Management of Virtual Large-scale High-performance Computing Systems

Geoffroy Vallée
Oak Ridge National Laboratory
`valleegr@ornl.gov`

Thomas Naughton
Oak Ridge National Laboratory
`naughtont@ornl.gov`

Stephen L. Scott
Tennessee Tech University and Oak Ridge National Laboratory
`sscott@tntech.edu`

Abstract

Linux is widely used on high-performance computing (HPC) systems, from commodity clusters to Cray supercomputers (which run the Cray Linux Environment). These platforms primarily differ in their system configuration: some only use SSH to access compute nodes, whereas others employ full resource management systems (e.g., Torque and ALPS on Cray XT systems). Furthermore, the latest improvements in system-level virtualization techniques, such as hardware support, virtual machine migration for system resilience purposes, and reduction of virtualization overheads, enable the usage of virtual machines on HPC platforms.

Currently, tools for the management of virtual machines in the context of HPC systems are still quite basic, and often tightly coupled to the target platform. In this document, we present a new system tool for the management of virtual machines in the context of large-scale HPC systems, including a run-time system and the support for all major virtualization solutions. The proposed solution is based on two key aspects. First, Virtual System Environments (VSE), introduced in a previous study, provide a flexible method to define the software environment that will be used within virtual machines. Secondly, we propose a new system run-time for the management and deployment of VSEs on HPC systems, which supports a wide range of system configurations. For instance, this generic run-time can interact with resource managers such as Torque for the management of virtual machines.

Finally, the proposed solution provides appropriate abstractions to enable use with a variety of virtualization solutions on different Linux HPC platforms, to include Xen, KVM and the HPC oriented Palacios.

1 Introduction

Virtual machines are widely used for server virtualization aiming at consolidating physical resources and, ultimately, increase the resource usage. As a result, many tools are available for the definition, deployment, and management of virtual machines (VMs) on servers or a small set of servers. Furthermore, over the past few years, two new trends appeared and were the focus of many research efforts: the deployment of cloud infrastructures, and the deployment of "virtual clusters". In fact, it has been shown that virtual clusters are an interesting solution for high-performance computing (HPC).

Even if these computational platforms are very different in nature, they are most of time running the Linux kernel, even if the Linux distribution on top of it can be very different and/or customized. For instance, most servers in the business world are running Linux, in the context of server consolidation, many VMs are Linux based. In the context of HPC, the trend is even more clear since more than 90% of HPC systems on the Top500 list [5] are Linux based.

This document is focused on the HPC context, for which it is preferable to have a few VMs running on the nodes of the HPC platforms (typically one VM per core), rather than running many virtual machines on a given core (over-subscription). This is mainly because in the context of HPC, input/output (I/O) operations are crit-

[1]The submitted manuscript has been authored by a contractor of the U.S. Government under Contract No. DE-AC05-00OR22725. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

ical (because they are involved in communication between the execution entities of a parallel application).

To the best of our knowledge, all ongoing efforts for the design and development of tools that aims at managing a large number of VMs are focusing on the “many VMs on a few nodes” paradigm rather than the “a few VMs on many nodes” paradigm. As a result, existing solutions are not adapted for the management of many VMs on HPC systems and ultimately the execution of parallel applications within these VMs.

In this document, we present the architecture of a new system-level solution for the deployment and management of many VMs that are used for the execution of large-scale parallel applications on HPC platforms. Because the primary target of the proposed system is large-scale HPC systems, the following characteristics are critical for its design:

