USENIX Association

Proceedings of the 17th Large Installation Systems Administration Conference

San Diego, CA, USA
October 26–31, 2003

Virtual Appliances for Deploying and Maintaining Software

Constantine Sapuntzakis, David Brumley, Ramesh Chandra, Nickolai Zeldovich, Jim Chow, Monica S. Lam, and Mendel Rosenblum – Stanford University

ABSTRACT

This paper attempts to address the complexity of system administration by making the labor of applying software updates independent of the number of computers on which the software is run. Complete networks of machines are packaged up as data; we refer to them as *virtual appliances*. The publisher of an appliance controls the software installed on the appliance, from the operating system to the applications, and is responsible for keeping the appliance up to date. These appliances can be configured by users to fit their needs; the configuration is captured such that it can be reapplied automatically when the appliance's software is updated. We have developed a compute utility, called the Collective, which assigns virtual appliances to hardware dynamically and automatically. By keeping software up to date, our approach prevents security break-ins due to fixed vulnerabilities.

This paper presents the concept of virtual networks of virtual appliances and describes our prototype of the Collective Utility. We demonstrate the feasibility of our approach by creating appliances for groupware servers, Windows desktop environments, and software development environments.

Introduction

On July 24, 2002, Microsoft released a patch for buffer overruns in SQL Server 2000 [11]. Six months later, on January 25, 2003, the SQL slammer worm inundated network links with packets, slowing Internet connections and costing an estimated $1 billion. The worm exploited a vulnerability on unpatched servers [19]. Unpatched software affects more than just services; desktop systems are also in jeopardy when security patches go unapplied. On June 5, 2003, Stanford University disabled all outgoing mail delivery due to the BugBear.B virus, which was leaking confidential documents [20]. The hole exploited by BugBear.B was fixed by Microsoft in a patch [10] issued more than two years before, but many users had not updated their desktops.

These two incidents underscore the importance of keeping systems up to date with respect to security patches. But security patches are released frequently, and end users may not be aware of patches or have the know-how to update their systems. Patching today is done through a variety of ad-hoc mechanisms; applying a patch sometimes breaks a system. To improve security, we must make updates automatic, reliable, and even mandatory.

Software update is only one of the problems facing system administrators. Setting up and maintaining a computing infrastructure requires much effort. While large organizations may have IT departments, smaller organizations, such as start-up companies and university research groups, may not have professional staff to create and manage infrastructure. With home users, the situation is even worse. They are often poorly versed in system administration and waste much time as a result.

Approach

We observe that computers do not have to be difficult to install and maintain. The TiVo personal video recorder has much of the same hardware and software as a PC, yet it automatically downloads updates and installs them, without hassling the user. Computing appliances, like the TiVo, provide a more predictable environment for software updates since users do not install software. Instead, the software installed on the appliance is controlled by the appliance vendor, who can test all the software to ensure it works together before distributing it.

Inspired by the ease of administering of appliances, we have proposed organizing software systems as *virtual appliances* in previous work [17]. A virtual appliance (VAP) is like a physical appliance but without the hardware; as such, a VAP is like software and can be shipped and stored electronically. Like the software in a real appliance, the software in the virtual appliance is written to run on top of hardware. We chose x86 hardware due to the vast amount of x86 software and hardware available. Rather than run the VAPs on bare hardware, we run them on an x86 virtual machine monitor, VMware GSX Server, to ease management. Recognizing that network management now plays a substantial part in system administration, we have extended the concept of a virtual appliance to include the network. A virtual appliance can be a network of virtual appliances, which we call a *virtual appliance network* (VAN). For example, a groupware VAP may consist of separate DNS, LDAP, web, and mail VAPs, connected by a virtual network, and protected by a firewall VAP. By bundling appliances into VANs, we amortize the cost of network administration among users of the bundle.

Appliance publishers create, publish, and update VAPs; like software publishers, they will often be organizations but may just be sophisticated individuals. Users get copies of VAPs from publishers and run them. Instead of installing software in VAPs, users acquire the features they need by getting additional VAPs.

We propose running VAPs on a compute utility. The utility automatically manages hardware, deciding which appliances run where. Appliances are not tied to specific hardware and can be moved to balance load or route around failures. Professionals, called *utility administrators*, procure and maintain the utility's hardware; they also install and maintain the utility's software.

An appliance's software is stored on virtual disks provided by the publisher; we call these disks *program disks*. The publisher controls the contents of program disks and can publish new versions to update the appliance's software. When an appliance is restarted, the utility will automatically pick up the most recent versions of the program disks unless instructed otherwise. To allow customizations and data to persist across updates, data is stored on separate *data disks* or in network storage.

By automating software update, our proposal makes software administration independent of the number of computers at a site. Not only does this reduce cost, but making software easy to update improves security, and reducing the management overhead encourages more software to be used.

Overview

This paper presents a prototype system, called the Collective, designed to support the creation, publication, execution, and update of VAPs. We also report our preliminary experiences with the system.

The Collective has three main components: a configuration language called CVL for describing VAPs, an appliance repository for publishing VAPs, and a utility for running VAPs. The Collective Virtual appliance Language (CVL), pronounced "civil," describes a VAP, including its parameters and, for VANs, the network layout. Appliance repositories allow publishers to post VAPs and to update them. Users refer to a repository using a URL when telling the Collective what appliance they wish to run. Finally, the Collective Utility manages a cluster of computers, runs VAPs, and performs updates.

The paper discusses the following in more detail:

- **Specification and implementation of VANs**. Just as virtual machines make whole computer states readily manipulable, virtual appliance networks ease the manipulation of networks of computers. We have implemented techniques for starting, stopping, and updating VANs. Using CVL, we describe how to compose appliances into VANs and how to attach VANs to other VANs.
- **Configurable and extensible VAPs**. To be reused, published appliances must be customizable to fit the needs of the user. Using parameters set in CVL files, users can configure VAPs with such details as network parameters and domain names; these parameters are passed by the utility to a VAP on boot and on update. Also, in the CVL language, a derived appliance inherits parameters from its parent and can thus automatically take advantage of changes made to the parent appliance. Still, users can override parameters from the parent in the derived appliance to customize it for the local site.
- **Update support**. The Collective helps publishers maintain users' software installations by providing a predictable update model – replacing the program disks of the appliance. Since users refer to appliances from repositories, the Collective Utility can directly consult the repository to find the most up to date versions and even prevent users from running vulnerable software if that is desired. Also, the Collective Utility can minimize downtime when updating a running VAN by restarting only the modified appliances.
- **Experiences**. This paper also reports our preliminary experience with the system in three scenarios: (1) a network of common group services such as DNS, LDAP and Mail, (2) Linux software development environments, and (3) Windows desktops. Our experience suggests that VAPs are a feasible way of maintaining software.

