Contextualization: Providing One-Click Virtual Clusters

Katarzyna Keahey
University of Chicago
keahey@mcs.anl.gov

Tim Freeman
University of Chicago
tfreeman@mcs.anl.gov

Abstract

As virtual appliances become more prevalent, we encounter the need to stop manually adapting them to their deployment context each time they are deployed. We examine appliance contextualization needs and present architecture for secure, consistent, and dynamic contextualization, in particular for groups of appliances that must work together in a shared security context. This architecture allows for programmatic cluster creation and use, as well as mitigating potential errors and unnecessary charges during setup time. For portability across many deployment mechanisms, we introduce the concept of a standalone context broker. We describe the current implementation of the entire architecture using the Virtual Workspaces toolkit, showing real-life examples of dynamically contextualized Grid clusters.

1. Introduction

Providers of compute cycles in the cloud, such as Amazon EC2 [1] or the Science Clouds [2], enable users to acquire on-demand compute resources, usually in the form of virtual machines (VMs). To be useful, the acquired group of VMs typically still has to be configured into a working cluster: common applications such as shared file system or a batch scheduler have to be configured to reflect the group of machines belonging to the cluster domain. In other words, the cluster needs to establish its context: share the networking information assigned to individual VMs when they are deployed (such as IP addresses and hostnames) an the security information that is often specific to a particular deployment. In short, for the virtual cluster to be useful, a configuration phase needs to be completed at deployment time.

This configuration phase can happen in two ways: (1) we can deploy an image configured with a basic environment, then install and configure the context-sensitive applications at deployment time [3, 4], or (2) deploy fully configured images and adjust the configuration of context-sensitive applications after deployment. The first option often results in a long deployment time for nontrivial systems: the process of configuring a real-life scientific cluster may take many hours [5]. The second option, while potentially faster, typically requires knowledge of the applications that have been installed on the VM and is thus carried out manually. Both options have the potential to make the process of VM deployment long. We argue that by coordinating the process of appliance preparation and appliance deployment we can provide a generic, lightweight, and automated mechanism that will quickly deploy fully configured images and adapt them to their deployment context.

In [6] we introduced the term *contextualization* to describe such a process for single VMs. In this paper, we provide a more comprehensive discussion of what it takes to build and contextualize virtual clusters and other complex constructs. We present a detailed description of the architecture, discuss the security aspects of context creation, and describe how context information can be brokered between multiple VMs. We analyze two appliance deployment implementations: one provided by the Amazon EC2 service [7], the other by the Nimbus Toolkit [8], and describe how the Nimbus Context Broker service can work with each to deploy one-click virtual clusters of varying complexity. We also illustrate with examples how contextualization is used in practice.

This paper is organized as follows. In Section 2 we discuss the roles and responsibilities of appliance providers and appliance deployers. Section 3 describes a context brokering architecture, compares the features of two existing deployer implementations, and describes how they can work with the Nimbus Context Broker. Section 4 gives examples. Sections 5 and 6 present related and future work and Section 7 summarizes the findings.

2. Roles and Responsibilities

Virtual organizations (VOs) [9] bring together scientists collaborating to solve problems that require a common and consistent set of environments customized to satisfy the needs of a VO-specific applications. These needs can range from providing a software stack capable of supporting VO applications to defining levels of isolation and security associated with the work in those environments. In order to serve the needs of its community, a VO must find ways of expressing the required environments and mapping those environments onto resources.

This goal is hard to achieve in today's grids because the environment configuration is almost entirely in the hands of resource providers who configure and maintain environments trying to find a compromise between the needs of as many VOs as possible. This strategy often fails because reconciling the needs of different VOs is time-consuming at best and impossible at worst: different VOs require environment updates in different timeframes and sharing relationships are often ad hoc, complex and ill defined.

The past few years have seen the emergence of virtual appliances [10] that define an environment as an abstraction independent of its deployment. In doing so, appliances decouple the notion of environment configuration and maintenance from the notion of their assignment to resources. Such appliances no longer need to be maintained by the resource providers; they can be maintained by the communities that use them, and then mapped onto resources. This rethinking of the division of labor between the providers and consumers of resources adds a new role to the process, that of an appliance provider, and new flexibility (provided by new software) that allows us to explicitly provision environments on available resources.