- **Scalable bootstrapping**: the bootstrapping on many VMs running on many nodes of HPC systems is a challenging tasks, from the staging of both the VM image and the application, to the initialization of the VMs and the launch of parallel applications within the running VMs. Based on the large number of nodes and VMs, it is necessary to implement some advanced methods for startup of VMs and applications, with linear approaches being too expensive.
- **Portability**: HPC systems may have very different hardware configurations, as well as very different software configurations. Furthermore, different virtualization solutions will most certainly be deployed on different HPC systems. As a result, the proposed solution must support all major virtualization solutions, and abstract the underlying virtualization solution away from the users (so they can easily run their applications on various virtualized HPC platforms).
- **Customization**: one of the benefits of system-level virtualization is to allow users to define their own execution environments within the VMs (typically software configuration). Tools are already available for the specification and deployment of customized environments that will perfectly match the requirements of parallel applications. The proposed tool must support such customization capabilities.
- **Fault tolerance**: because the system is composed of many distributed hardware components, the probability of a failure during the execution of a parallel application at scale increases accordingly. As a result, even if the goal of this study is not to provide fault tolerance mechanisms for parallel applications, we have to ensure that the proposed system and its overall infrastructure will tolerate failures to some extent, at least to allow users to cleanly terminate the execution of their applications. For that, it is necessary to **detect failures**, and **guarantee communications** even in the context of link failures.

To address these challenges, the proposed solution is based on three different abstractions:

1. a set of tools and methods for the specification and instantiation of customized execution environments,
2. a control infrastructure that will be used for the management of VMs (startup, monitoring, termination), as well as the control of the execution of the parallel application; this “run-time” system must be scalable and fault tolerant,
3. a tool for the abstraction of the underlying virtualization solution so the user will not have to deal with technical details specific to the virtualization solution deployed on a given HPC system, which means that this abstraction must support all major virtualization solutions.

The remainder of this document is organized as follows: Section 2 presents how users can specify a customized execution environment for their applications that will be used for the deployment of VMs on HPC systems. Section 3 presents the control infrastructure used to control and orchestrate many VMs used in the context of the execution of a large-scale parallel applications. Section 4 presents the abstraction layer used to allow the user to implicitly switch between different virtualization solutions. Finally, Section 6 concludes.

2 Customization of Execution Environments

Based on the 36th edition of the Top500 list (September 2010), 91% of the 500 most powerful HPC systems are based on Linux. However, the software configuration of these systems vary greatly, from customized kernels and Linux distributions to out-of-the-box Linux distributions and the kernel they provide by default. Furthermore, each of the HPC systems have a well-defined set of available software, including scientific libraries and tools. So far, the users had to modify their application to fit the configuration of the target HPC platforms, leading to wasted resources and redundant effort (scientists should focus on the science gathered in their applications and not on modifications because of technical details of the HPC system).

An approach to address this challenge is to allow the users to define their execution environments based on the requirements of their applications. In a previous work, we introduced the concept of *Virtual System Environment* (VSE) [6] that enables the description of the software requirements of a given applications. We also proposed a set of tools for the instantiation of a VSE on a given HPC platform.

As a result, it is possible to specify the "static" requirements of the scientific application; requirements that are not specific to a given run of the application on a given platform. For instance, the user can specify requirements such as the Linux distribution to be used (e.g., Red Hat Enterprise Linux), the version of the Linux kernel, a set of scientific libraries. The specification is translated into terms of "packages" available from repositories. During the instantiation of a VSE on a given HPC platform, the list of packages is used to create a new image, which can then be used to setup a VM. Note that the tools associated with the VSE ensure that the image can be deployed independently of the virtualization solution that is ultimately used. For instance, the users do not have to know whether KVM or Xen will be used as the virtualization solution, the provided tools create a VM image that is agnostic to the different virtualization solutions. Note that the tool that abstracts the virtualization solution (presented in Section 4) ensures that the image is correctly "loaded" based on the target virtualization solution.

3 Control Infrastructure

The previous section presents how a user can customize the execution environment for their applications. This task can typically be done off-line and is independent from the target HPC system. Once on the HPC systems, the users must deploy the required VMs and start the application execution. A typical way to see the execution of a parallel application is the concept of *job*: a job is the combination of the application and an allocation for its execution (typically a set of nodes). Unfortunately, HPC systems can be used with very different configurations: some provide tools that assign an allocation to a given job (based on the number of requested nodes, the *job manager* allocates nodes to the job); while some other systems allow direct access to compute nodes (e.g., via SSH). This heterogeneity directly impacts how VMs will be deployed.

