly.

The goal of both the image and recipe approach is to describe appliances in a way that integrates well with existing open-source tools, and to introduce as fewer additional requirements on the infrastructure as possible.

Deployment of Appliance Images is very simple, consisting of little more than creating a virtual machine using the disk images; for Appliance Recipes, additional steps amounting to a system install are needed. On the other hand, changing the configuration of Appliance Images is much harder and requires special care to ensure changes are preserved across updates, especially when updates are image-based, whereas Appliance Recipes provide much more flexibility in this area. In addition, Appliance Recipes provide a clear record of what exactly goes into the appliance, something that can be hard to determine with image-based appliances.

By virtue of consisting of virtual machines, appliances share some characteristics with bare-metal machines, and some of the techniques for managing bare-metal machines are also useful for appliances: for example, Stateless Linux [3] introduced the notion of running a system *readonly root* by making almost the entire file system readonly. For Stateless, this allows running multiple clients off the same image, since the image is guaranteed to be immutable. For image-based appliances, this can be beneficial since it cleanly delineates the parts of the appliance that are immutable content from those that contain user data. It does make it harder for the user to modify arbitrary parts of the appliance, especially its configuration, though Stateless provides mechanisms to make parts of the readonly image mutable, and to preserve client-specific persistent state.

Similarly, managing the configuration of machines is not a problem unique to appliances, and using the same tools for managing bare-metal and appliance configurations is very desirable, especially in large-scale enterprise deployments.

3.1 Relation to Virtual Machines

Eventually, an appliance, i.e., the artifacts given to the appliance user by the appliance builder, is used to create and run a virtual machine. If a user will only ever run the appliance in a single virtual machine, the appliance and the virtual machine are interchangeable, particularly for Appliance Images, in the sense that the original image the appliance user received is the one off which the virtual machine runs.

When a user runs several instances of the same appliance in multiple virtual machines, for example, to balance the load across several instances of the same web appliance, this relation is more complicated: appliances in general are free to modify any of their disk images, so that each virtual machine running the appliance must run off a complete copy of all of the appliance's disk images. The existence of the copies complicates image-based updates as the copy for each virtual machine must be found and updated, making it necessary to track the relation between original appliance image and the images run by each virtual machine. For Appliance Recipes, this is less of an issue, since the metadata takes the place of the original appliance image, and a separate image is created for every virtual machine running that appliance recipe.

3.2 Combining Images and Recipes

Appliance Images and Appliance Recipes represent two points in a spectrum of ways to distribute appliances. Image-based appliances are easy to deploy, whereas recipe-based appliances give the user a high degree of control over the appliance at the cost of a more demanding deployment.

A hybrid approach, where the appliance is completely described in metadata, and shipped as an image combines the advantages of both approaches: to enable this, the appliance builder creates images from the recipe metadata, and distributes both the metadata and the images together.

4 Appliance Images

The description of an Appliance Image, by its very nature, is focused on describing a virtual machine in a transportable manner. Such a description is not only useful for distributing appliances, but also anywhere where a virtual machine and all its parts need to be saved and recreated over long distances or long periods of time, for example for archiving an entire application and all its dependencies.

Appliance Images are also suitable for situations where the internals of the virtual machine are unimportant, or because the appliance is running an O/S that the rest of the environment can't understand in more detail.

With no internals exposed, the appliance then consists just of a number of virtual disks, descriptions of needed virtual hardware, most importantly a NIC, and constraints on which virtualization platform the appliance can run.

Appliance Images need to carry just enough metadata in their description to allow running them safely in an environment that knows nothing but the metadata about them. To enable distribution, Appliance Images need to be bundled in a standard format that makes it easy to download and install them. For simplicity, we bundle the metadata, as an XML file, and the disk images as normal tarballs.

Ultimately, we need to create a virtual machine based on the appliance metadata and the disk images; therefore, it has to be possible to generate a libvirt XML description of the virtual machine from the appliance's metadata. Using libvirt XML verbatim as the appliance description is not possible, since it contains some information, such as the MAC address for the appliance's NIC, that clearly should be determined when the appliance is deployed, not when it is built.