Figure 1 shows the interdependencies between the different roles:

- *Appliance providers* configure environments, take responsibility for maintaining them (e.g. applying security updates), and guarantee their consistency and freshness. In doing so, the appliance providers may be assisted by appliance management tools [11][] and provide mechanisms for convenient updates, versioning, verification, and so forth.
- *Resource providers* provide resources with limited configuration requirements designed to support appliances but no longer to provide end-user environments for multiple communities. The same appliance may be moved seamlessly between open, proprietary, and leased platforms to cope with peak demand.
- *Appliance deployers* coordinate the mapping of appliances onto available resource platforms and information exchange between groups of appliances to enable them to share information and sharing relationships that constitute the *context* of their deployment.

In a Grid environment, a logical choice for the appliance provider is a VO (or representatives designated by the VO), since VOs are typically associated with a set of well-defined compute environment required by a community. To be effective, the VOs will need to develop infrastructure or have available tools to provide and maintain appliances. Introducing the concept of a virtual appliance does not necessarily save work; it simply puts work in the hands of the party that is most qualified and motivated to do it.

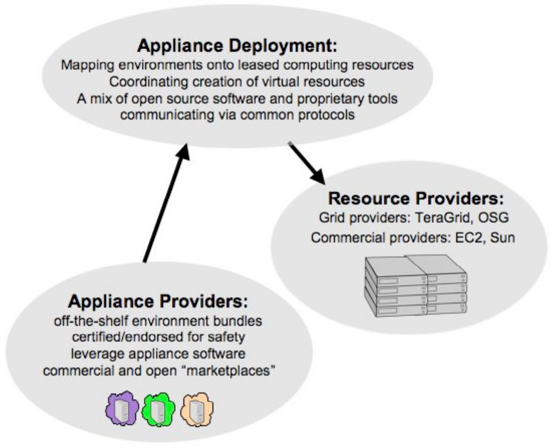
Appliance Deployment:
Mapping environments onto leased computing resources
Coordinating creation of virtual resources
A mix of open source software and proprietary tools
communicating via common protocols
Resource Providers:
Grid providers: TeraGrid, OSG
Commercial providers: EC2, Sun
Appliance Providers:
off-the-shelf environment bundles
certified/endorsed for safety
leverage appliance software
commercial and open "marketplaces"

Figure 1: Roles and responsibilities: appliance deployers map appliances prepared by appliance providers onto resources.

This paper describes the challenges and solutions facing the appliance deployer. These can be seen as the interplay of two layers: (1) mapping appliances onto resources and (2) configuring them to represent functional units aware of the surrounding context. We addressed (1) in our research in [12, 13]. We now focus on (2): the contextualization process.

3. The Contextualization Process

Each appliance is deployed in a specific deployment *context* that may be defined by a Grid, a site, other appliances (e.g., when the appliance is part of a "virtual cluster" [14]), or all of the above. Since appliances are deployed dynamically, each appliance deployment instance is potentially associated with a different context. For example, the appliance's IP address and hostname may be new or reassigned and deployment-specific security data generated. Therefore, each time the appliance is deployed, it must be able to integrate information about its current deployment context in order to function within that context. We call this process of adapting an appliance to its deployment context *contextualization*.

In [6], we defined an appliance as an environment capable of being contextualized (a prerequisite for dynamic deployment), that is, an environment that defines the required context information (*contextualization template*) and can integrate this information into the appliance so that the appliance works in its deployment context. We call the agent acting to integrate the contextualization information into an appliance the *contextualization agent.* We also described a simple contextualization mechanism that allowed us to configure appliances as long as all the context information was available. This was achieved as a collaboration between the appliance provider and appliance deployer and worked as follows. Each time an appliance provider put a new application into the appliance, they would define what context-dependent information was required to make the application work (e.g., a Grid service might require a host certificate). The appliance provider would then specify the required information in the appliance template and write a script (contextualization agent) capable of integrating the context information into the appliance (e.g., by modifying configuration files for an application). The appliance deployer would provide the information described in the contextualization template at boot time, and on boot the contextualization scripts/agents would integrate the information into the appliance.

