Managing the Virtual Machine Lifecycle of the CernVM Project

I Charalampidis, J Blomer, P Buncic, A Harutyunyan and D Larsen
CERN, CH-1211 Genève 23, Switzerland
E-mail: ioannis.charalampidis@cern.ch

Abstract. CernVM is a virtual software appliance designed to support the development cycle and provide a runtime environment for LHC applications. It consists of a minimal Linux distribution, a specially tuned file system designed to deliver application software on demand, and contextualization tools.

The maintenance of these components involves a variety of different procedures and tools that cannot always connect with each other. Additionally, most of these procedures need to be performed frequently.

Currently, in the CernVM project, every time we build a new virtual machine image, we have to perform the whole process manually, because of the heterogeneity of the tools involved. The overall process is error-prone and time-consuming. Therefore, to simplify and aid this continuous maintenance process, we are developing a framework that combines these virtually unrelated tools with a single, coherent interface. To do so, we identified all the involved procedures and their tools, tracked their dependencies and organized them into logical groups (e.g. build, test, instantiate). These groups define the procedures that are performed throughout the lifetime of a virtual machine.

In this paper we describe the Virtual Machine Lifecycle and the framework we developed (iAgent) in order to simplify the maintenance process.

1. Introduction

CernVM is a Virtual Software Appliance that provides a complete, portable and easy to configure user environment for developing and running LHC data analysis locally and on the grid, independently of Operating System software and hardware platform. CernVM is based on the Scientific Linux (SL) distribution and is extended with our own additions in order to make it friendly to virtualized environments.

The development, maintenance and deployment of such virtual machines involves several steps, ranging from the maintenance of a Linux distribution, to the quality assurance of the final VM image, or its dynamic contextualization at run time. The effort required to maintain the virtual machine grows rapidly with the size of the project. A good strategy to overcome this scaling problem is to unify all the involved tools in order to enable tuning and orchestration from a single interface.

For the CernVM project, we tried to perform this unification process in two steps: Firstly we identified all the procedures involved in the maintenance process – along with their tools – and then we designed a coherent framework (iAgent) that joins them together. The involved procedures form the Lifecycle of the virtual machine.
2. The Virtual Machine Lifecycle

A virtual machine passes through various different stages throughout its life. These stages are just a logical separation of the fundamental procedures that are common for the maintenance of every virtual machine (VM). They are usually independent and are associated with a specific set of tools. For instance, the life of the VM begins when the specifications of the build process are prepared and stored in a reference database, and it is terminated after it has completed the job it was instantiated for. In order to find an optimal solution it is important to identify those stages, the tools associated with them and their dependencies. This way the appropriate tools can be grouped with the stages and form a stand-alone and independently-maintained component.

In the CernVM Project we pass through all stages every time we release a new version. In our case, after a VM instance completes its cycle, user feedback is processed and a new development cycle begins. Because of this cycling pattern, we decided to use the term lifecycle to refer to the life of CernVM. This lifecycle can be split into two logical sub-cycles: the development cycle and the deployment cycle (Figure 1).

The development cycle begins with the definition of the specifications and finishes with the production of the distributable VM media. This cycle is performed entirely inside the CernVM infrastructure.

The deployment cycle begins with the instantiation of the released image and finishes with the termination of the instance. This cycle is performed outside the CernVM infrastructure, such as a public or private cloud infrastructure (e.g. Amazon or OpenNebula) or an individual computer (e.g. desktop hypervisors or a small computer farm). In all these cases, the OS needs to contact the CernVM infrastructure in order to obtain contextualization information and software packages from our repository.

The two cycles are connected via two intermediate stages: the release of the produced image to the public and the feedback that is collected from the users and triggers a new development cycle. The two stages are in the borders that split the private infrastructure and the public.

As was mentioned before, each stage is independent and is typically supported by a number of specialized tools.

Plan: This is a stage at which the desired functionality of the VM is planned. The resulting
information could be the choice of the base OS, the packages to install etc. Information can be also stored in a database and used as a reference for the next stages.

