Private Cloud Configuration with MetaConfig

As published in the proceedings for IEEE 4th International Conference on Cloud Computing (CLOUD 2011)
http://thecloudcomputing.org/2011
Private Cloud Configuration with MetaConfig

Thomas Damgaard Nielsen
and Christian Iversen
Cirego A/S
Copenhagen, Denmark
Email: support@cirego.com

Philippe Bonnet
IT University of Copenhagen
Copenhagen, Denmark
Email: phbo@itu.dk

Abstract—With the advent of private clouds, the challenge
of configuring a mix of physical and virtual machines is no
longer reserved to a few system administrator gurus. How to
assign virtual machines onto physical machines to leverage the
available resources? How to maintain the virtual machine con-
fugurations congruent with a given specification, even in the face
of unexpected changes? System administrators should be able
to rely on appropriate tools to tackle these questions. However,
today, even if tools exist to automate system configuration,
there is no solution that seamlessly integrate the administration
and configuration of virtual as well as physical machines. In
this paper, we present MetaConfig, a system that supports (a)
convergent as well as congruent configuration management, (b)
full bootstrapping of blank physical and virtual machines and (c)
automatic allocation of virtual machines onto physical machines
based on their resource requirements. We describe our design
and explain how MetaConfig differs from existing systems such
as Cfengine and Puppet. Based on a robust implementation,
we evaluate how MetaConfig handles virtual machine allocation,
unexpected changes, and scalability requirements.

I. INTRODUCTION

Private clouds are emerging as a way for mid-sized or-
ganizations to dynamically control the scope of the IT ser-
vices they provide, while guaranteeing the privacy of user
data. For example, a private cloud is an efficient way for a
public university to improve the level of service provided to
staff and students (e.g., improved up-time for email services,
improved storage space for backups), while conforming to
the privacy requirements of a public organization. In this
context, Infrastructure as a Service offers most flexibility, at
the cost of a much increased burden on the organization’s
system administrators. Indeed, configuring a dynamic set of
virtual and physical machines is a challenge. In this paper,
we introduce MetaConfig, a system that integrates virtual
machine placement, bootstrapping and congruent configuration
management, thus providing system administrators with the
tools they need to manage a private cloud.

The key to efficient private cloud infrastructure management
is a seamless configuration of virtual and physical machines.
As a system administrator it is cumbersome to rely on a set
of tools for automatic configuration of physical machines,
while virtual machine configuration is still largely a manual
task. Seamless configuration of virtual and physical machines
entails the following requirements:

1) Configuration management Given a set of machines –
virtual or physical – and their respective configuration
specifications, we wish to observe the machines’ state
and make any adjustments necessary to make reality fit
our specification. This requires that the specifications are
given in an unambiguous, executable form.

2) Virtual Machine allocation Given a set of physical
machines and a set of virtual machines, we wish to find
a mapping of each virtual machine to a physical machine
that provides the required resources and optimizes a
given function (e.g., minimizes the number of used
physical machines).

3) Bootstrapping Configuring a machine from scratch is
a different and more complex task than configuring a
running machine. When a running machine is config-
ured, all dependencies are already met: file systems exist,
the operating system is installed, drivers are installed,
esential libraries and applications are installed. When
a machine is to be configured from scratch – because
new capabilities are required or because an unexpected
change requires a new machine – all these dependencies
must be resolved in the correct order just to get a
basic working system up, then the task of configuring a
running host must be performed.