This method assumed that all of the context information was available on boot, that the context did not change during the appliance's deployment, and that the deployer of an appliance was the same entity that coordinated the context exchange between appliances and the larger context. However, these assumptions are not necessarily true: if we simultaneously deploy several appliances depending on each other for context information, not all of the context information will be available on boot (i.e., each appliance needs to *provide* as well as *consume* context information). Also, in practice the appliance deployer (e.g., the Amazon's EC2 service) may not be privy to, say, VO-specific authorization policy information required by the appliance. We therefore extended our model to account for those situations. We still assume that context integration will occur at boot time (i.e., we provide no mechanisms for recontextualization).

3.1. Architecture

The process of contextualization depends on the collaboration of three parties, each potentially in a distinct trust domain, and each potentially providing information in a different idiom. Figure 2 shows the interactions between the appliance and the three components. The *appliance provider* configures the appliance, providing the disk image and corresponding contextualization template that describes what information is required or contributed by the appliance toward the establishment of the context (1). The *appliance deployer* start sup the appliance and provides some generic appliance information (2). The *context broker* coordinates the exchange of application-specific contextualization information (3).

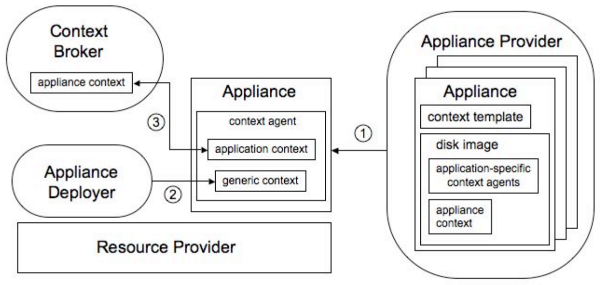

Figure 2: Relationship between appliance provider, appliance deployer, and context broker.

A context broker manages objects describing information associated with a specific context. A context object captures context information relevant to a specific virtual construct (a virtual cluster, collaboration, or Grid), serves as an exchange board for that information, and defines the security and trust for that context. Context information may be provided by the client (e.g., an access control list defining individuals who can access a virtual cluster), by the appliance deployer, by any appliance that exist within the context and contribute to it, or from other sources. Since we currently assume that all the context information will become available within a short time after the appliance boots, we require that a context object eventually reach a stable state (when all the expected information has been provided). At that point the information is distributed to the appliances.

The appliance providers, deployers, and context distributors interact as follows (see Figure 2):

1) The appliance gets configured by the appliance provider. As part of the configuration, each application participating in the appliance provides a script that integrates context information into the appliance at boot time and a description of contextualization requirements to be put in the contextualization template. In addition to this application-specific context information, each appliance requires generic context information (see below). Both the script and the template are provided as part of appliance packaging process.

2) When the appliance is deployed by the appliance deployer, it is associated with a specific context broker. The deployer delivers to the appliance (either via push or pull, see Section 3.2) the generic context information including a way to contact the context broker. In order to obtain more context information, the appliance will contact the context broker.

3) After the appliance is booted, the contextualization agents first gather all the context information *provided* by the appliance. They then contact the context broker and deposit the information in the appliance's context. After the context reaches a stable state, they collect the context information the appliance *requires*.

The generic context information delivered by the deployer is as follows:

a) Network id of the appliance (IP address/hostname)

b) Address of the context broker

c) Context identifier

d) A set of credentials that will allow the appliance to prove its identity to the context service and verify the identity of the context service.

Note that b–d are required only if a context broker is used (simple appliances, such as base images, may not need a context broker at all). In addition, if the context broker shares the trust domain with the deployer and the appliance, the security information is not needed.

The model described above illustrates the inter-relationships among the actors in the contextualization process and defines the protocols they need to agree on. The contextualization agent needs to be able to consume and interpret the context information provided by the deployer. The agent also may need to be able to contact the context service and provide required context information itself. While a variety of implementations can be used in all of these cases, standards in this area would greatly facilitate the adoption of the technology.

3.2. Implementation

We now discuss how the architecture described above has been implemented in two systems we are familiar with: the Amazon Elastic Compute Cloud (EC2) [7] and the workspace service [5].