Paper Organization

The rest of the paper is organized as follows. The next section motivates our design with some examples. Then, we explain how we specify a virtual appliance network. Subsequently, we present the design of the appliance repository and an overview of the interface of the Collective Utility and its implementation. After that, we describe our experiences and related work. Finally, we present our closing remarks and conclusions.

Motivating Examples

Virtual appliances reduce system administration by having one organization, the appliance publisher, manage the software for all users of an appliance. Because the publisher controls the operating system, shared libraries, and applications in the appliance, the publisher can test them and ensure they work together.

Networks of virtual appliances have performance, isolation, and maintenance benefits over individual appliances. The performance benefits come from being able to run multiple appliances on multiple computers in parallel. The isolation benefits come from separating services: for example, firewall, LDAP, DNS, and mail servers can each have their own appliance. Updating and rebooting an appliance affects a single service or, if the service is replicated, a fraction thereof.

The maintenance benefits come from two sources. First, the cost borne by the publisher when maintaining an appliance is amortized over all users of the same appliance. Second, since a virtual appliance network can specify topology and network infrastructure services like DHCP and DNS, the user does not need to deal with the complexity of setting up networks.

We illustrate the use of virtual appliances with three concrete scenarios:

- **Groupware.** Many organizations need groupware tools to collaborate. Groups often find it difficult to create and keep their groupware up to date, especially in non-technical organizations. As a result, many departments that would benefit from their own groupware go without for lack of administrators.

  Virtual appliance networks make it possible to bundle together a number of common services, such as DNS, LDAP, and Mail, and release them as a unit. In keeping with security best practices [12], which suggest that each service run on a separate operating system, these functions are split across multiple appliances connected by a virtual network. With the Collective system, we can instantiate a new network of groupware appliances quickly and keep them up to date.
- **Software development.** Some software systems are difficult to compile, link, and install. They have been well tested only on specific versions of tools and are picky about where libraries and files are placed. It is time-consuming to track down and install the tools necessary for a complete build.

  Instead of creating an installer, the software publisher can bundle the necessary tools in an appliance and distribute it to users. Each user runs a copy of the appliance and uses the tools. To share data between the appliance and other systems, the user mounts a network file system into the appliance.
- **Telecommuters' desktops.** The advantages of appliance and utility computing can also be applied to desktop computing. In this scenario, the IT department gives a user a configured network of appliances that the user can run at home. For example, a standard office worker may have a productivity appliance with an office suite and e-mail tools. To allow the user to access company resources from home, the company bundles a VPN appliance with the productivity appliance. The VPN appliance also has a firewall to protect the productivity appliance and the company data on it from being attacked by other appliances.

Virtual Appliance Networks

We use the term *virtual appliance* (VAP) to refer to either a virtual machine (VM) appliance or a virtual appliance network (VAN) composed of VAPs. To support customization, VAPs are configurable; the behavior of a virtual appliance can be changed by changing the values of parameters.

The interface between a VAP and the Collective Utility takes the form of a *pre-defined set of parameters*. These parameters are either used by, or set by, the Utility. For example, for each VM appliance, the Collective needs to know the name of the VMware `.vmx` configuration file. For each VAN, the Collective needs to know the network topology, the appliances connected to the network, and the dependencies between them. The Collective can also assign values to parameters. After allocating certain resources, such as IP addresses, to an appliance, the utility sets parameters on the appliance corresponding to these resources; the appliance and other appliances can use these parameters to configure themselves.

Every VAP can also have *appliance-specific parameters* that are specific to configuring the software in the appliance. While complex software packages such as OpenLDAP have tens of configuration parameters, the appliance publisher can reduce this number by providing sensible defaults, thus saving the users the time-consuming task of learning all the configuration parameters in a package. Furthermore, the publisher of a VAN can pre-configure the appliances in it to talk to each other or propagate parameters between them, freeing the user from having to manually create these connections or propagate values.

An appliance specification includes a set of parameters and values. Parameters can be set by appliance publishers, the user, and the Collective Utility. A publisher may wish to compose a network of appliances out of other published appliances. Or, he may extend an existing appliance by assigning specific values to some of the parameters and republish it. For example, a university system administrator may inject the university domain name into an appliance and make it available to all users. A user may further extend the appliance by, for example, supplying the appliance with credentials in order to gain access to their data. Updates should propagate down a chain of publishers while maintaining customization; if the original publisher updates the appliance, the extended ones should update automatically, maintaining customizations wherever possible.

It is thus desirable that our configuration language support composition, extension, and allow changes to a VAP to be propagated to extended versions. This argues for a configuration language that supports abstraction and inheritance. To satisfy these requirements, we have defined the Collective Virtual appliance Language (CVL).

The CVL Language

The CVL language, version 0.8, has a generic syntax suitable for describing configurations of any types of objects and a set of pre-defined objects that model the semantics of virtual appliances.

An object may consist of component objects, a set of parameters, and possibly their values. An object can inherit from one other object. The value of a parameter is set using an assignment statement. Assignments in parent objects are executed before assignment in the derived objects, allowing specialized assignments to override the generic. Without looping constructs or even conditional statements, the language is far from being Turing-complete. It is a simple configuration language whose goal is to generate parameter and value pairs for each object. For reference, the BNF grammar for the CVL language is shown in Appendix A.

The semantics of virtual appliances are captured by four pre-defined types of objects:

- **Interface** objects represent virtual Ethernet network interfaces in VAPs.
- **Appliance** is the base object for all appliances.
- **VMAppliance**, inheriting from Appliance, is the base object for VM appliances. VMAppliance has a vm parameter that points to the contents of the virtual machine, which is a .vmx file in the case of a VMware virtual machine.
- **VANAppliance**, also inheriting from Appliance, is the base object for VAN appliances.

```
Interface {
  var "required" mac, ip, subnet, netmask;
  var defaultroute;
}

Appliance {
  var requires, provides;
  var "required" vanIF;
}

VMAppliance extends Appliance {
  var "required" vm;
  var datadisks;
  Interface ethernet0;
  vanIF = "ethernet0";
}

VANAppliance extends Appliance {
  var defaultroute;
}
```

Figure 1: Pre-defined objects in CVL.

Figure 1 shows all of the parameters defined for each of the base objects. The semantics of these parameters are discussed in more detail below. The pre-defined objects and their parameters are used by the Collective Utility in configuring and running virtual appliances. We can set the values of these parameters and add new appliance-specific parameters by deriving new appliances from either VMAppliance or VANAppliance. Only VANAppliances can have VMAppliance components; only VMAppliances can have Interface components. Currently, CVL does not allow the definition of any other kinds of objects.