Furthermore, because we target large-scale HPC systems, it is not efficient to setup VMs and start the application execution in a linear fashion, more advanced startup methods are required.

Finally, even if failures occur during the execution of a parallel application running within VMs, we must continue to keep control on the running VMs. In the context of this study, our goal is not to provide fault tolerance capabilities for the VMs or even for the application, but to guarantee that even if compute nodes or VMs fail, it will still be possible to control remaining VMs and let the user decide the best solution (e.g., cleanly terminate VMs that are still alive and therefore, terminate the job).

3.1 Architecture Overview

In order to deploy VMs on compute nodes and control the execution of applications within these VMs, we need to have control on each compute node of the job allocation. Furthermore, in order to separate the system aspects (such as resource allocation) from the job management, the proposed architecture is based on the concept of agents, and five different types of agents have been defined: root agents (typically system agents), session agents (specific to a job), and tool agents (specific to a "tool", a tool being a self-contained part of a job, e.g., one of the binaries of a job when the parallel application is composed of different sub-applications).

- **Root agent**: agent in charge of resource allocation and release. Thus, this agent is a privileged agent. Only one root agent is on each compute node and is used to deploy other agents (both session and tool agents). Root agents are not specific to a job.

- **Session agent**: agent in charge of instantiating a job on allocated compute nodes. This is not a privileged agent and it acts on behalf of a user. A single session agent is deployed on compute nodes of a given job allocation.

- **Tool agent**: agent that instantiates the job itself; multiple tool agents can be deployed on compute nodes of a job allocation, and all tool agents act on behalf of the users. In the context of this paper, the tool agents are used to manage VMs. For that, we developed a specific tool agent that can be used to drive the tool that implements the abstraction of the underlying virtualization solution (presented in Section 4). For instance, the dedicated tool agent can instantiate a VM, pause it, or terminate it.

- **Controller agent**: agent in charge of creating an internal representation of a job and of coordinating the deployment of the different agents and the creation of communication channels between the agents. The communication channels are organized based on *topologies* (e.g., trees, meshes) that describe how the controllers, the root agents, the session agents, and the tool agents can communicate. In this example, root agents are running on different compute nodes, and both session and tool agents, that are children of a given root agent, actually run on the same compute nodes. Figure 1 presents an example of a tree-based topology. Topologies are also used to set routing tables up (which are then used to send messages from one agent to another), and to stage files, including the VM image.

- **Front-end agent**: agent that runs on the user's machine or on the HPC system login node. The front-end provides a MPI-like user interface to submit a job where the user specifies the VSE specification file and the number of nodes required.

3.2 Scalable Bootstrapping

To efficiently startup agents, we define a boot topology (the initial implementation is based on a binary tree but any k-ary tree could be used). This tree allows us to start the different agents in parallel, and provides good scalability. This approach is used in various HPC specific run-times and has proven to be efficient.

If failures occur during the bootstrapping phase, the different agents are designed and implemented to automatically terminate. This is implemented using a handshake mechanism with the agent's parent within the bootstrapping topology, as well as timers. Typically, if the handshake does not succeed within a window of time, we assume a failure and the agent terminates. On the other hand, if the handshake succeeds, the bootstrapping phase is assumed successful; the agent's state switch to running, and the parent assumes that the agent is running and reachable. As such, the failure detection is then in charge of detecting and reporting agent failures.

3.3 Fault Tolerance

For fault tolerance purposes, we provide two capabilities: failure detection and a fault tolerant topology. These two capabilities ensure that even if a node fails or if an agent fails, it will still be able to send/receive messages between agents that are still alive. This allows the user to decide the best policy to apply in the context of failure, for instance, triggering the clean termination of remaining agents and ultimately VMs.