**Prepare repositories:** In this stage the materials for the new virtual machine are collected and placed in the operating system’s software repository. For CernVM we first synchronize our repository with the Scientific Linux *upstream* repository and then we build and add the CernVM-specific packages, such as the CernVM Filesystem (CVMFS) [1] and virtualization additions. In the end we define one software group for every *flavor* of CernVM (desktop, head node, batch node, basic, etc.) that describes the appropriate set of packages that needs to be installed. For CernVM we are using *Conary*¹[2] as our package manager and repository server.

**Build:** During the build stage the virtual hard disk image is generated. During this process the packages from the previously defined software groups are installed into a new VM image. Then that image is converted into different formats, compatible for most of the widely-used hypervisors, and packaged along with the appropriate definitions. For the production of CernVM, we have developed our own build system that uses the *Conary* package manager to install the software and *QEMU* image utilities² for converting the disk image into various formats.

**Test:** At the test stage all the previously built images are checked for errors using an automated testing software. This software instantiates and configures a virtual machine exactly like an end-user is expected to do, and runs a set of functionality tests. The testing framework is based on the *Test Anything Protocol (TAP)*³ by AMD, adapted to the needs of testing a virtual machine.

**Endorse:** The endorse stage involves publishing the produced images to the public repositories (e.g. CernVM portal) and cloud marketplaces (e.g. Amazon AWS Marketplace). Additionally, according to HEPIX specifications [3] [4], a list with the cryptographic hashes of the released images is generated and digitally signed. The signature of the list expires after a predefined timeout and acts as an authenticity signature. The tools required at this stage are a custom utility to maintain the endorser list and an uploading utility that is compatible with the Amazon S3 and EC2 protocols.

**Instantiate:** The instantiate stage starts when a user is starting a CernVM instance, either on his computer or on a bigger cloud infrastructure. At this stage, the tools needed are a hypervisor and a utility to communicate with it.

**Contextualize:** The contextualize stage takes place inside the VM during the first boot [5]. For contextualization, CernVM uses the *Amiconfig* tool. CernVM supports both contextualization via UserData field for Amazon/EC2 or similar infrastructures (e.g. CloudStack), and contextualization via CD-ROM images for small clusters (e.g OpenNebula). This dual contextualization technique has been accepted by the HEPIX community [4].

**Monitor:** If the image is contextualized with monitoring support, a monitor agent can provide real-time machine health information. This information can either be observed via a centralized monitoring interface or can be used to trigger automated tasks. The iAgent framework provides the skeleton upon which such monitoring agents can be developed.

**Terminate:** This is a stage the VM enters when the user has finished using a CernVM instance and shuts it down.

**Retire:** This is the final stage in the lifetime of a CernVM image. At this stage a published VM image is removed from the public. This can occur due to security, interoperability or policy issues.

**Feedback:** During this stage the feedback from the users and the logs from the monitoring agents is collected and processed. Based on it, the specifications for the new release are set and a new development cycle begins.
3. The iAgent Framework

As we have seen, the maintenance of the virtual machine involves many different processes and many different tools, virtually unrelated. In reality however, they are all part of the same job. Moreover, most of these processes are logically interconnected, but physically, there are no obvious means to connect their underlying tools.

In the CernVM project the maintenance of our virtual machine required a reasonable effort, since CernVM was initially an R&D project with limited resources. This led us to develop a system that can simplify and automate the maintenance process. Our main goal was to create a specialized agent for every stage of the lifecycle of the virtual machine. These agents could then be connected with each other and used from a single point. To aid the development of such agents, we developed a framework called iAgent (Figure 2).

The iAgent framework provides a skeleton to develop adapters between any kind of existing tools and a common message interface. And with all the tools accessible with the same API, the framework provides some additional features:

- It provides a unified, extensible user interface
- It provides a unified command line interface for scripting purposes
- It provides a distributed, server-less workflow engine (given the XMPP server), with automatic resource discovery and with minimal additional configuration.

In short, the iAgent framework not only provides the means to monitor and use all the tools from a single interface, but also provides the means to chain these tools with a workflow definition.
3.1. Technical details

The iAgent framework is divided into two parts, the agent framework and the user interface framework. Both of them are using Extensible Messaging and Presence Protocol (XMPP) [6] as their communication channel. We chose XMPP because it is widely used, well tested in high load (e.g. Google Talk), and easily extensible. Additionally, most of the XMPP server implementations provide an out-of-the-box support for server clustering (for example ejabberd4).