From http://virtualappliance.org/DNS:

```
/* Language version number */
CVL = "0.8";
DNS extends VMAppliance {
  var "required" domain, dnshosts;
  var port;
  port = "53/udp";

  /* Virtual machine configuration */
  vm = "dns.vmx";
  datadisks = { device => "ide0:1",
                size   => "100mb" };
  /* Dependencies between appliances */
  provides = "DNS";
}
```

From http://virtualappliance.org/OpenLDAP:

```
CVL = "0.8";
OpenLDAP extends VMAppliance {
  var port, sport;
  port = "389/tcp";
  sport = "636/tcp";

  /* Virtual machine configuration */
  vm = "ldap.vmx";
  datadisks = { device => "ide0:1",
                size   => "100mb" };

  /* Dependencies between appliances */
  provides = "LDAP";
  requires = "DNS";
}
```

From http://virtualappliance.org/Firewall:

```
CVL = "0.8";
Firewall extends VMAppliance {
  Interface ethernet1;
  var services;

  /* Virtual machine configuration */
  vm = "fw.vmx";
}
```

Figure 2: A few virtual appliances used in Groupware.

Figures 2 and 3 show an example of a groupware network called Groupware with three components, a DNS server, an LDAP server, and a firewall. We will use this example in the rest of the section to explain the CVL language. We first describe how to specify components and how to declare parameters and assign to them, and then describe the semantics of the parameters in the pre-defined base types.

Components

Component VAPs of a VAN must be *imported* into the name space of the file defining the VAN. The import statement specifies the appliance definition to be used and a short name by which the definition is referred to. The definition includes the URL of the appliance repository, the name of the .cvl file, and optionally a version number. If no version number is specified, the latest version is assumed.

Specifying a particular version of an appliance is useful in cases when only that specific appliance version supports some feature. Additionally, a specific version of an appliance may be desired when a set of appliances of certain versions have been tested to work together, as is the case of the Groupware appliance. Most of the time we expect the user not to specify a particular appliance version when starting an appliance, thus allowing the

appliance software to be automatically updated as new versions become available.

From http://virtualappliance.org/Groupware:

```
/* Language version number */
CVL = "0.8";
/* Import component appliance definitions */
import {
  url => "http://virtualappliance.org/DNS",
  cvl => "DNS.cvl",
  version => "3"
} DNS;

import {
  url => "http://virtualappliance.org/Firewall",
  cvl => "Firewall.cvl",
  version => "5"
} Firewall;

import {
  url => "http://virtualappliance.org/OpenLDAP",
  cvl => "OpenLDAP.cvl"
} OpenLDAP;

Groupware extends VANAppliance {
  var "required" domain;

  /* components */
  DNS d;
  OpenLDAP l;
  Firewall f;

  /* configuration */
  d.domain = domain;
  l.domain = domain;

  d.dnshosts = { name => l.name,
                 ip   => l.ethernet0.ip },
               { name => d.name,
                 ip   => d.ethernet0.ip },
               { name => f.name,
                 ip   => f.ethernet0.ip };
  f.services = { port => l.port,
                 ip   => l.ethernet0.ip },
               { port => l.sport,
                 ip   => l.ethernet0.ip },
               { port => d.port,
                 ip   => d.ethernet0.ip };

  /* network topology */
  vanIF = "f.ethernet1";
  defaultroute = f.ethernet0.ip;
}
```

Figure 3: A virtual appliance network for running Groupware.

Components in VANs are declared by specifying the appliance type followed by the name of the component. For example, the Groupware appliance in Figure 3 declares that it has three components: a DNS server named d, an OpenLDAP server named l, and a firewall named f. The import statement for the DNS appliance specifically requests version 3 of the appliance from the repository, while the import statement for the Firewall appliance does not specify the version number, so the latest version present in the repository will be used.

Parameters and Assignments

Each appliance inherits all the parameters from its parent appliance and can define new parameters. It may assign to its parameters or the parameters defined in any of its components. We refer to parameter v of an object o as o.v.

Parameters may be given attributes. Parameters declared with the attribute "required" must have a value before the appliance can be started. With this construct, the Collective Utility can detect errors early and help users by providing them with a meaningful error report. As shown in Figure 2, the OpenLDAP appliance has a required domain parameter, which requires the user to set the domain name for the LDAP server. The Firewall appliance, on the other hand, has an optional services parameter, which allows the user to specify what services the firewall should expose. If the value of the services parameter is not specified, the firewall appliance will still function without exposing any services.

Some parameters, like user keys and passwords, are sensitive; we declare such parameters with the attribute "sensitive". Sensitive parameter values should not be stored unencrypted in the file system. Instead, before sending parameters to an appliance, the Collective Utility passes them to a user agent service, which keeps a cache of sensitive values. If the requested parameter is not present in the cache, the agent prompts the user for the value. To allow sensitive data to persist across instances of the user agent, the user agent stores the data in a file encrypted with a user-supplied password.

Parameter values in CVL are lists of one or more strings. Each list element can be either a quoted string constant, a parameter, or a map. A map is a set of key-value pairs. The map notation eliminates the need to remember the ordering of values, and makes the meaning apparent. Map keys are string constants, and values can be string constants, other parameters, or even other maps. A map is just syntactic sugar for defining a string: for example, the map {a=>"b", c=>"d"} evaluates to "a=b&c=d". In the string representation of a map, non-alphanumeric characters in the keys and values are escaped, allowing for recursive map structures.

Let us now look at how the configuration parameters in the Groupware VAN are defined. The domain name, to be specified by the user for the entire network, is passed onto both the DNS and OpenLDAP appliances. The firewall appliance has a services parameter, which specifies the services that the firewall should allow and hosts to which those services should be forwarded. The Groupware VAN declaration puts the addresses and ports of the DNS and LDAP appliances into this parameter, via a map, in order to expose those services. This example illustrates how a VAN may have fewer parameters than the sum total provided by its components. The components may share common parameters, and values from one appliance can be used to configure another.

Disks in VM Appliances

Users are not allowed to make changes to the installed software in VM appliances, but of course must be able to modify their data. When updating a VAP, we must preserve the user's data. Our solution is to store user data either outside of the appliance by

using a network file system or on a separate disk dedicated to storing user data in the appliance. Appliances that need to access existing user files would likely want to use a network file system for user data. Service appliances, or appliances whose data is of no use outside of the appliance, would likely opt for the second option of a dedicated data disk in the appliance, as it introduces fewer dependencies.

Each virtual disk in a VM appliance is used either for storing appliance software or for storing user data. Disks storing appliance software are called *program disks*; they define the operation of the appliance. Disks storing user data are called *data disks*. All data disks, each described by a device name and an initial disk size, must be listed in the datadisks parameter inherited from VMAppliance. For example, Figure 2 illustrates a DNS appliance with a 100 megabyte data disk as device ide0:1; the data disk stores zone files for the appliance.