4.1 Metadata for Appliance Images

The metadata for an Appliance Image consists of three parts:

1. General information about the appliance, such as a name and human-readable label
2. The description of the virtual machine for the appliance
3. The storage for the appliance, as a list of image files

4.1.1 Virtual Machine Descriptor

The metadata included with an appliance needs to contain enough information to make a very simple decision that anybody (and any tool) wanting to run the appliance will be faced with: can this appliance run on this host? Since the differences between running a virtual machine on a paravirtualized and fully-virtualized host from an application's point of view are small, the appliance metadata allows describing them simultaneously with enough detail for tools to determine how to boot the appliance ahead of time; for each supported virtualization platform, the metadata lists how to boot the virtual machine on that platform, together with a description of the platform.

The boot descriptor roughly follows libvirt's `<os>` element: for fully-virtualized domains, it contains an indication of which bootloader and what boot device to use, and for paravirtualized domains, it either lists the kernel and initrd together or that `pygrub` should be used as well as the root device and possible kernel boot options. If the kernel/initrd are mentioned explicitly, they must be contained in the tarball used to distribute the appliance as separate files.

The platform description, based on libvirt's *capabilities*, indicates the type of hypervisor (for example, `xen` or `hvm`), the expected architecture (for example, `i686` or `ia64`), and additional features such as whether the guest needs `pae` or not.

Besides these two items, the virtual machine metadata lists how the disk images should be mapped into the virtual machine, how much memory and how many virtual CPUs it should receive, whether a (graphical) console is provided, and which network devices to create.

4.1.2 Disk Images

The disk images for the appliance are simple files; for the time being, we use only (sparse) raw files, though it would be desirable to use compressed `qcow` images for disks that are rarely written to.[1]

Disk images are classified into one of three categories to enable a simple update model where parts of the appliance are replaced on update. Such an update model requires that the appliance separates the user's data from the appliance's code, and keeps them on separate disks. The image categories are:

- *system* disks that contain the O/S and application and are assumed to not change materially over the appliance's lifetime
- *data* disks that contain application data that must be preserved across updates
- *scratch* disks that can be erased at will between runs of the appliance

The classification of disk images mirrors closely how a filesystem needs to be labeled for Stateless Linux: in a Stateless image, most files are readonly, and therefore belong on a system disk. Mutable files come in two flavors: files whose content needs to be preserved across reboots of the image, which therefore belong on a data disk, and files whose content is transient, and therefore belong on a scratch disk.

Updates of an Appliance Image can be performed in one of two ways: by updating from within, using the update mechanisms of the appliance's underlying operating system, for example, yum, or by updating from the outside, replacing some of the appliance's system disks. If the appliance is intended to be updated by replacing the system disks, it is very desirable, though not strictly necessary, that the system disks are mounted readonly by the appliance; to prepare the appliance for that, the same techniques as for Stateless Linux need to be applied, in particular, marking writable parts of the filesystem in `/etc/rwtab` and `/etc/statetab`.

Data disks don't have to be completely empty when the appliance is shipped: as an example, a web application with a database backend, bundled as an appliance will likely ship with a database that has been initialized and the application's schema loaded.

4.2 Building an Appliance Image

An appliance supplier creates the appliance by first installing and configuring the virtual machine for the appliance any way they see fit; typically this involves installing a new virtual machine, starting it, logging into it and making manual configuration changes. Even though

[1]Unfortunately, the Xen blktap driver has a bug that makes it impossible to use `qcow` images that were not created by Xen tools.

these steps will produce a distributable Appliance Image, they will generally not make building the Appliance Image reproducible in an automated manner. When this is important, for example, when software development rather than simple assembly is an important part of the appliance build process, the creation of the Appliance Image should be based on an Appliance Recipe, even if the appliance is ultimately only distributed as an image.

Once the appliance supplier is satisfied with the setup of the virtual machine, they create an XML description of the appliance, based on the virtual machine's characteristics. Finally, the virtual machine's disk is compressed and, together with the appliance descriptor, packed into a tarball.

This process contains some very mechanical steps, particularly creating the initial appliance descriptor and packing the tarball that will be supported by a tool. The tool will similarly support unpacking, installing, and image-based updating of the appliance.

4.3 Deploying an Appliance Image

An appliance user deploys an appliance in two steps: first, she downloads and installs the appliance into a well-known directory, uncompressing the disk images.

As a second step, the user simply runs `virt-install` to create the virtual machine for the appliance; `virt-install` is a command line tool generally used to provision operating systems into virtual machines. For Appliance Images, it generates the libvirt XML descriptor of the appliance's virtual machine from the appliance descriptor and additional user-provided details such as an explicit MAC address for the virtual machine's NIC, or whether and what kind of graphical console to enable.