Both frameworks use three kinds of XMPP messages for communication:

- For the core communication API, XMPP Info-Query (IQ) messages are used. This is because IQ messages can co-exist with other kinds of messages (for example Instant Messages) and the IQ protocol definition enforces a response from the remote endpoint, requiring no additional transfer monitoring code.
- The framework also provides a command-line interface via standard XMPP Instant Messages (IM)
- For the resource discovery, used by the workflow system, the XMPP Publish-Subscribe mechanism is used [7].

3.2. The agent framework

The agent framework is written in perl, has a pluggable architecture and provides the skeleton upon which the agent logic is built. It consists of a small kernel and a collection of utility modules. The kernel is responsible only of loading the modules, monitoring their health and routing the internal API messages between them. The logic is implemented entirely in the modules (Figure 3).

The utility modules hide from the developer the complexity of most of the operations that need to be performed. They either perform automated tasks using some simple configuration variables (e.g. the workflow modules – it requires only some pointers to the functions exported as workflow actions) or they provide an abstraction layer for other protocols (e.g. the XMPP module that translates the XMPP messages into protocol-independent kernel API messages). Either way, they offer a reusable logic, accessible through the simple kernel API.

Figure 3: The architecture of an agent that uses the iAgent framework.
The kernel extends the single-threaded multitasking functionality of the POE (Perl Object Environment)\textsuperscript{5} framework \textsuperscript{8}, by providing a new message broadcast and broadcast reply mechanism. Each module runs in a separate POE Session and has a priority preference. The kernel loads the modules in a module stack, trying to satisfy this priority preference by placing the ones with higher value on top and the ones with lower value on bottom. The broadcasting works by sending API messages to the entire stack, starting from top and propagating them to the bottom. Each module is allowed to alter the message and/or stop the broadcast (Figure 4).

The broadcast reply works in the same way, but the propagation starts from the module that received the message and continues towards the module that initiated the broadcast.

![Module Stack Diagram]

Figure 4: The iAgent's module stack with a visual representation of how a message is broadcasted using the kernel's Dispatch routine and how a reply is propagated back to the sender.

This filter pattern enhances the extensibility of the agent, since the modules can modify existing functionality without changing the source code. For example, the authentication module resides right after the communication module (e.g. XMPP) and intercepts the API messages it broadcasts when a user command arrives. The module then validates the signature of the sender and it either stops the propagation of the message (in case of denied access) or it appends an array of granted permissions in it, making them available to the rest of the modules that will receive this message later on. In the same way, when a module wants to reply to a previously broadcasted message, it uses the broadcast reply mechanism that propagates the message again by the authentication module that appends the sender signature in the message.

This design encourages the development of independent, reusable modules that can complement the functionality of each other. Along with the kernel, the following utility modules are provided by the framework:

\textbf{iAgent::Module::XMPP:} This module implements the communication mechanism. It has the highest priority and translates XMPP messages into abstract communication commands for the module API.

\textbf{iAgent::Module::LDAPAuth:} This module resides right after the XMPP module in the module stack and implements the authorization mechanism. It intercepts the messages the XMPP module dispatches and validates the identity of the sender against an LDAP server.
If the user is valid, it injects the user permissions in the message. Otherwise, it just rejects the message.

**iAgent::Module::WorkflowAgent:** This module implements the distributed workflow engine of the iAgent. It receives workflow-specific messages from the XMPP module and calls the exposed workflow actions via the WorkflowActions module.

**iAgent::Module::WorkflowActions:** This module exports the workflow actions defined by other modules to the WorkflowAgent module.

**iAgent::Module::WorkflowServer:** This module provides a service that monitors the progress of an ongoing workflow. It also contains a database were workflow definitions can be stored. This module is not critical for the actual workflow execution.

### 3.3. The workflow subsystem

The workflow subsystem has two powerful assets:

Firstly, it requires no centralized orchestrator in order for it to work. Each entity can act both as a workflow endpoint and as a workflow orchestrator. These two algorithms are implemented in two utility modules in the agent framework and can operate concurrently: the *Module::WorkflowActions* and the *Module::WorkflowAgent* respectively.