When updating an appliance, contents of the program disks are updated, but data disks are untouched. This allows the user's personal settings and data to persist across updates of the appliance.

Network Topology

A VANAppliance allows one or more appliances to be grouped into a single VAN appliance. This results in a strictly hierarchical network of virtual appliances. More general topologies are supported by CVL but, for simplicity, are omitted from this paper.

All components of a VAN are connected to the same virtual Ethernet network, which can be attached, via a gateway appliance, to another virtual network or the Internet. The gateway typically implements firewall, routing and NAT functionality.

The Interface ethernet0 declaration in VMAppliance guarantees that every VM appliance has a virtual interface. It is possible for a VM appliance to declare additional interfaces. The Collective Utility is responsible for assigning MAC and IP addresses to each of the interfaces in a VM appliance. Components in a VAN are connected to the same Ethernet segment via their VAN network interfaces, specified by the vanIF variable.

A VAN specification wishing to export a network interface must set the vanIF variable to an interface of one of its constituent appliances. For example, the Firewall appliance in Figure 2 has two interfaces, ethernet0 and ethernet1. Its ethernet1 interface, f.ethernet1, serves as Groupware's network interface. This allows the Groupware appliance to be connected as an appliance to another network (such as another VAN or the outside world). Groupware's defaultroute is set to f.ethernet0.ip so that all packets from Groupware's appliances are routed through the firewall.

Dependencies Between Appliances

Just as virtual machines can be started and stopped, so can VANs. Because services have dependencies, we have to start and stop them in a specific order. Appliance publishers list the services that their appliance provides and the services that their appliance needs for its operation. The Collective uses this information to construct a boot order and a shutdown order.

Every appliance inherits two variables from the Appliance object: provides and requires. These variables contain a list of strings representing the services provided or required by this appliance, respectively. For example, in Figure 2, the DNS appliance sets the provides variable to "DNS", and the OpenLDAP appliance sets the requires variable to "DNS". A VAN containing these two appliances would start the DNS appliance before the OpenLDAP appliance.

Repositories

A repository provides a location where a publisher can post successive versions of an appliance and users can find them. This section explains what repositories are and how they are used by publishers, users, and the Collective Utility.

Each Collective appliance repository holds the versions of a single appliance; the versions are numbered using integers starting from 1. Once a version has been written to the repository, that version becomes immutable. Each version of an appliance has a CVL file. For VM appliances, the VMware virtual machine files (.vmx, .vmdk, and .vmss) are also stored. To save time, disk space, and bandwidth, the virtual disks typically contain only the changes from the previous version of the appliance.

Publishers create and update repositories through the UNIX Collective User Interface command (cui for short). The publisher runs the command

```
cui create <repository>
```

to create an empty repository at the file path repository. The publish operation

```
cui publish <repository> <cvl>
```

stores the files representing a virtual appliance as the latest version of the appliance in the repository. For all appliances, this involves copying the CVL file into the repository. For a VM appliance, the VMware configuration file contains a list of all virtual disks comprising the VM appliance, and the CVL file designates some of the virtual disks as data disks. Virtual disks not designated as data disks are assumed to be program disks. The publish operation copies the contents of the program disks to the repository but does not copy the contents of data disks. This means that an appliance repository only contains appliance software and does not store any data disk content.

A repository can be hosted anywhere in the file system where a user can create a subdirectory. We access and store our repositories through SFS [9], which provides a secure access to a global namespace of files.

The Collective Utility

The Collective Utility manages both virtual appliances and hardware. The utility executes requests to start, stop, and update VAPs from users, and answers queries on the status of VAPs. It allocates hardware and network resources to VAPs and configures VAPs to use those resources.

The Collective Utility consists of a central cluster manager service and a host manager service for each computer in the cluster. The cluster manager accepts appliance management requests from users, decides on the allocation of resources across the entire cluster, and interfaces with the host managers, which are responsible for executing appliances on the respective hosts. The cluster manager also keeps track of the ''truth'' in the system, including the list of physical resources in the system, the VAPs that have been started, and the resources allocated to them. This information is stored on disk to survive cluster manager restarts.

The utility administrator is responsible for registering all the resources in the system with the cluster manager, so that it can keep track of the available resources and perform resource allocation to appliances. Using a command line tool, the administrator can register a host, specifying its resources – memory size and the maximum number of VMs hosted at any one time. In our prototype, we require each registered host have Red Hat Linux 9, VMware GSX Server 2.5.0, and the Collective software installed. The administrator also registers a set of VLAN numbers and public IP addresses with the cluster manager. These VLAN numbers and public IP addresses are assigned to virtual networks and to network interfaces connected to the public Internet.

The utility administrator can restrict the appliances the utility runs by providing it with a *blacklist* of repositories and versions that should not be run. For example, the administrator may wish to place all appliances with known security vulnerabilities on the list. The utility will not start new appliances that use versions on the blacklist. However, the utility will not stop already running appliances that violate the blacklist; instead, the administrator can query the utility for these appliances. The administrator can then either ask users to update or can forcibly stop the appliances.

Before using the utility, the user must first create a new appliance by creating a CVL file that inherits from the appliance to be run. As part of writing that CVL file, the user sets the values of the parameters of interest. The following is an example of a CVL file created by a user for the Groupware appliance:

```
import {
  url => "http://virtualappliance.org/Groupware",
  cvl => "Groupware.cvl",
} Groupware;

AcmeGroupware extends Groupware {
  domain = "acme.org";
}
```

The user can then use the utility to start, stop, and update the appliance. Below, we describe each of the available user commands in more detail and overview their implementation.

Starting a VAP

The command cui start <CVL> starts the appliance as specified in the <CVL> file. We first discuss how we handle virtual networks and then describe the implementation of the command.

Virtual Networks

Our design allows the component VM appliances in a VAN be run on one or more machines. Each running VAN has its own Ethernet segment, implemented as a VLAN (Virtual Local Area Network) on the physical Ethernet. All VM component appliances of a VAN on each host are connected to a dedicated VMware virtual switch on the host, which is bridged to the VAN's VLAN. Physical Ethernet switches that recognize and manage VLANs may need to be configured to pass traffic with certain VLAN tags to certain hosts. Since our experimental setup uses switches that ignore VLAN tags, no configuration is required.

The Collective also takes over the chore of assigning IP addresses to appliances. Each VAN is assigned a subnet in the 10.0.0.0/8 site-local IP address range, with the second and third octets of the address derived from the VLAN tag. So, each VAN has 256 IP addresses. Each virtual Ethernet adapter in each VM appliance is given a unique MAC address and an IP address from the pool of VAN's IP addresses.

In the case of sub-VANs, the internal interface of a gateway on a sub-VAN is assigned an IP address from the sub-VAN's IP address space. The external interface of the gateway is assigned an IP address from the address space of the parent VAN. Exported interfaces that do not connect the VAN to another VAN are given public IP addresses from a pool of IPs.