To meet these challenges, we decided to design a new
configuration management system that integrates configuration
management, virtual machine allocation and bootstrapping.
Existing systems such as Cfengine1 or Puppet2 only focus
on configuration, and as a result additional tools are required
to automate the administration of virtual machines. In fact,
the solutions we adopted for configuration management (DSL,
congruent engine) are comparable to Puppet (see Section VII
for a detailed discussion) — and while we believe that an in-
grated solution is most usable, the components of our system
that deals with virtual machine allocation and bootstrapping can
be combined with an existing Puppet system. Our contribution
is the following:

• We present the design of our system, MetaConfig, with
an emphasis on the seamless configuration of virtual and
physical machines. We explain how MetaConfig (a) finds
candidate physical hosts that satisfy the requirements for
running a specified virtual machine or group of virtual
machines, (b) allocates a virtual machine or group of

1http://www.cfengine.org/
2http://www.puppetlabs.com/
virtual machines on a set of candidate physical hosts, and (c) starts, stops, and migrates virtual machines according to the specification in the model.

Based on our implementation of MetaConfig, we evaluate how it handles virtual machine allocation, unexpected changes, and scalability requirements.

II. SYSTEM DESIGN

The overall design of our system is shown in Figure 1. The system is composed of a collection of physical and virtual machines (7 virtual machines deployed on 3 physical machines on the Figure), a control server and a database. Conceptually, MetaConfig is composed of three subsystems:

1) The configuration subsystem: Each physical and virtual machine is running a configuration client that communicates with the configuration server (running on the control server). The configuration subsystem is responsible for maintaining the configuration of a given machine based on a configuration specification stored in the database.

2) The resource subsystem: Each physical machine runs a resource client that manages the virtual machines on it. These clients communicate with the resource server (running on the control server).

3) The installation subsystem: Finally, for bootstrapping, the physical machines run an install client. Virtual machines are partially bootstrapped in the context of their controlling physical machine, partially in their own context when they can be booted. Thus, the resource client communicates with the install server to provide initial bootstrapping of virtual machines. After this, the virtual machine is rebooted and the configuration client takes over.

III. CONFIGURATION MANAGEMENT

We wish our configuration specification to be compact, modular, non-redundant and complete, and the execution to be reproducible and idempotent. To meet these goals, we defined a declarative domain-specific language (DSL): An administrator expresses what a configuration should be, not how it should be obtained [1].

Each configuration file is compiled and the run-time system dynamically computes the difference between the current configuration and the target configuration as expressed with the DSL, and make the necessary adjustments. These adjustments are encoded as micro-operations in the run-time system. There is thus no need for hand-written shell script in the system.

Modularity is achieved through an object-oriented style of declaring files, settings, packages, and other resources in components. Any conditional evaluation can be overridden on a per-host basis. There is thus no need for hard-coding conditional expressions to match specific hosts.

Because of lack of space, we cannot present the details of our domain-specific language in this paper. Interested readers are referred to [2] for a complete description of the language (including a BNF) and the run-time system (including a detailed description of all micro-operations and the database).

In order to illustrate our design, we show here the specification of a mail server configuration:

```bash
## Example configuration for mail server

import "server/default"
# Import default server settings
import "service/postfix"  # Import postfix service
import "service/dovecot"  # Import dovecot service

[apt]
install += "procmail"
# Create group and user account for mail multiplexer

[group.mailplex]
gid = 1000
uid = 1000
gid = 1000
gecos = "Mail multiplexer account"
home = "/home/procmail"
shell = "/bin/zsh"

[files.def.config-hostname]
hostname = "mail"

[files.def.config-fstab]
fs1_src = "/dev/sda2"
# Set of postfix service

[files.def.service-postfix]
mailerdaemon_alias = "staff@example.org"
content_filter = "mailscan.example.org"
smtplib = "example.org"
web_lists = "sbl-xbl.spamhaus.org"
mynetworks = "127.0.0.0/8 192.168.42.0/26"
myhostname = "mail.example.org"
mydestination = ["localhost", "mail.example.org"]
ssl_support = 1
ssl_cert = "/data/certs/local/mail.example.org.crt"
ssl_key = "/data/certs/local/mail.example.org.key"
recipient_access = 1
inbound_accept = "staff@example.org"

# Set up PostgreSQL extension to postfix

[files.def.config-postfix-pgsql]
host = "pgsql.example.org"
user = "hostingdb_ro"
database = "hostingdb"
```

Fig. 1. A view of the overall design of MetaConfig.
To compile the global knowledge map, we need a network infrastructure that enables communication between all the physical servers so that we will have all the necessary information available somewhere in the group.