3.2.1. Delivery of Generic Context Information. Both EC2 and the workspace service leverage existing contextualization mechanisms to provide basic context information to the VM. Specifically, they leverage the standard DHCP broadcast call (a part of typical boot sequence) to provide an IP address. In EC2's case two addresses are assigned to the same NIC: a private IP address and a public IP address. The DHCP request returns the private IP; traffic directed to the public IP is eventually redirected to the NIC associated with the private IP [15]. The workspace service delivers all IP address information via a DHCP delivery tool described in [6] or via the site's DHCP service.

The remaining generic context information can be delivered to EC2 instances as follows. For each VM, EC2 creates an "instance metadata" structure on startup for a group of VMs deployed at the same time – a "launch group." Among others, the metadata contains a "user-data" field, allowing the user to provide unstructured data (at most 16KB) to be shared among all the members of the launch group. The information can be provided by an external client via a secure HTTPS connection guaranteeing the privacy of the data. The VM can read this data via an unsecured HTTP GET call; however, since the assumption is that the network between the VM and EC2's data structure is secure, user data can be used to convey, for example, a private key or another secret.

The workspace service likewise exposes the means for a client to provide context information to a group of workspaces via a private HTTPS channel. The workspace service conveys this information to the VM by "image patching" (putting a file with the required information on the VM disk image). The workspace service patches the image with a file containing the generic context information. At boot time, the contextualization agent reads and interprets the information in the file.

Of the three discussed delivery methods, leveraging existing mechanisms (DHCP) would clearly be most convenient – unfortunately it is not feasible to employ it for all applications. We chose image patching in our implementation because it is simple for the contextualization agent (although not necessarily simple for the deployer), because it imposes no practical size constraint, and because it can be used securely without requiring the network on the deployer's side to be trusted. Providing contextualization information via the network (EC2) is also simple but it requires a trusted network to share secrets, which is not always feasible. In addition, contextualization information may be delivered via kernel parameters, but this approach may significantly limit the size of the information that can be delivered.

Since both EC2 and the workspace service provide a secure delivery channel of unstructured information (user data and the image patch, respectively), both can be used to convey the generic context information including the service URL to the context broker, the WSRF key identifying the specific context object, and the security information consisting of the public key identifying the context broker and a private key identifying the context object.

3.2.2. Context Broker Implementation. Neither EC2's user data nor workspace service image patching is suitable for the kind of context brokering described in Section 3.1. First, both methods work one way only (from client via deployer to the VM): neither allows a VM to send information back, so that the VM cannot share its "provides" information. In addition, both are a deployer-specific context mechanism: they rely on the assumption that the VM does not need to create a security context with deployer because it is within the deployer's domain and this domain creates conditions for trusted exchange. In other words, these mechanisms cannot be used to broker information across different deployers or where we cannot assume the existence of a trusted domain.

To overcome these shortcomings, we implemented a context broker to fulfill those tasks. The context broker is implemented as a WSRF service that creates and manages context objects. A context object is created by a client (e.g., by a deployer to assist with the creation of virtual cluster or by the end user who wants several VMs to share a context). On creation, a context object generates a keypair that is used to root a trust/security environment for the context: the private key of this keypair is conveyed to the VM as part of the generic context information along with the public key of the context service. The private key identifying a context can be obtained by the deployer using the HTTPS protocol. A context broker implements two operations: (1) "add workspace," used by a deployer to register the IP of a deployed VM as well as the contextualization template corresponding to the VM, and (2) "add information," which allows a client to add information expressed as an XML document to the context. In addition, a client can also set a flag saying that there are no more workspaces and no more information to be added to the context.

The contextualization template is composed of two sections: *provides* and *requires*. The *provides* section contains a list of labels that describe the role of the VM in the context of a specific application (e.g., if a VM is an NFS server, it will contain the "nfsserver" label in the *provides* section). The *requires* section contains a list of labels that describe what information is required to contextualize the VM (e.g., if a VM is an NFS client, it will contain the "nfsclient" label in the *requires* section). Labels are arbitrary, but they must be such that the contextualization scripts can interpret them.

On deployment, a VM is passed the generic context information described in Section 3.1. When the VM boots, the context agent mutually authenticates with the context broker using the generic context information and provides its identity (VM identity is composed of three typed objects: hostname, IP address, and public host key). The context broker matches this information to the VM's contextualization template and "fills in" the templates by sorting the *provides* information into the *require*s fields of contextualization templates of each VM participating in the context. After all the context identity information has been received and sorted, the context broker releases the templates with filled-in information to the waiting contextualization agents. When the context broker marks a response to a specific context agent as "complete," that context agent invokes the applications-specific contextualization scripts on the VM, which integrate the necessary information into application configuration.