3.3.1 Fault Detection

A key point to tolerate failures is to first detect failures. By detecting failures, it is possible to update routing tables and eventually re-establish failed communication channels to ensure that we can still control live agents. For this context, we propose a set of *detectors*. For instance, a mesh-based detector establishes connections between root agents and reports an error if the connection is closed. Another detector establishes connections between root agents based on a mesh topology and perform periodic ping-pong probes. If the ping-pong fails, a failure is reported. Finally, we provide a signal-based detector that can be used on compute nodes to detect the failure of any local session or tool agent (by catching the SIGCHLD signal).

3.3.2 Communication Fault Tolerance

Since our boot topology is a tree-based topology, the failure of any agent will prevent communications between different parts of the tree, leading to unreachable agents. To address this issue, we setup a topology based

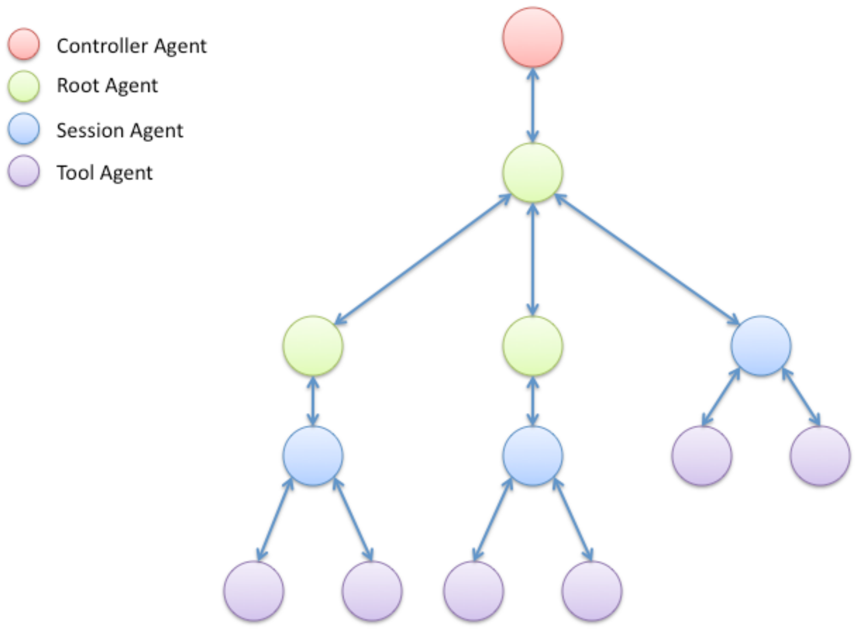

Figure 1: Example of Topology

on a binomial graph (BMG) [1] that provides redundant communication links between agents. As a result, even if a communication channel is closed because of a failure, it is possible to find another route to reach the destination.

4 Abstraction of the Underlying Virtualization Solution

Many tools are available for the management of virtual machines, such as libvirt [3], However, these tools, to the best of our knowledge, try to represent the union of all the capabilities of all the virtualization solutions, leading to overly complex tools. In the context of our study, we only require a lightweight tool that provides a simple API, typically start, stop, pause, un-pause a given VM (we may support migration in a future version of the system). For that, we propose the V2M tool from a previous study [7]. This tool abstracts the underlying virtualization solution via the implementation of plug-ins. Each plug-in is in charge of translating management tasks to commands that are specific to the underlying virtualization solution. The tool is also in charge of making sure that the VM image is correctly setup to be used with the target virtualization solution.

V2M is based on the concept of *profiles*, which specify how to deploy a VM based on VSE image. This profile is automatically created based on job data such as the allocation specification.

5 Use Case: the Palacios Virtualization Solution

Palacios is a virtualization solution specifically designed for HPC [2, 4]. For that, Palacios is focusing on minimizing its resource footprint and optimizing I/O (since efficient I/O is critical for HPC applications). Figure 2 presents an overview of the architecture for Palacios.