Since our design requires a central authority for keeping the global knowledge map, we choose the classic client-server model for network communication. Only the server, which has the knowledge map, has the information necessary to implement changes. Thus, we can keep the clients simple: they merely serialize state information, and implement changes sent from the server.

The resource clients run on each physical machine and send information about the virtual machines running on it to the resource server at regular intervals. A conceptual overview is seen in figure 1. The clients continually update the global knowledge map. Then the server queries the database for the global knowledge map and the specification for each machine. Then it compares these, creates a list of changes for the client, and sends the list of changes back as a reply to the client update.

To avoid being tied to a specific virtualization platform, we use the virtualization control library libvirt [3]. Libvirt provides an API for managing platform virtualization through technologies such as KVM, Xen, VMware, etc. This allows us to control many different virtualization technologies through a uniform interface.

B. Allocation Algorithm

Our allocation algorithm exclusively focuses on placing new machines. This ensures that it will always produce correct output, and will be simple to implement. We then continuously run an optimization daemon to improve the virtual machine distribution among our physical hosts. This has the added advantage that we can freely implement our optimization policy in this daemon, while keeping the allocation algorithm constant. Another advantage of this design is the ability to smooth out hot-spots in the data center. If some physical machines are being overloaded, the optimization daemon can dynamically move virtual machines away from them to improve performance.

Given a set of physical machines $P$ and a set of virtual machines $V$, our goal is to define an allocation planner that maps virtual machines in $V$ onto $P$ with a minimal number of physical machines used and a minimal number of virtual machine unallocated. This virtual machine allocation problem is NP-hard. A similar well-known problem is the bin packing problem [4]. The bin packing problem is NP-hard. By treating the physical servers as bins with available resources (RAM, CPUs, and architecture) as the dimensions and the virtual machines as objects with size equal to their resource requirements, we can reduce the multi-dimensional bin packing problem to the virtual machine allocation problem.

3Obviously, other optimization functions could be used here, e.g., minimize the number of virtual machines per physical machine in order to try to eliminate data center hot-spots.
Physical machines can be seen as 3 dimensional bins and virtual machines as 3 dimensional objects that can be placed inside bins. A physical machine provides three types of resources: RAM, CPUs, and an architecture. Usually, a physical machine can provide only one architecture. This makes the problem somewhat easier, since the one dimension (architecture) will always be of size 1, however, for a virtual machine to be allocated on a physical machine, the physical machine must provide a compatible architecture.

Since the problem of efficient allocation of virtual machines is NP-hard, we will not try to find an optimal solution. We will instead base our allocation algorithm on the well-known First Fit Decreasing heuristic.

Under the assumption that there are always enough available bins (physical machines), the algorithm will find a valid solution if such one exists. However, it is not guaranteed that the solution will be optimal.

The First Fit Decreasing sorts the objects to be allocated in decreasing order by volume. Then it inserts each object in to the first bin with sufficient remaining space. The time complexity of this algorithm is $O(n \log(n))$, where $n$ is the number of objects to be allocated, since it is the most expensive step is sorting which takes $O(n \log(n))$. Insertion takes $O(n)$.

Since each physical machine provides only one architecture, we can treat the set of physical machines as several sets of 2 dimensional bins: One set of 2 dimensional bins for each available architecture. We can do the same for virtual machines. Since each virtual machine requires only one architecture, we can treat the virtual machines as several sets of 2 dimensional objects. One set of 2 dimensional objects for each different architecture.

We have now simplified our problem from one 3 dimensional bin packing to several 2 dimensional bin packing problems. We can go on and design an algorithm based on First Fit Decreasing to solve the problem.