We use network address translation (NAT) to help route traffic between VANs and their parent networks. We must use NAT between the public Internet and our VANs since we assign site-local addresses to our VANs. Even though each VAN has a distinct site-local range, we still use NAT between VANs and sub-VANs to avoid setting up routing tables. For this reason, a VAN's chokepoint appliance, such as a firewall or router, should provide NAT functionality.

Implementation

To implement the start command, the cluster manager parses the CVL file. It imports CVL files from repositories where necessary, remembering the version number. It sets up the VAP's disks and then, if it finds that the VAP's requirements can all be satisfied, brings up the VAP. Note that a VAP may be a VAN whose components may themselves be VANs. From now on, we use the term *component VM appliances* of a VAP to refer to all the VM appliances defined by a VAP, its components, its components' components and so forth.

In the first step, the cluster manager sets up the program and data disks for all the component VM appliances in the directory containing the CVL file. Every component VM appliance is given its own subdirectory. The manager creates a new copy-on-write demand-paged version for each program disk, and if a specified data disk does not already exist, an empty data disk of the size specified in the CVL file is created. The appliance is responsible for detecting an all-zero data disk on boot and initializing it appropriately.

In the second step, the cluster manager ensures that all the required services are available, required parameters set, and required resources reserved. It generates a dependency graph from the provides and requires variables of all component VM appliances, and propagates parameter values to all the CVL files. For fault tolerance reasons, the cluster manager determines which resources are available by computing all the resources currently in use by all the running VAPs. It then decides where each VM appliance is to be hosted and reserves the memory requested in the appliance's .vmx files. Next, the cluster manager reserves a VLAN for each subnet and an IP address for each VM Appliance.

In the third and final step, the cluster manager brings up the VAN. It first sets up the hierarchy of networks by instructing all participating host managers to allocate VMware virtual switches and bridge them to the appropriate VLAN. It then starts up the component VM appliances, possibly in parallel, in an order satisfying the dependencies. The VMware Scripting API [24] is used to pass a VM appliance its parameters, including the assigned MAC and IP addresses, and to power it on. As soon as an appliance signals that it has successfully started, the cluster manager starts any appliances whose dependencies are now satisfied.

Stopping a VAP

The command cui stop <CVL> [<comp>] stops the entire appliance defined in the <CVL> file if no <comp> is given, otherwise it stops only the component appliance <comp>. As in CVL, components are specified using a dot-separated path, for example sub-component f of component g is written g.f.

Stopping a virtual appliance is more straightforward than starting one. Component VM appliances are stopped in the reverse order of startup. To stop a VM appliance, the cluster manager uses the VMware Scripting API to instruct the virtual machine to stop. VMware passes the stop request to a daemon running inside the appliance, which then initiates a clean shut down. If the appliance does not shut down within three minutes, VMware forcibly terminates it.

Updating a VAP

The command cui update <CVL> [<comp>] updates the entire appliance if no <comp> argument is given, otherwise it updates just the component <comp>. To minimize disruption, we do not require that all the VM appliances be shut down to update a VAN; only the affected VM appliances are. The cluster manager automatically derives the actions necessary to update an old version to the new by finding the differences between the two.

The cluster manager re-parses the CVL file, and simulates its execution to determine what the final state of the VAN should look like. It then finds the difference between that final state and the current VAN state, and determines the list of actions to transform the current state to the desired final state. The actions include:

- Starting an appliance that is present in the final state but not present in the current VAN state.
- Removing an appliance that is present in the current VAN state and not in the final state. First, it stops the appliance and the appliance data is moved to an attic directory. This prevents conflicts with any new appliance in future updates that might be given the same name as the removed appliance.
- Updating a VM appliance if its version has changed. This involves first stopping the VM, copying over the new program disks, and restarting the appliance. If an update requires data disks be modified, the new version of the appliance should include logic that, on boot, detects whether the user data is in the old format and, if so, migrates the data to the new format.
- Resending parameters to a VM appliance, if any have changed. This is done using the VMware Scripting API. An appliance wishing to respond to changes would run a daemon that checks for changes in the parameters and reconfigures the appliance appropriately. For example, when parameters are resent to a DHCP appliance, the daemon rewrites the DHCP configuration file and restart the DHCP server.

Costs of Virtual Appliances

The benefits of isolation and ease of management obtained from using virtual appliances are not without costs. First, starting up an appliance requires booting its operating system, which takes much longer than starting a service on an already booted operating system. Note, however, that this same procedure will bring up a pristine copy of the latest software on any machine. There is no extra cost associated with provisioning a machine to run a new service, re-installing the operating system to fix an error or updating a software to a new version.

Virtual appliances also have higher disk and memory storage requirements and slower execution due to virtualization overheads. Fortunately, virtual machine technology continues to improve. Recent work on VMware ESX Server improves the utilization of physical memory by adapting the allocation of physical pages dynamically and sharing identical memory pages across virtual machines [25]. Our previous work has shown

that demand paging and copy-on-write disks significantly decrease the cost of storing and sending updates using disk images [16]. Finally, it should be noted that our approach is not limited to virtual machines; with SAN adapters and appropriate management support from the computer's firmware, virtual appliances could be booted on raw hardware.

Experimental Results

We describe using the Collective system to build the following appliances: a Groupware virtual appliance network, a software development appliance, and Windows desktop appliances.

The Groupware Appliance

We created a groupware virtual appliance that provides a base Internet infrastructure for a small collaborating group. The plone web-based content management system was built with Debian GNU/Linux 3.0, and all the other appliances were built with Red Hat Linux 8.0. Open source software was used for the services. The software packages for the services and the versions used are shown in Figure 4.

Configuration and Deployment

Since the groupware bundles together mostly-configured appliances, this reduces the amount of configuration effort necessary for each instantiation. With the goal of building an infrastructure for a small group, we were able to set most software configuration options to reasonable defaults. For example, even though the OpenLDAP and Mail (Postfix and Courier) applications have a large number of configuration options, the corresponding appliances in groupware need only 12 and 9 parameters, respectively. In addition, some appliances share the same parameters. As a result, although a sum total of 30 parameters need to be set individually in the groupware appliances, deploying the whole suite of groupware services requires setting only 14 parameters. Figure 5 gives a complete list of all the parameters for the groupware appliances, along with a short description of each of them. The variables and their relationships are all specified in the CVL file, excerpts of which are shown in Figures 2 and 3.