Palacios proved to be very scalable and is therefore a good candidate for experimentation at scale. In other terms, by selecting Palacios, we can setup an experimental configuration that is scalable and fault tolerant.

To support Palacios, a new V2M plug-in is created in order to interface with our infrastructure; no other modifications or extensions are required.

6 Conclusion

In this document we present the architecture for a new system-level infrastructure for the management of many virtual machines to support the execution of parallel applications on large-scale high-performance computing

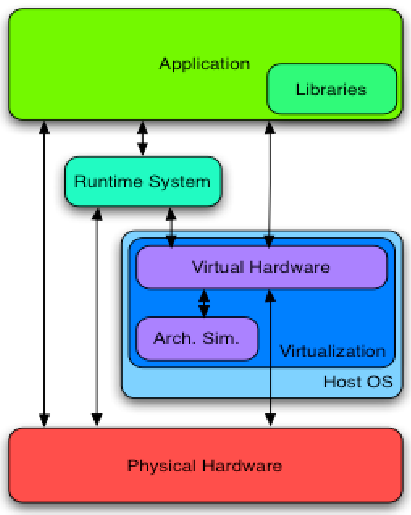

Figure 2: Overview of the Architecture of the Palacios Virtualization Solution

systems. The proposed architecture focuses on scalability and fault tolerance. Furthermore, the proposed solution abstract the underlying virtualization solution and can therefore be used with most of the current virtualization solutions such as Xen or KVM. Finally, our solution enables the customization of the execution environment that is deployed inside the virtual machines, which ultimately allows scientists to focus on science rather than technical details associated with the configuration and execution of their application on elaborate high-performance computing systems.

To implement these capabilities, we propose three distinct abstractions: the concept of VSE for the customization of the execution environment; a scalable and fault tolerant control infrastructure for the coordination of the VMs running across the compute nodes, and finally an abstraction of the underlying virtualization solution.

The implementation of the proposed architecture is still ongoing but initial experimentation shows that the design of the control infrastructure is scalable and maintains connectivity between the different nodes involved in the execution of a given application even in the event of failures.

Finally, Palacios, a virtualization solution designed for high-performance computing, is able to scale to a few thousand VMs, and we are working with the Palacios development team to perform experiments at scale using our tool on some of the world's larger HPC systems.

Acknowledgments

We would like to thank the Application Performance Tools group from ORNL, and especially Richard L. Graham, for their support on the development of the control infrastructure.

References

[1] Thara Angskun, George Bosilca, and Jack Dongarra. Binomial graph: A scalable and fault-tolerant logical network topology. In *International Symposium on Parallel and Distributed Processing and Applications*, pages 471–482.

[2] John Lange, Kevin Pedretti, Trammell Hudson, Peter Dinda, Zheng Cui, Lei Xia, Patrick Bridges, Steven Jaconette, Mike Levenhagen, Ron Brightwell, and Patrick Widener. Palacios and kitten: High performance operating systems for scalable virtualized and native supercomputing.

[3] The virtualization api. http://libvirt.org/.

[4] Palacios – an os independent embeddable vmm. http://v3vee.org/palacios/.

[5] Top 500 supercomputer sites. http://top500.org/.

[6] Geoffroy Vallée, Thomas Naughton, Hong Ong, Anand Tikotekar, Christian Engelmann, Wesley Bland, Ferrol

Aderholdt, and Stephen L. Scott. Virtual system environments. In *Systems and Virtualization Management. Standards and New Technologies*, volume 18 of *Communications in Computer and Information Science*, pages 72–83. Springer Berlin Heidelberg, October 21-22, 2008.

[7] Geoffroy Vallée, Thomas Naughton, and Stephen L. Scott. System management software for virtual environments. In *Proceedings of ACM Conference on Computing Frontiers 2007*, Ischia, Italy, May 7-9, 2007.