The algorithm we have designed takes a set of virtual machines $V$, and a set of physical machines $P$ as input:

\begin{algorithm}
\textbf{ALLOCATION($V, P$)}
\begin{algorithmic}
1. \textbf{Sort} $V$ decreasing by required RAM. \\
2. \textbf{Sort} $P$ increasing by provided CPUs. \\
\Proc{for $v$ in $V$} \\
\Indent \Proc{for $p$ in $P$} \\
\Indent \Proc{if $p$ satisfy the requirements of $v$} \\
\Indent \Proc{Allocate $v$ on $p$} \\
\Indent \Proc{break} \\
\EndProc \EndProc \EndProc \\
\end{algorithmic}
\end{algorithm}

If there are not enough available physical machines or if some requirements cannot be satisfied the algorithm will leave some virtual machines unallocated.

The complexity of the algorithm is dominated by the two for loops which runs in quadratic time as well as the sorting of $V$ and $P$.

V. BOOTSTRAPPING

Configuring a host from scratch is a different and more complex task than configuring a running host. When a running host is configured, all dependencies of the running host are already met: file systems exist, the operating system is installed, drivers are installed, essential libraries and applications are installed, etc. When a host is to be configured from scratch, all these dependencies must be resolved in the correct order just to get a basic working system up, then the task of configuring a running host must be performed.

A. The problem

When a new host is added, it must be given an operating system and some initial configuration so that it can be brought into the final specified configuration. We call this process bootstrapping a host.

For physical machines, the problem is: Given a computer with at least one Ethernet network interface and a hard drive, bring that computer to a specified configuration. This problem consists of several sub problems:

- Boot the computer
- Partition the hard drive and create a file system
- Install operating system
- Install MetaConfig client
- Obtain configuration specification
- Apply the configuration

A virtual machine does not have an actual hard drive, instead it has an image file. When the virtual machine is not running, we can just create a file system on the image file offline. So the tasks needed to bootstrap virtual machines are a subset of the tasks needed to bootstrap a physical machine. In the following, we will therefore base the discussion on bootstrapping physical machines.

There are different ways of bootstrapping hosts. We are focusing on the following goals:

1) Minimal interactivity It must be easy to bootstrap hosts, so it should require a minimal amount of user interaction.

2) Efficient storage utilization When a host is bootstrapped, an operating system and other files are installed. This means that a lot of data must be stored somewhere to be available for copying onto the hosts. Thus, it is important to consider what data has to be stored.

3) Flexible It should be possible to bootstrap any kind of host without changing the bootstrap system.

When manually bootstrapping a host, the system administrator will typically boot the host up on an installation medium — e.g. an installation CDROM. The installation medium will launch an installer — a program that will install the operating system. The installer will partition the hard drive and create an initial root file system. It will then copy initial files onto the file system and install a boot-loader. It will typically ask the administrator questions about which components to install and about the basic configuration — e.g. host name,
network configuration, initial users and so on. The installer will configure the operating system based on the choices made by the administrator during installation. When the installation is complete, the administrator will remove the installation medium and reboot the machine. The machine will then boot up the newly installed operating system.

B. Metacfg Solution

We have created a network boot program from a modified standard initrd from the Debian GNU/Linux project. We have put various tools such as fdisk, mkfs, chroot, dhclient, and rsync plus their dependencies into the initrd. This program is our install client.

When booting, the install client will detect the hard drive and then partition it and format a root file system on it. It will then bring up the network, mount the root file system, and copy initial files onto it from a central install server.

The install server has the responsibility of handling the server side of the PXE protocol — i.e. serving DHCP request from hosts that are to be installed and serving the custom initrd via TFTP. The install server must also run different services used during bootstrap.

The install server will run the following services:

• **DHCP server** A DHCP server is needed to perform the initial PXE boot. It must send the next-server parameter in the first DHCP response.

• **TFTP server** A TFTP server is needed to perform the initial PXE boot. It must serve the initrd network boot program.

• **rsync server** The initrd copies files from the install server onto the root file system on the target host. The files are copied via rsync.