Note that our implementation assumes that the VM authenticates as a "member of context" only (rather than an individual entity) and that the members of context are trusted between themselves (i.e., they are trusted to identify themselves within the context).

4. Contextualization Examples

We implemented the mechanisms described above and released them in 1.3.3 release of our software. We successfully used the context broker both in conjunction with the Science Clouds workspace deployment [2] using workspace-specific mechanisms to convey the generic context information and in conjunction with Amazon EC2, where we used the EC2 user-data to convey the generic context information. The techniques proved effective in producing "instant virtual clusters" for multiple applications, including the high energy physics STAR experiment, which ran on 100 nodes on Amazon EC2. Below we describe some contextualization examples.

4.1. Network File System (NFS)

Contextualizing NFS enables us to dynamically deploy a simple cluster with a shared filesystem. We show here a simple example of contextualizing a cluster that has two client nodes and one NFS server node that exports directories. To keep the example simple, we assume that the network can be trusted for authentication, that node's identity is composed of the IP address only, and that the volumes to export and mount are embedded in the VM contextualization scripts.

NFS VMs are all configured in the same way; contextualization consists of annotating which VM will play the server role and which will be the clients. For example, server node context template looks like this (the client's template looks similar but with the nfs labels switched):

```
<provides>
  <identity />
  <label>nfsserver</label>
```

```
</provides>
<requires>
  <role name=”nfsclient”/>
</requires>
```

During deployment, the identity playing the “nfsserver” role will be filled in once it is known, for example, with the IP address of 10.0.0.1. An XML representation of the node’s provides section in the context will now look like the following.

```
<provides>
  <identity><ip>10.0.0.1</ip></identity>
  <label>nfsserver</label>
</provides>
```

Similarly, the client nodes’ IP addresses (in our example, 10.0.0.2 and 10.0.0.3) are filled into their respective templates as they become known.

The information gets sorted and given to each client node as follows.

```
<requires>
  <role name=”nfsserver”>10.0.0.1</role>
</requires>
```

and to the server as follows.

```
<requires>
  <role name=”nfsclient”>10.0.0.2</role>
  <role name=”nfsclient”>10.0.0.3</role>
</requires>
```

The scripts on the client nodes take the “nfsserver” IP address and use it to construct the proper line to add to the fstab file. The scripts on the head node take each “nfsclient” IP address and append an authorization line to the exports policy file. Then, the server process and client mounts (depending on the role) are started. Since we do not assume that the NFS server will be online when the client node’s contextualization retrieval completes for each mount requirement, the NFS client nodes try to mount the volume in a loop that checks whether the mount was successful.

4.2. STAR Cluster

To support STAR [16] workloads, we created a virtual cluster using a Scientific Linux 4.4 base image, VDT [17] packages, and Torque [18]. The OSG 0.6.0 CE installation was used for the head node template image, and the OSG 0.6.0 wn-client installation was used for the worker node template image. The virtual head node runs a Globus GRAM2 job gateway to a localhost Torque server, a GridFTP server [19], and an NFS server. The worker nodes run Torque MOM processes (processes that sit on each worker node to run jobs) and mount NFS directories from the head node (OSG's typical $HOME, $APP, and $DATA). The head node has two network interfaces, one for its Internet addressable processes (GRAM, GridFTP) and one for a private network. The worker nodes have one interface each, all on the private network. This is a typical Grid cluster gateway + NAT setup.

The contextualization demands in this example are more complex. Since the headnode has both a public and private IP address, we have to be careful that it is the private address of the cluster headnode that gets connected to the worker nodes for NFS and Torque. Also, using Torque requires contextualization features that are new in this example, namely, full identity distribution (including SSHd host keys). The contextualization process is also used to late-configure GRAM and GridFTP to handle identity configuration (they both need to be configured with the proper public facing fully qualified domain name).