The appliances use these parameters to configure themselves. Configuration works especially well for

Appliance	Software Package	Function	Size	Depends on		
				DHCP	DNS	LDAP
Firewall	iptables-1.2.6a	Forwards traffic only to group services	495 MB	---	---	---
DHCP	bind-9.2.1	Provides IP addresses to appliances	501 MB	---	---	---
DNS	dhcp-3.0pl1	Serves names of group services	501 MB	Yes	---	---
LDAP	openldap-2.0.25	Authentication and user database	578 MB	Yes	Yes	---
Mail	postfix-1.1.11 courier-0.42.2	SMTP, IMAP, and POP server	627 MB	Yes	Yes	Yes
Plone	plone-1.0.2	Web-based content management system	441 MB	Yes	Yes	---

Figure 4: Properties of appliances in the Groupware appliance.

Parameter Name	Appliances Using Parameter	Parameter Description
hostname	DHCP, DNS	Used to auto-populate the dhcpd.conf and DNS zone files.
domain	DHCP, DNS, LDAP	The DHCP server uses the domain parameter to set the proper domain name of the appliance. The LDAP server derives the root DN from the domain name.
rootdnpass	LDAP	The root DN password.
proxypw	LDAP, Mail	A proxy user is used for LDAP client authentication. This is the proxy user's password.
smtppw	LDAP, Mail	The smtppw is the password of the user who can read a user's Maildir parameter in the LDAP server.
country, state, city, org, ou, certemail	LDAP, Mail	These parameters are used to create X.509 certificates.
defmailhost	LDAP	The default mail host the SMTP server should deliver mail to.
forwarders	DNS	DNS caches to forward queries to.
rootpw	All appliances	The Unix root password.

Figure 5: Parameters of the Groupware appliance.

UNIX, where most configuration is done using text files and it is easy to interpose on the boot sequence. In our appliances, the boot sequence is modified to read in the appliance parameters and generate configuration files based on them, before appliance software is started. Typically, these configuration files are generated by using a template and filling in values based on appliance parameters.

For example, the firewall appliance has a parameter for specifying the ports and addresses of all the groupware services, and it configures itself to accept traffic only to those services. The DHCP appliance has parameters for specifying the MAC addresses, IP addresses, and hostnames of all the groupware appliances, and it transforms these parameters into a DHCP server configuration file. After rewriting the DHCP configuration file, the appliance restarts the DHCP server. Other appliances configure themselves in a similar fashion.

Characteristics of Groupware

As shown in Figure 4, the sizes of all the VM appliances are around 500 MB. This is large compared to the sizes of the services themselves. However, since we use SFS, during an appliance start-up only the required parts of the appliance are demand paged in. Furthermore, techniques exist (see the previous section) for decreasing appliance disk space requirements.

We timed the groupware start-up and shut-down on five 2.4 GHz Pentium 4 client machines with 1 GB of RAM each. The program and data disks were stored on a sixth 2.4 GHz Pentium 4 machine which acted as the SFS server. The clients and server were connected via 100 Mbps Ethernet.

The start-up time was 6.3 minutes with cold caches at the SFS server and clients and averaged 5.0 minutes over three runs with a warm cache. The shut-down time averaged 2.2 minutes over three runs. During start-up, the cluster manager took on average 1.1 minutes to set up appliance disks, reserve resources, and set up virtual networks. The remainder of the time is spent starting up the component VM appliances. The Red Hat 8.0-based component appliances took on average 1.8 minutes each to start before optimization. After removing unnecessary services from start, the appliances took about one minute to start. Currently, of that minute, the network start script takes 20 seconds; by taking advantage of the virtual nature of the network interface, we believe it can be optimized to under one second.

With further optimizations, appliances with smaller disk space and boot times could be built. For example, the floppyfw Linux distribution [22] combines both DHCP and firewall functionalities in a single 1.44 MB floppy disk that boots in less than 20 seconds in a virtual machine.

Software Development Environments

Software development today typically relies on a large number of tools and often specific versions of those tools. Acquiring all the tools required to develop and build a piece of software can take a lot of time. Appliances allow complex software to be used immediately and easily by the user by bundling all required tools and libraries into the appliance. We expect this will encourage more users to experiment with software and to participate in its development; for example, an appliance made up of free software could include the source and tool chain so that users could fix bugs and add features.

We have created an appliance for the SUIF research compiler infrastructure [1], a system that has been used by research teams both in and out of Stanford. Over the years, new researchers have spent hours setting up their environment to build the system before they could start experimenting with it. For example, SUIF is compiled and tested using the GCC 2.95 compiler and header files; users of Red Hat Linux 8 and 9 use GCC 3, and users of these systems would need to install the older compiler and headers. Even then, they would probably need to modify the build process to point SUIF to the alternate compiler and headers. In sum, we can avoid a lot of trivia by distributing the entire tool chain as a unit, which we did when we made SUIF into an appliance.

Users of the SUIF appliance and other desktop appliances may wish to mount their user files. We expect that users may want to access their existing files from the SUIF appliance, and therefore the SUIF appliance stores user files on a network file system rather than on a data disk. However, we feel it would be cumbersome if the user had to repeatedly enter their username, location of their files, and password into each appliance.

To mount user files into an appliance, we give the appliance a list of ways to mount the user files. Each way includes a URL and credentials; the URL tells the appliance how and what to mount and the credentials tell it who to mount as. The list is ordered. As the appliance traverses the list, it checks if it supports the protocol indicated in the URL. If it does support the protocol, it looks at the credentials. If it supports the authentication method indicated in the credentials, it tries to mount the file system. If it succeeds, it sets the home directory in the user record to the mounted file system. If it fails, it tries the next way of mounting the files.

Others have grappled with the problem of mounting user files across multiple operating systems. In Windows networks, *domains* authenticate users and provide roaming profiles; many other systems have similar notions. Recognizing the diversity in today's network sites, rather than forcing any specific system, we try to configure appliances with user files in as generic a fashion as possible.

Windows Appliances

To investigate the feasibility of Windows virtual appliances, we created virtual appliances reflecting common Windows desktop environments. The first

subsection describes the base of software we used in our experiments. While building these appliances, we had to deal with the peculiarities of Windows. In particular, we take a different approach (described in the second subsection) to configuring a Windows appliance since system properties, like the computer's *Security Identifier* (SID), do not reside in simple text configuration files as on Linux. Additionally, updating an appliance requires customizations be stored separately from programs, and this is difficult to enforce on Windows, but in the final subsection we observe that most applications partition themselves nicely.

The DesktopNet Appliance

We created two Windows virtual appliances; each virtual appliance runs a single application. We created one for Office 2000 and another for Internet Explorer 6, both running Windows 2000 Professional. To share program data and configuration settings between programs in different appliances, all appliances are configured to use roaming profiles and mount network shares from a central Samba server.

The Samba appliance runs Samba 2.2 on Redhat 8.0. It is configured to serve profile data and network shares off of its data disk, which starts off empty. Also, to add a user, the administrator must currently log into the Samba appliance and add the account manually. Alternatively, the Samba appliance could be configured to require an LDAP appliance for authenticating accounts.