• **HTTP server** HTTP is used as a simple communication between the install server and the target host. During installation, the target host sends status reports to the install server via HTTP. The HTTP server must perform certain actions upon each status notification. These actions will be explained below.

In our implementation, the install server runs all the above services. However, for scalability it should be possible to run the services on different hosts as needed.

Apart from the services mentioned above, the install server must have access to the control database. The database contains information on each host. It is the install server’s responsibility to continuously update the install status in the database as the install progresses.

We will not go into details on every bootstrapping step. Interested readers are referred to [2] for details.

One of the most significant advantages of our bootstrap solution is the integration between the configuration subsystem and resource subsystem. In the bootstrap process of virtual machines, certain actions must be taken — reboot, file system creation, skeleton file copy, etc. — by various parts of the system to achieve complete bootstrapping. We have found, that without an integration effort between these two systems, the bootstrap process is difficult to automate.

VI. EVALUATION

A. Virtual Machine Allocation

To illustrate the virtual machine allocation algorithm, we report on the following tests:

1) Allocating several virtual machines with different hardware requirements.
2) Allocating several virtual machines with different hardware requirements, some of which cannot be satisfied.
3) Reallocating running machines to a more compact solution.

We consider a test environment that consists of the following physical machines:

- **phys1** with 12288 MB RAM and 1 AMD64 compatible CPU.
- **phys2** with 4096 MB RAM and 2 AMD64 compatible CPUs.
- **phys3** with 8192 MB RAM and 2 AMD64 compatible CPUs.
- **phys4** with 8192 MB RAM and 1 PowerPC compatible CPU.
- **phys5** with 4096 MB RAM and 2 i386 compatible CPUs.

These physical machines can host a number of virtual machines. In the following tests, we will use the following virtual machines:

- **mail1-6** 6 machines that each requires 512 MB RAM and 1 AMD64 CPU.
- **web1-8** 8 machines that each requires 256 MB RAM and 2 AMD64 CPUs.
- **db1-4** 4 machines that each requires 2048 MB RAM and 2 AMD64 CPUs.

When we ask the allocation planner to allocate all these virtual machines on the physical machines, we obtain the following mapping — note that there is no unallocated node and that all the virtual machines are allocated on three physical machines:

```plaintext
phys5: CPUS: 2 x i386
       RAM: [____________________]
4096 MB/4096 MB free
Domains:
phys4: CPUS: 1 x powerpc
       RAM: [_________]
8192 MB/8192 MB free
Domains:
phys1: CPUS: 1 x amd64
       RAM: [#####__________]
9216 MB/12288 MB free
Domains:
<Host mail1:512MB,1,amd64>
<Host mail2:512MB,1,amd64>
<Host mail3:512MB,1,amd64>
<Host mail4:512MB,1,amd64>
<Host mail5:512MB,1,amd64>
<Host mail6:512MB,1,amd64>
phys3: CPUS: 2 x amd64
       RAM: [################################################]
0 MB/8192 MB free
```
Let us now change the specification of mail1 so it requires 4 CPUs and the specification of each database server so that it RAM requirements increases from 2048 to 3072. We expect the allocation planner to return a mapping with mail1 unallocated due to the fact that there are no physical servers that provide 4 CPUs. We also expect that one of the database servers cannot be allocated due to the fact that they require 2 AMD64 CPUs and the servers that provides 2 CMD64 CPUs, phys2, and phys3, do not have enough RAM to hold all servers that require 2 CPUs. The allocation planner returns the following mapping:

As expected, all virtual servers except mail11 and db4 were allocated.

**B. Unexpected Changes**

In order to test the configuration system, and to make sure it can correct simple user-errors that fall within the configuration we have specified, we test each micro operation separately. We have created a test case for each micro operation, and tested if the system can detect and repair an error involving that operation.