The headnode’s two network identities are both reflected in the provides section (as shown below) in order to introduce tags for each. The eth1 tag is given to the private interface, and this is indicated in the “torqueserver” and “nfsserver” provided roles. Hence, anything requiring a match for these roles will get the eth1 network identity in response.

```
<provides>
  <identity>
    <interface>eth0</interface>
  </identity>
  <identity>
    <interface>eth1</interface>
  </identity>
  <role interface="eth1">
     torqueserver</role>
  <role interface="eth1">nfsserver</role>
</provides>
<requires>
  <identity />
    <role name="torqueclient"
      hostname="true" pubkey="true" />
    <role name="nfsclient" />
</requires>
```

In the requires section, the “torqueclient” annotation indicates that more than the IP address is necessary. The hostname is required as well as the SSHd host key because SSHd host-based authentication is used with Torque and GRAM2 to allow jobs to run. This sets up free SSH access from node to node if the source and target system account are the same. Again, the worker node annotations are similar, with the role labels reversed (but no dual networking).

On boot, each worker node generates an SSHd host key, and the agent reports this to the contextualization service (using secure channel) with the rest of identity information.

After the contextualization information has been retrieved, SSHd on all nodes is configured by populating the node’s global “known_hosts” file as well as the “hosts.equiv” file (we implement a “many-to-many” approach to handle nonserial workloads where there will be intercommunication among the nodes). Further, the /etc/hosts file on each node is populated with all known IP address and hostnames. This avoids any DNS problems if the site has not configured things correctly, especially for reverse DNS, which typically affects network security software. On the worker nodes, the Torque "server" file is populated with the head node hostname, and NFS is configured as in the previous example. On the headnode, Torque's "nodes" file is populated with all of the authorized MOM hostnames, Torque's "server" file is populated with the head node hostname (configuring itself as the master), and NFS is configured as in the example before.

GRAM and GridFTP require the public fully qualified domain name of the intended contact address in order to work correctly with GSI. We found that on multi-NIC nodes this was not trivially deduced in a startup script. Thus, the contextualization engine helps identify the proper hostname for configuring these components. On EC2, the public address is not even an actual on-board network interface (each EC2 VM is behind a NAT and the public address is known only via EC2 instance metadata), and so this was especially useful in that case.

5. Related Work

One approach to appliance deployment is to only partially rely on preconfigured images. In this approach, an appliance is deployed by deploying an image with as much of a base configuration as possible ("golden image") and installing applications on the fly. This approach has been used by VMPlant [3] as well as [4]. While for this only generic contextualization is sufficient, it makes the appliance deployment potentially lengthy.

The term virtual appliance was introduced by Sapuntzakis and Lam [10], and their work describes the first attempts at defining contextualization information as well as explaining the requirements for appliance management. We build on this work, generalizing the method and enabling the use of generic tools and protocols for configuration management.

Configuration management tools such as LCFG [20], Quattor [21], and Bcfg2 [22] are somewhat similar to appliance creation and deployment. However, they rely on traditional configuration techniques and do not (as of now) cleanly separate the process of appliance creation and contextualization. Much work has also been done in the industry by companies that explicitly manage appliances (e.g., rPath [23]); we collaborate with those efforts as builders of deployer-side software. In particular, the Open Virtualization Framework [24] defines high-level concepts and best practices similar to the work described here; our approach is more detailed and serves the specific needs of our community.

6. Future Directions

While our current approach allows us to solve current problems (namely, provide a cluster on the fly for nontrivial applications), it needs to be refined to provide more features. While our implementation currently operates on identity information we see increasing demand for the exchange of application-specific data that could be brokered as "blobs" to be interpreted by contextualization agents; we are currently generalizing the techniques described here to accommodate this requirement. Also, virtual clusters are only one type of context; in general, virtual constructs could span the range from individual VMs through clusters to virtual Grids that could potentially benefit from a hierarchical organization.

Further, our methods to date do not address the critical issue of recontextualization: redistributing the context based on dynamically changing context information. The ability to do so would allow us to add VMs to a context on the fly, for example, by adding new nodes to an MPI computation, or account for changes due to, for example, appliance migration. The ability to make those changes, however, will require tighter collaboration with OS-level tools.