A workflow can be started from any agent by preparing a workflow definition and sending the appropriate API message to the *Module::WorkflowAgent* module. The agent will then act as a workflow orchestrator i.e. it will pick the first action in the workflow and it will look for targets capable of handling it. The agent will then invoke the action on the remote target and monitor it’s progress. If anything goes wrong both the orchestrating and the target agents are capable of detecting the error and recovering from it.

When an agent receives an invocation request for a workflow action it exposes, the *Module::WorkflowActions* takes control. It runs the appropriate function and collects it’s output and result code. In this mode, the agent reports frequently to the orchestrating entity about it’s status. If the agent does not respond in time it is assumed to be faulty.

After the completion of the action the orchestrating entity passes the control to the target entity. The target entity then switches into orchestrator mode and the workflow continues like this.

Secondly, the workflow subsystem requires no predefined topology, since for each action in the workflow an appropriate target will be discovered at the moment of its execution. This provides great elasticity since resources can be added or removed at run-time without interrupting active workflows.
Each workflow definition can be represented by a graph, as illustrated in figure 5. Each node represents the action that needs to be performed and each connection represents the decision to be taken based on the return code of the action.

There is an additional kind of node, the fork node. This node tells the engine to start a number of predefined workflows in parallel. The parent workflow will monitor the forked workflows and will wait until the last one is completed.

In order to be robust and reliable even in high load, the workflow engine is implemented as a Finite-State-Machine (Figure 6). There is a limited number of allowed running instances of the engine, and a timer checks the state change conditions for these instances in a fixed time interval. This way unwanted CPU load is avoided.

When a new workflow starts an empty context is created and a new workflow engine is instantiated. The engine picks the first action in the definition and enters the Scheduled state. There, the engine checks if a module in the same agent exports that action. If yes, it starts it. Otherwise it broadcasts a lookup request and enters the Lookup state. The lookup request contains the name and the context of the action in order to let the targets do the appropriate validation.

If no agent responds in time, even after a fixed number of retries, the workflow is considered impossible and the engine enters the Failed state. Otherwise the workflow definition and the current workflow context are passed to the remote target and the engine enters the Invoked state.

The engine will then wait in the Invoked state until it gets feedback from the remote endpoint that the action is completed. Then, it enters the Observe state and waits until the remote endpoint has finished with the workflow. This state is important in order get the final return code of the workflow.

After the action has been executed locally during the Run state, or remotely during the Observe state, the engine enters the Continue state. The next node in the workflow is selected, based on the result of the previous action(s). If there are no nodes left, the workflow is completed, the engine enters the Dead state and is scheduled for deletion. Otherwise, the engine purges the visited nodes from the workflow definition graph, starts a new instance of the workflow engine (or multiple instances if a parallel job is required) with the new definition and enters the Wait state.
Figure 6: The states of the workflow engine’s internal Finite-State-Machine.

There it will wait until all the child workflows are Dead and then notify all the interested entities that the workflow is completed.

The workflow subsystem provides an multiple error detection and failover mechanisms. Every operation is expected to finish in a strict time frame and without system errors. If it fails to comply, depending on the kind of the failure, a single workflow action or the entire workflow can be restarted or marked as unreliable and stopped.

Additionally, every workflow instance has a unique ID. This way, if an instance fails and the workflow is restarted the old instance will be ignored by default, even if it manages to recover at a later time.
Finally, the workflow progress can be monitored by multiple entities. By default, each entity
notifies the parent (orchestrating entity) about its progress. However it is possible to set in the
workflow definition any number of additional entities that will receive progress information during
the execution of the workflow. This way a user can easily monitor or debug the whole process
since the required hooks are already provided.

3.4. The user interface framework
For the main user interface of the iAgent framework we are using the front-end of the Archipel
Project\textsuperscript{6}, since it is sharing many similarities with the iAgent architecture. The front-end is a
web application written in Objective-J that runs in a web browser. For the connection with the
XMPP server it is using Bidirectional Streams Over Synchronous HTTP (BOSH)\textsuperscript{9}.