To inject an error, we create a change in the system that is the opposite of the operation we need to test. For example, when testing the “remove file” operation, we create the file first. When testing the “install package” operation, we remove the package first, etc. The system starts out in a fully configured state where the configuration system reports no necessary changes.

We cannot describe all the test cases because of lack of space, interested readers are referred to [2]. Here is for illustration, the and the MetaConfig output for the “install package” operation: the first command removes the pwgen package, the second command displays the discrepancy detected by MetaConfig between the observed state and the specification, and the third command actually executes the necessary configuration tasks (here installing the pwgen package).

```
[root@eos ~] #dpkg -P pwgen
(Reading database ... 358272 files and directories currently installed.)
Removing pwgen ...
Processing triggers for man-db ...
[*] Would have apt-get [pwgen]
[*] Plan finished
[root@eos ~] #spye metaconfig update-machine
[*] Updating apt database..
[*] Installing [pwgen]
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
pwgen
0 upgraded, 1 newly installed, 0 to remove and 36 not upgraded.
Need to get 0B/21,7kB of archives.
After this operation, 86,0kB of additional disk space will be used.
Selecting previously deselected package pwgen.
(Reading database ... 358268 files and directories currently installed.)
Unpacking pwgen
(from .../pwgen_2.06-1ubuntu2_amd64.deb) ...
Processing triggers for man-db ...
Setting up pwgen (2.06-1ubuntu2) ...
localpurge:
space freed in /usr/share/locale: 0 KiB
space freed in /usr/share/man: 0 KiB
Total disk space freed by localepurge: 0 KiB
[*] Metaconfiguration done
```

Fig. 2. Install apt package.
C. Scalability

In order to test how the system scales to a large number of hosts, we need access to a large number of machines. For this experiment, we negotiated access to 100 virtual hosts on the Amazon EC2 service. We have chosen to measure the time taken to simultaneously configure up to 100 hosts. The configuration specification that has been used for this test corresponds to the simple mail server from Section A.

The hosts to be configured are located in the Amazon EC2 east region, located in Virginia, USA. The Amazon EC2 hosts are each connected to the Internet via 100 Mbit/s Ethernet link. During configuration, the hosts must obtain the configuration specification from Subversion repositories and communicate with the control server. These servers are located Copenhagen, Denmark. The Subversion server is connected to the Internet via 100 Mbit/s Ethernet. The hosts must also obtain software packages from the Debian software mirrors located on various locations in USA.

We configure up to 100 machines in parallel. We expect to see no dramatic increase in the time taken to configure the hosts as the number of hosts increase. Figure 3 shows maximum, average, medium and minimum duration of the configuration as a function of the number of hosts configured.

As expected, MetaConfig scales up with an increasing number of physical machines configured in parallel. Interestingly, initial runs of this experiment exposed scalability issues on the Subversion server and on the Debian mirror used for the configuration — we fixed those initial problems by implementing caching. After caching the remote content, we obtained the results shown in Figure 3.

Cfengine is quite popular. In 2001 the system was estimated to be used on 100,000 nodes worldwide [6]. According to statistics from the Cfengine company website Cfengine is estimated to be used on several million computers worldwide.

Cfengine uses a declarative DSL for configuration specification. Although Cfengine can be used for congruent configuration management by first deploying a baseline configuration to all hosts and then applying purely orthogonal and non-interacting changes, Cfengine is designed for convergent configuration management.

Cfengine uses so-called agents to achieve the desired configuration on the hosts. Each host has an agent installed. This agent is responsible for performing the operations needed in order to bring the host into the desired configuration.

Cfengine cannot guarantee that a system is in the desired configuration but supports statistical compliance with policy. This means that it converges towards the desired configuration by a best effort [7].

B. ISConf

ISConf is a configuration management system that is written in Python [8], and released under the free software license GPL. The project was started in 1998, but does not seem to be actively developed as the latest stable release is from June 2006. Unlike Cfengine, ISConf uses the congruence approach to configuration management.

Instead of a custom DSL, ISConf uses a combination of makefiles and shell scripts to represent configuration and dependencies between configuration modules.