Since Samba allows program data and configuration to live outside of the appliance, desktop applications generally require little direct configuration. The only required configuration variables for the desktop appliances are for making it talk to the Samba server. For example, Samba requires credentials for both the user (to access their files) and the machine (to join/authenticate to the domain). Appliances receive domain configuration (domain name and domain user/password) on-the-fly from the CVL specification, and users must log in to their desktop appliances before using them.

Configuring a Windows-Based Appliance

Under UNIX, appliance configuration involves taking in parameters supplied by the Collective and rewriting text configuration files residing in the appliance. Under Windows, however, changing such settings may require the modification of undocumented registry entries and a reboot of the appliance. Therefore we use published and tested tools where possible; our current approach for configuring appliances uses Microsoft's System Preparation Tool sysprep. Parameters configured through the appliance's CVL file are propagated to a script running in the appliance on boot, which prepares a sysprep unattended install file (sysprep.inf) with the passed-in parameters, and then initiates sysprep in the appliance. This requires a series of reboots to get the appliance into the appropriate state, but using sysprep guarantees that Windows-specific machine IDs are properly changed in each copy of the appliance.

Updating a Windows-Based Appliance

Normally, updating an appliance means replacing the appliance's program disks with updated ones. As long as data and user configuration reside on a separate data disk, no data is lost in this update process. Unfortunately, some Windows applications' use of the file system and registry can make separating program state onto a dedicated data disk difficult – any data stored under the application's Program Files directory or in system-wide registry entries could get blown away during an update.

Fortunately, many Windows applications store per-user configuration in the user's roaming profile directory (either through special per-user registry sections, which get backed by the roaming profile, or in the user's Documents and Settings folder, which also resides in the user's roaming profile). This behavior is actually mandated by Microsoft's Logo certification program so that roaming profiles work, and we have observed correct behavior in practice by doing updates on our desktop appliances.

For example, we made user customizations in Office 2000 (e.g., adding keyboard macros, dictionary entries, etc.) and updated from Office 2000 to Office 2000 Service Pack 3 by completely replacing the program disk; we observed that our user customizations remained intact. Likewise, we made similar customizations to Internet Explorer 5 (e.g., bookmarks, proxy settings, homepage, toolbar settings, etc.) and observed that these too were recognized after a complete overhaul of the publisher disk to Internet Explorer 6. Thus, although Windows applications can be more difficult to coerce into using particular parts of the file system, many turn out to be well-behaved on their own.

As illustrated by the experiences described in this section, we have encountered challenges in both configuring and updating Windows appliances. And although they present interesting problems, we have demonstrated techniques for turning Windows applications into parameterized virtual appliances.

Related Work

Package tools like Depot [5] and Debian's apt simplify the maintenance of individual software packages. Unlike most packages, appliances bundle groups of applications and provide a unified configuration interface. Also, virtual appliances running on the same computer are better isolated from each other than packages installed on the same computer. Tools like Sasify [13, 18], as well as the aforementioned package tools, simplify the task of keeping up with patches and software updates. Virtual appliances update the entire system, thus providing more assurance in the way of overall system testing at each version.

Like the Collective, Sun's N1 [21], HP's Utility Data Centre [8], and VMware's Control Center [23]

aim to automate managing hardware and software. The customer of the utility provides a higher-level description of the services they want to run and the performance they want from those services, and the utility decides how many servers to instantiate and what hardware to assign them to. The utility dynamically monitors load and can reassign computers from serving one service to another. The utility can also work around failures of hardware by booting a new instance of the service on good hardware.

Grid computing, of which Globus [7] is the leading example, automates the harnessing of thousands of computers across the world for running scientific code. The grid software for managing processes on thousands of machines may be useful for non-scientific code too. As a result, it seems that grid and utility computing are merging in some parts; Globus has recently expanded to web services.

Our research is focused on reducing software system administration cost by amortizing the configuration, installation, and update efforts over a large number of users. This led to the development of VANs, a way of distributing software that can be configured to suit individuals' needs and can be updated automatically.

Web application servers, like JBoss and Websphere, automate the deployment and management of services written in Java across a cluster of machines. They require software to be written to Java APIs. In contrast, our approach of using virtual x86 machines can be applied to most existing software.

Storage-area networks (SANs) and disk imaging [3, 14] have been used for years to reliably configure computers. Virtual appliances use disk imaging to predictably deliver updates to virtual disks. VMMs allow us to implement SAN-like capabilities in software without modifying hardware or the guest OS.

CFengine [4] is a tool for configuring operating system images from a central description. Like CFengine, CVL strives to describe what the state of a network of machines should be rather than describe the steps for configuring the network. Unlike CFengine, CVL passes information to appliances through a generic key-value pair interface and leaves the details of configuring individual nodes in the network to the appliance publishers.

Like the Desktop Management Task Force's (DMTF's) Common Information Model (CIM) [6] and SNMP MIBs [15], CVL describes objects with sets of properties. Unlike CIM and SNMP, the focus of CVL is not returning status and statistics but configuring objects, specifically networks of virtual appliances. To reduce duplication of values when configuring objects and to provide intelligent defaults, CVL has constructs for propagating one value to many parameters.

Conclusion

This paper develops the concept of virtual networks of virtual appliances as a means to reduce the cost of deploying and maintaining software. We have presented a language for specifying virtual appliances and algorithms for implementing them. The language is designed to allow user customization while supporting automatic updates. The Collective prototype we developed assigns virtual appliances to hardware in the system automatically and dynamically. The prototype uses repositories to find up-to-date versions of appliances.

We have shown how VAPs can be used to create a complete Groupware appliance that can be instantiated anywhere, a software development appliance that reduces the overhead associated with working on new software, and discussed how the approach can be used to create Windows-based appliances.

Our approach makes the management of software independent of the number of computers running the software. By placing the burden of maintenance on the appliance publisher, this approach makes it easier for users to run new computer software and to keep their systems up to date. The Collective Utility can even be used to disallow the execution of vulnerable software. This would eliminate incidents like the one where Microsoft itself was compromised by the Slammer worm when it failed to locate all vulnerable servers [2].

For information about releases of the Collective, please visit our web page at http://collective.stanford.edu/ .

Acknowledgements

This material is based upon work supported in part by the National Science Foundation under Grant No. 0121481 and Stanford Graduate Fellowships. We thank our shepherd Gerald Carter, Satoshi Uchino, and Will Robinson for their comments on the paper.

About the Authors

Constantine Sapuntzakis is currently pursuing a Ph.D. in Computer Science. To avoid the stresses of grad school, he likes to play system administrator for his research group and at home. He wants to make computer systems easier to use.