To allow for multiple virtual machines running the same appliance concurrently, the disk images for the appliance are copied into per-VM locations, and `virt-install` records the relation between the appliance and the VM image. This is another reason why using `qcow` as the image format is preferrable to simple raw images, as it has a facility for *overlay images* that only record the changes made to a base image. With `qcow` images, instantiating a virtual machine could avoid copying the usually large system disks, creating only an overlay for them. Data disks, of course, still should be created through copying from the original appliance image, while scratch disks should be created as new files, as they are empty by definition.

It is planned to integrate Appliance Image deployment into `virt-manager`, too, to give users a simple graphical interface for appliance deployment.

In both cases, the tools check that the host requirements in the appliance descriptor are satisfied by the actual host on which the appliance is deployed.

4.4 Packaging Appliance Images as RPM's

Packaging appliances as tarballs provides a lowest-common-denominator for distributing appliances that is distribution, if not operating system, agnostic. Most distributions already have a package format tailored to distributing and installing software and tools built around them.

For RPM-based distributions, it seems natural to package appliances as RPMs, too. This immediately sets a standard for, amongst others, how appliances should be made available (as RPMs in yum repositories), how their authenticity can be guaranteed (by signing them with a key known to RPM), and how they are to be versioned.

5 Appliance Recipes

Appliance Recipes describe a complete virtual machine in metadata during its whole lifecycle from initial provisioning to ongoing maintenance while the appliance is in use. The recipe contains the specification of the appliance's virtual machine, which is very similar to that for Appliance Images, and a description of how the appliance is to be provisioned initially and how it is to be configured. In contrast to Appliance Images, it does not contain any disk images. An Appliance Recipe consists of the following parts:

1. An appliance descriptor describing the virtual machine; the descriptor is identical to that for Appliance Images, except that it must contain the size of each disk to be allocated for the virtual machine.

2. A kickstart file, used to initially provision the virtual machine from the recipe.

3. A puppet manifest, describing the appliance's configuration. The manifest is used both for initial provisioning, and during the lifetime of the appliance; updating the manifest provides a simple way to update existing appliances without having to reprovision them.

Before an appliance based on a recipe can be deployed in a virtual machine, the virtual machine's disks need to be created and populated, by installing a base system into empty disk images. Once the disk images have been created, deployment follows the same steps as that for an Appliance Image.

The Appliance Recipe shifts who builds the disk images from the appliance builder to the appliance user, with one very important difference: the appliance user has a complete description of the appliance, consumable by tools, that they can adapt to their needs. With that, the appliance user can easily add site-specific customizations to the appliance, by amending the appliance's metadata; for example, if all `syslog` messages are to be sent to a specific server, the appliance user can easily add their custom `syslog.conf` to the appliance's description. It also provides a simple mechanism for the appliance builder to leave certain parts of the appliance's configuration as deploy-time choices, by parameterizing and templating that part of the appliance's metadata.

We use kickstart, Fedora's automated installer, for the initial provisioning of the appliance image, and puppet, a configuration management tool, for the actual configuration of the appliance. It is desirable to expose as much of the appliance's setup to puppet and to only define a very minimal system with kickstart for several reasons: keeping the kickstart files generic makes it possible to share them across many appliances and rely only on a small set of well-tested stock of kickstart files; since kickstarting only happens when a virtual machine is first provisioned, any configuration the kickstart file performs is hard to track over the appliance's lifetime; and, most importantly, by keeping the bulk of the appliance's configuration in the puppet manifest it becomes possible for the appliance user to adapt as much as possible of the appliance to their site-specific needs. The base system for the appliance only needs to contain a minimal system with a DHCP client, `yum`, and the puppet client.

Strictly speaking, an Appliance Recipe shouldn't carry a full kickstart file, since it contains many directives that should be controlled by the appliance user, not the appliance builder, such as the timezone for the appliance's clock. The most important parts of the kickstart file that the appliance builder needs to influence are the layout of the appliance's storage and how disks are mounted into it, and the yum repositories needed during provisioning. The recipe should therefore only ship with a partial kickstart file that is combined with user- or site-specific information upon instantiation of the image.

5.1 Configuration Management

The notion of Appliance Recipes hinges on describing the configuration of a virtual machine in its entirety in a simple, human-readable format. That description has to be transported from the appliance builder to the appliance user, leaving the appliance user the option of overriding almost arbitrary aspects of the configuration. These overrides must be kept separate from the original appliance configuration to make clean upgrades of the latter possible: if the user directly modifies the original appliance configuration, updates will require cumbersome and error-prone merges.