7. Summary

In this paper we described a new technique, called contextualization, enabling the dynamic creation of functioning virtual constructs aware of their context. We discussed two existing implementations providing generic contextualization information, their respective assumptions and capabilities, and gave examples ofow they can be used in conjunction with a context broker to deploy virtual clusters. Our purpose in this paper was to describe a general solution and a process that can be used with any deployer and any appliance provider that fulfill the specified conditions of secure transfer of information. Based on this process, we highlighted the need for standards on the deployers and appliance provider's side.

Making contextualization an accepted technology will require the collaboration of many branches of technology. Besides the obvious ones of appliance configuration and deployment, better and more flexible methods of context information delivery to appliances will need to be developed to allow for recontextualization. Further, applications will also need to develop the awareness of the potential of contextualization in order to leverage it.

Acknowledgements

This work was supported by NSF CSR award #527448 and, in part, by the MCS Division subprogram of the Office of Advanced Scientific Computing Research, SciDAC Program, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

References

1. *Amazon Elastic Compute Cloud (Amazon EC2)*: http://www.amazon.com/ec2.
2. *Science Clouds*: http://workspace.globus.org/clouds/.
3. Krsul, I., A. Ganguly, J. Zhang, J. Fortes, and R. Figueiredo. *VMPlants: Providing and Managing Virtual Machine Execution Environments for Grid Computing*. in *SC04*. 2004. Pittsburgh, PA.
4. Nishimura, H., N. Maruyama, and S. Matsuoka, *Virtual Clusers on the Fly -- Fast, Scalable and Flexible Installation.* CCGrid, 2007.
5. Keahey, K., I. Foster, T. Freeman, and X. Zhang, *Virtual Workspaces: Achieving Quality of Service and Quality of Life in the Grid.* Scientific Programming Journal, 2005.
6. Bradshaw, R., N. Desai, T. Freeman, and K. Keahey. *A Scalable Approach to Deploying and Managing Virtual Appliances*. in *TeraGrid 2007 Conference*. 2007. Madison, WI.
7. *Amazon Elastic Compute Cloud (EC2)*: www.amazon.com/ec2.
8. *Virtual Workspaces*: http://workspace.globus.org.
9. Foster, I., C. Kesselman, and S. Tuecke, *The Anatomy of the Grid: Enabling Scalable Virtual Organizations.* International Journal of Supercomputer Applications, 2001. **15**(3): p. 200-222.
10. Sapuntzakis, C. and M.S. Lam. *Virtual Appliance in the Collective: A Road to Hassle-free Computing*. in *9th Workshop on Hot Topics in Operating Systems*. 2003.
11. *rBuilder Online*: http://www.rpath.com/rbuilder/.
12. Freeman, T., K. Keahey, I. Foster, A. Rana, B. Sotomayor, and F. Wuerthwein, *Division of Labor: Tools for Growth and Scalability of the Grids.* ICSOC 2006, 2006.
13. Freeman, T. and K. Keahey, *Flying Low: Simple Leases with Workspace Pilot.* EuroPar 2008, 2008.
14. Freeman, T., K. Keahey, B. Sotomayor, X. Zhang, I. Foster, and D. Scheftner, *Virtual Clusters for Grid Communities.* CCGrid, 2006.
15. *Amazon Elastic Compute Cloud. Developer Guide. API Version 2008-02-01*, in *available from* *www.amazon.com/ec2*.
16. *The STAR Experiment*. 2007: www.star.bnl.gov.
17. *Virtual Data Toolkit*: http://www.lsc-group.phys.uwm.edu/vdt/documentation.html.
18. *Torque*: http://www.clusterresources.com/pages/products/torque-resource-manager.php.
19. Allcock, W., *GridFTP: Protocol Extensions to FTP for the Grid*. 2003, Global Grid Forum.
20. Anderson, P. and A. Scobie. *Large Scale Linux Configuration with LCFG*. in *4th Annual Linux Showcase and Conference*. 2000.
21. *Quattor*: http://cern.ch/quattor.
22. Desai, N., A. Lusk, R. Bradshaw, and R. Evrard. *BCFG: A Configuration Management Tool for Heterogeneous Environments*. in *IEEE International Conference on Cluster Computing (CLUSTER'03)*. 2003.
23. *rPath*: www.rPath.com.
24. *The Open Virtual Machine Format (whitepaper for OVF specification version 0.9).*