ISConf uses strict ordering of commands to ensure congruency with baseline configuration. Each host is given the same initial configuration. ISConf will then enforce that each host will execute a specified set of commands in the same order, thus obtaining the exact same configuration. If a host is down during a configuration deployment, this particular host will execute the same commands as the other hosts did when it comes back up.

ISConf halts on errors. This means that if any errors occur in deploying the configuration, the system exits. This halt on error policy guarantees that all hosts have executed the same steps up to the point when the error occurred. No host will have skipped a step or executed a step out of order.

C. PIKT

Another approach to automated system configuration is PIKT [9]. The project was started in 1998 and is released under the free software license GPL. The project does not seem to be actively developed, as the latest stable release is from September 2007.

PIKT is designed primarily to monitor systems and report problems. It can then try to fix detected problems.

Like Cfengine, PIKT is implemented in the C programming language. On the clients PIKT consists of two daemons: piktc_svc and piktd. The daemon piktc_svc is a network server that listens for command requests, while piktd
periodically launches Pikt scripts at specified intervals — much like cron.

Pikt scripting language is a low-level, procedural language. Pikt scripts often make use of auxiliary scripts written in Perl, expect and shell scripts.

Although PIKT’s is designed primarily for monitoring, it also provides features for configuration files installation and management. Configuration files can be managed centrally and distributed to the clients from a central server. PIKT can be configured to monitor the current state of configuration files continuously and report any changes or update configuration files according to a central specification. However, the learning curve for PIKT is steep and setup is difficult [9].

D. Puppet

Puppet is a configuration management system that is written in Ruby. Puppet is the newest of the surveyed systems. The first release of Puppet was in 2005, so it is relatively new compared to Cfengine, Iconf and PIKT.

Puppet uses a declarative DSL to describe the desired configuration. The configuration is then deployed on the hosts according to the specification. Puppet uses a DSL to represent configuration. The DSL describes resources. A resource can be an entity like a file, a service, or a package. Our DSL is comparable to Puppet’s. In terms of system design, Puppet covers most of the functionality of our Configuration subsystem. Puppet does not provide equivalent to our Resource and Install subsystems. Integrating Puppet as the configuration subsystem of a Metaconfig installation is future work.

E. SaaS Configuration

The configuration of software services within a private cloud is largely complementary to our approach. Hagen et al [10] describe an object-oriented model of services such as database and web-based applications as well as an infrastructure for maintaining these services. The SmartRod framework [11] focuses on the resources used by Java-based services and adapts a service configuration based on the resources available.

VIII. CONCLUSION

MetaConfig is a system goal for managing virtual machines and their resources, including support for automatic deployment of configuration and installation of both virtual and physical machines. Future work includes:

- **Windows support** Supporting a non-Unix platform such as Windows requires major effort to implement. Not only are file semantics vastly different from Unix, users and groups are also handled in different ways. Even more different is the package system used. Windows uses MSI packages which are similar in scope to the Debian packages, but without a package distribution infrastructure like Apt. To support package installation on windows, we would have to implement a package distribution system.

- **Windows registry** Being able to manage the Windows registry database would be a useful feature, even under Unix-emulation environments such as Cygwin. Many programs use the registry instead of configuration files, so adapting our templating language to work with the registry is likely a useful implementation strategy.

- **More handlers** We lack handlers for a number of configuration aspects. These include Samba user accounts, database user accounts, Aplt sources, cryptographic keys for Aplt, sysvinit enabled services, etc.

- **Directory content handling** Some programs store configuration in a set of files in a directory, and work with the sum of all content of the files. This means that a stray file will affect the program even if all the files we intend to manage stay the same. It would therefore be useful to support the deletion of unmanaged files in specified directories, to ensure that only the managed files can affect the programs. If the directory handling policy is flexible enough, this could be used for other tasks as well, such as cleaning up old files from /tmp.

REFERENCES