The first requirement, describing the configuration of a system, is the bread and butter of a number of configuration management tools, such as cfengine, bcfg2, and puppet. These tools are also geared towards managing large numbers of machines, and provide convenient and concise ways to expressing the similarities and differences between the configuration of individual machines. We chose puppet as the tool for backing recipes, since it meets the other two requirements of making configurations transportable and overridable particularly well.

The actual setup and configuration of an appliance, i.e. the actual appliance functionality, is expressed as a *puppet manifest*. The manifest, written in puppet's declarative language, describes the configuration through a number of *resource definitions* that describe basic properties of the configuration, such as which packages have to be installed, what services need to be enabled, and what custom config files to deploy. Files can either be deployed as simple copies or by instantiating templates.

Resource definitions are grouped into *classes*, logical units describing a specific aspect of the configuration; for example, the definition of a class `webserver` might express that the `httpd` package must be installed, the `httpd` service must be enabled and running, and that a configuration file `foo.conf` must be deployed to `/etc/httpd/conf.d`.

The complete configuration of an actual system is made up of mapping any number of classes to that system, a node in puppet's lingo. This two-level scheme of classes and nodes is at the heart of expressing the similarities and differences between systems, where, for example, all systems at a site will have their logging subsystem configured identically, but the setup of a certain web application may only apply to a single node.

Site-specific changes to the original appliance configuration fall into two broad categories: overriding core parts of the appliance's configuration, for example, to have its webserver use a site-specific SSL certificate, and adding to the appliance's setup, without affecting its core functionality, for example, to send all syslog messages to a central server.

These two categories are mirrored by two puppet features: overrides are performed by subclassing classes defined by the appliance and substituting site-specific bits in the otherwise unchanged appliance configuration. Additions are performed by mapping additional, site-specific classes to the node describing the appliance.

The configuration part of an Appliance Recipe consists of a puppet *module*, a self-contained unit made up of the classes and supporting files. Usually, puppet is used in client/server mode, where the configuration of an entire site is stored on a central server, the *puppetmaster*, and clients receive their configuration from it. In that mode, the appliance's module is copied onto the puppetmaster when the recipe is installed.

5.2 Deploying an Appliance Recipe

Deploying an Appliance Recipe requires some infrastructure that is not needed for Appliance Images, owing to the fact that the appliance's image needs to be provisioned first. Recipes are most easily deployed using `virt-factory`, which integrates all the necessary tools on a central server. It is possible though to use a recipe without `virt-factory`'s help; all that is needed is the basic infrastructure to perform kickstart-based installs with `virt-install`, and a puppetmaster from which the virtual machine will receive its configuration once it has been booted.

In preparation for the appliance instantiation, the appliance's puppet manifest has to be added to the puppetmaster by copying it to the appropriate place on the puppetmaster's filesystem. With that in place, the instantiation is entirely driven by `virt-install`, which performs the following steps:

1. Create empty image files for the virtual machine
2. Create and boot the virtual machine and install it according to the appliance's kickstart file
3. Bootstrap the puppet client during the install
4. Reboot the virtual machine
5. Upon reboot, the puppet client connects to the puppetmaster and performs the appliance-specific configuration and installation

5.3 Creating an Appliance Recipe

For the appliance builder, creating an Appliance Recipe is slightly more involved than creating an Appliance Image. The additional effort is caused by the need to capture the appliance's configuration in a puppet manifest. The manifest can simply be written by hand, being little more than formalized install instructions.

It is more convenient though to use cft [4], a tool that records changes made to a system's configuration and produces a puppet manifest from that. Besides noticing and recording changes made to files, it also records more complex and higher-level changes such as the installation of a package, the modification of a user, or the starting and enabling of a service.

With that, the basic workflow for creating an Appliance Recipe is similar to that for an Appliance Image:

1. Write (or reuse) a kickstart file and install a base system using `virt-install`
2. Start the virtual machine with the base system
3. Log into the virtual machine and start a cft session to capture configuration changes
4. Once finished configuring the virtual machine, finish the cft session and have it generate the puppet manifest

5. Generate the appliance descriptor for the virtual machine
6. Package appliance descriptor, kickstart file, and puppet manifest into an Appliance Recipe

Note that all that gets distributed for an Appliance Recipe is a number of text files, making them considerably smaller than Appliance Images. Of course, when the user instantiates the recipe, images are created and populated; but that does not require that the user downloads large amounts of data for this one appliance, rather, they are more likely to reuse already existing local mirrors of yum repositories.