David Brumley is a Ph.D. student in Computer Science at Carnegie Mellon University. Previously, he was the computer security officer for Stanford University, where he responded to over 1000 incidents and authored such programs as the remote intrusion detector (RID) and SULinux (Stanford University Linux). David received his Bachelor's degree in Mathematics from the University of Northern Colorado and his Master's degree in Computer Science from Stanford.

Ramesh Chandra is a Ph.D. candidate in Computer Science at Stanford University. He received his B.Tech from the Indian Institute of Technology, Madras, India, and his M.S. from the University of Illinois at Urbana-Champaign, both in Computer Science. He is interested in systems research in general, and in particular operating systems, networking, and distributed systems.

Nickolai Zeldovich is a Ph.D. student in Computer Science at Stanford. He received his Bachelor's and Master's in Computer Science from MIT in 2002. In his spare time, he likes to windsurf, hack on OpenAFS, and collect old computer equipment.

Jim Chow is currently a Ph.D. student in Computer Science at Stanford University. His interests include operating systems and systems security. He graduated in 2000 with a BS in Electrical Engineering and Computer Science from UC Berkeley.

Monica Lam is a Professor of Computer Science at Stanford. She received a Ph.D from Carnegie Mellon University in 1987 and a B.S. from University of British Columbia in 1980. Her current research interests are in improving software productivity and usability of computers. Honors for her research include an NSF Young Investigator Award and an ACM Most Influential Programming Language Design and Implementation Paper Award.

Mendel Rosenblum is an Associate Professor of Computer Science at Stanford. His contributions to the field of systems, including the Log-structured File System, SimOS, Hive, Disco, and VMware, earned him the ACM SIGOPS Mark Weiser award in 2002.

References

[1] Aigner, G., A. Diwan, D. Heine, M. S. Lam, D. Moore, B. Murphy, and C. Sapuntzakis, *An Overview of the SUIF2 compiler Infrastructure*, http://suif.stanford.edu/suif/suif2/doc-2.2.0-4 .

[2] Associated Press, *Microsoft also gets slammed by worm*, http://www.cnn.com/2003/TECH/biztech/01/28/microsoft.worm.ap/ , January, 2003.

[3] Barnett, T., K. McPeek, L. S. Lile, and J. Ray Hyatt, ''A web-based backup/restore method for Intel-based PC's,'' *Proceedings of the 11th LISA Conference*, pp. 71-78, October, 1997.

[4] Burgess, M., ''A Site Configuration Engine,'' *USENIX Computing Systems*, Vol. 8, Num. 2, pp. 309-337, 1995.

[5] Colyer, W. and W. Wong, ''Depot: A tool for managing software environments,'' *Proceedings of the 6th LISA Conference*, pp. 153-162, October, 1992.

[6] *Desktop Management Task Force – Common Information Model*, http://www.dmtf.org/standards/standard_cim.php .

[7] Foster, I., C. Kesselman, J. M. Nick, and S. Tuecke, ''Grid Services for Distributed System Integration,'' *IEEE Computer*, Vol. 35, Num. 6, pp. 37-46, 2002.

[8] *HP Utility Data Center*, http://www.hp.com/solutions1/infrastructure/solutions/utilitydata/ .

[9] Mazières, D., M. Kaminsky, M. F. Kaashoek, and E. Witchel, ''Separating key management from file system security,'' *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, Kiawah Island, South Carolina, December, 1999.

[10] *Microsoft Security Bulletin MS01-020: Incorrect MIME header can cause IE to execute attachment*, March, 2001.

[11] *Microsoft Security Bulletin MS02-039: Buffer overruns in SQL Server 2000 resolution service could enable code execution*, July, 2002.

[12] Red Hat, *Red Hat Linux 8.0: The official Red Hat Linux security guide*, http://www.redhat.com/docs/manuals/linux/RHL-8.0-Manual/security-guide/ .

[13] Ressman, D. and J. Valdes, ''Use of CFengine for automated, multi-platform software and patch distribution,'' *Proceedings of the 14th LISA Conference*, pp. 207-218, December, 2000.

[14] P. Riddle. ''Automated Upgrades in a Lab Environment,'' *Proceedings of the 8th LISA Conference*, pp. 33-36, September, 1994.

[15] Rose, M. and K. McCloghrie, *RFC 1212: Concise MIB definitions*, March, 1991.

[16] Sapuntzakis, C., R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum, ''Optimizing the Migration of Virtual Computers,'' *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pp. 377-390, December, 2002.

[17] Sapuntzakis C. and M. S. Lam. ''Virtual appliances in the Collective: A Road to Hassle-free Computing,'' *Proceedings of the 9th Hot Topics in Operating System*, May, 2003.

[18] Shaddock, M., M. Mitchell, and H. Harrison, ''How to Upgrade 1500 Workstations on Saturday, and Still Have Time to Mow the Yard on Sunday,'' *Proceedings of the 9th LISA Conference*, pp. 59-66, September, 1995.

[19] *Slammer worm hits the Net*, http://news.com.com/1200-1001-982780.html , January, 2003.

[20] *Stanford's e-mail service restored following shutdown*, http://www.stanford.edu/dept/news/pr/03/virus611.html ,June, 2003.

[21] *Sun N1*, http://www.sun.com/n1 .

[22] *floppyfw*, http://www.zelow.no/floppyfw/ .

[23] *VMware Control Center*, http://www.vmware.com/products/cc_features.html .

[24] *VMware Scripting API*, http://www.vmware.com/support/developer/scripting-API/doc/ .

[25] Waldspurger, C. A., ''Memory resource management in VMware ESX server,'' *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, December, 2002.

Appendix A: BNF Grammar of CVL

```
Program       ::=  ProgramStmt*
ProgramStmt   ::=  ImportStmt | AsgnStmt |
                       ObjDecl

ImportStmt    ::=  import Map Item ;
ObjDecl       ::=  Item extends Item { Stmt* }

Stmt          ::=  VarDecl | CompDecl |
                       AsgnStmt
VarDecl       ::=  var AttrList ItemList ;
AttrList      ::=  QuotedStr*
CompDecl      ::=  Item ItemList ;

AsgnStmt      ::=  Ident = RhsList ;
RhsList       ::=  Rhs | Rhs , RhsList
Rhs           ::=  QuotedStr | Ident | Map
Map           ::=  { PairList }

PairList      ::=  Pair | Pair , PairList
Pair          ::=  Item => Rhs

URL           ::=  [file | http]://non-whitespace
QuotedStr     ::=  "any characters (quotes escaped)"
Ident         ::=  Item | Item . Ident
ItemList      ::=  Item | Item , ItemList
Item          ::=  alpha alphanumsym*
alphanumsym   ::=  alpha | 0-9 | _
alpha         ::=  a-z | A-Z
```