6 Example

As an example, we built a simple web calendaring appliance based on kronolith, part of the horde PHP web application framework; horde supports various storage backends, and we decided to use PostgreSQL as the storage backend.

To separate the application from its data, we created a virtual machine with two block devices: one for the application, and one for the PostgreSQL data, both based on 5GB raw image files on the host. We then installed a base system with a DHCP client, `yum` and `puppet` on it and started the virtual machine.

After logging in on the console, we started a cft session and performed the basic setup of kronolith: installing necessary packages, opening port 80 on the firewall, and starting services such as `httpd` and `postgresql`. Following kronolith's setup instructions, we created a database user and schema for it, and gave the web application permission to connect to the local PostgreSQL database server. Using a web browser we used horde's administration UI to change a few configuration options, in particular to point it at its database, and to create a sample user.

These steps gave us both an Appliance Image and an Appliance Recipe: the Appliance Image consists of the two image files, the kernel and initrd for the virtual machine, and the appliance XML description. The Appliance Recipe consists of the appliance XML description, the kickstart file for the base system, and the puppet manifest generated by the cft session.

Even an application as simple as kronolith requires a small amount of custom development to be fully functional as an appliance. For kronolith, there were two specific problems that need to be addressed: firstly, kronolith sends email alerts of upcoming appointments to users, which means that it has to be able to connect to a mailserver, and secondly, database creation is not fully scripted.

We addressed the first problem, sending email, in two different ways for the image and the recipe variant of the appliance: for the image variant, kronolith's web interface can be used to point it to an existing mail hub. Since this modifies kronolith's configuration files in `/etc`, these files need to be moved to the data disk to ensure that they are preserved during an image-based update. For the recipe case, we turned the appropriate config file into a template, requiring that the user has to fill in the name of the mail hub in a puppet manifest before deployment.

The fact that database creation is not fully scripted is not a problem for the kronolith Appliance Image, since the database is created by the appliance builder, not the user. For the recipe case, recipe instantiation has to be fully automated; in this case though, the problem is easily addressed with a short shell script that is included with the puppet manifest and run when the appliance is instantiated.

Another issue illustrates how low-level details of the appliance are connected to the environment in which it is expected to run: as the appliance is used over time, it is possible that its data disk fills up. For consumer or SMB use, it would be enough to provide a page in the web UI that shows how full the appliance's disk is—though a nontechnical user will likely lack the skill to manually expand the data disk, and therefore needs the appliance to provide mechanisms to expand the data disk. The appliance can not do that alone though, since the file backing the data disk must be expanded from outside the appliance first. For this use, the appliance tools have to provide easy-to-use mechanisms for storage management.

For enterprise users, the issue presents itself completely differently: such users in general have the necessary skills to perform simple operations such as increasing storage for the appliance's data. But they are not very well served by an indication of how full the data disk is in the appliance's UI because such mechanisms scale poorly to large numbers of machines; for the same reason, sending an email notification when the data disk becomes dangerously full is not all that useful either. What serves an enterprise user best is being able to install a monitoring agent inside the appliance. Since different sites use different monitoring systems, this is a highly site-dependent operation. With the recipe-based appliance, adding the monitoring agent and its configuration to the appliance is very simple, no different from how the same is done for other systems at the site. With the image-based appliance, whether this is possible at all depends on whether the appliance builder made it possible to gain shell access. Other factors, such as whether the appliance builder kept a package database in the appliance image and what operating system and version the appliance is based on, determine how hard it is to enable monitoring of the appliance.

Over time, and if appliances find enough widespread use, we will probably see solutions to these problems such as improved service discovery and standardized interfaces for monitoring and storage management that alleviate these issues. Whether they can cover all the use cases that require direct access to the insides of the appliance is doubtful, since in aggregate they amount to managing every aspect of a system from the outside. In any event, such solutions are neither mature nor widespread enough to make access and use of traditional management techniques unnecessary in the near future.

Details of the kronolith example, including the appliance descriptor and the commented recipe can be found on a separate website [5].

Acknowledgements

We wish to thank Daniel Berrange for many fruitful discussions, particularly around appliance descriptors, libvirt capabilities and the ins and outs of matching guests to hosts.

References

[1] `http://virt-manager.et.redhat.com/`

[2] `http://virt-factory.et.redhat.com/`

[3] `http://fedoraproject.org/wiki/StatelessLinux`

[4] `http://cft.et.redhat.com/`

[5] `http://people.redhat.com/dlutter/kronolith-appliance.html`