The Archipel front-end provides complete functionality but is too heavy to run on tablets
or mobile devices. That’s why we developed an additional, lightweight client (called PicoClient,
Figure 7a). This client is developed in HTML5 and Javascript, using the Scencha Touch\textsuperscript{7}
framework.

Both clients are stand-alone web applications that require no server-side scripting. They
have a modular design and they provide a basic skeleton upon which the user can build his own
additions.

4. Status and future plans
The project is still in an early beta state. So far we have a stable kernel and a beta version of
the workflow engine. We also created agents for the configuration and build stages of the lifecycle,
including some helpers, such as a storage agent and an automatic VM instantiation agent.

For our beta tests we managed to successfully build a CernVM release in our private cloud
infrastructure (Apache CloudStack\textsuperscript{8}). For this test four different kind of agents were used:
A Workflow Server (Module::WorkflowServer) was running in one machine and was monitoring the workflow progress.

A Cloud Agent (Module::CernVM::Cloud) was running in another machine and was communicating with our private cloud infrastructure using the CloudStack API.

A Storage Agent (Module::CernVM::Storage) was running in a CernVM instance, created by the cloud agent. This agent provides upon request a CHIRP export and a limited-time key to access this resource.

Three Build Agents (Module::CernVM::Build) were running in three different CernVM instances created by the cloud agent. These agents downloaded the required packages from our private repository, built the CernVM virtual disk images and then uploaded the resulted files to the storage agent.

A workflow was then defined via the Archipel GUI on the Workflow Server with the following six nodes:

1. Run 'cloud::instance' with the appropriate configuration to start a storage agent.
2. Run 'cloud::instance' again with the appropriate configuration to start 3 build agents.
3. Run 'storage::allocate' to allocate a storage slot and obtain the access credentials.
4. Fork node that creates three instances of 'iagent::build' with the appropriate configuration to build a CernVM version.
5. Run 'iagent::notify' with a success message.
6. Run 'iagent::notify' with a failure message.

All nodes were using the following routing rules:

- If result is non-zero (failure), go to node 6.
- If result is zero (success), go to the next node.

The workflow was started using the Archipel GUI from the Workflow Server and it’s progress was monitored from the same interface. The first two nodes (cloud::instance) were handled immediately, because the cloud agent was already running, while the next two (storage::allocate and iagent::build) were re-broadcasted a couple of times until the VM instances were booted and ready to handle the request. The iagent::notify messages were handled by a fifth agent (Module::Blinker) that was running in another machine and was producing a success or a failure sound depending on the parameters given.

In order to test the stability of the system we forcefully destroyed random build nodes. The workflow system was capable of detecting the error and re-broadcasting a new build request within a couple of minutes.

Overall the tests were successful, but both the agents and the workflow subsystem must be further tested in order to track down all the possible errors this agnostic design might incorporate.

Our plan for the near future is to connect this system with our Long-term preservation of analysis software environment [10] project, in order to enable on-demand builds of older CernVM versions.

Some other important additions to the iAgent framework is the support for automated VM deployment, as well as on-the-fly contextualization of newly started VM instances.

We plan to add more features after performing tests of the system at a larger scale and receiving more feedback from users.
References
[1] Blomer J, Aguado-Sanchez C, Buncic P and Harutyunyan A 2011 Journal of Physics: Conference Series 331
[2] Johnson M 2005 Linux Symposium Proceedings - Building Linux Software with Conary
[3] 2010 Policy Trusted Virtual Machines - CERN EDMS - Document 1080777 v.1 URL https://edms.cern.ch/document/1080777
[4] CASS T 2012 Hepix wg report Talk - Virtualized WNs and Cloud Resources URL https://indico.cern.ch/contributionDisplay.py?contribId=4&confId=155068
[5] Buncic P, Aguado-Sanchez C, Blomer J and Harutyunyan A 2011 Journal of Physics: Conference Series 331 052004
[6] P Saint-Andre E 2011 Extensible Messaging and Presence Protocol (XMPP): Core URL http://tools.ietf.org/html/rfc6120
[7] Peter Millard Peter Saint-Andre R M 2010 XEP-0060: Publish-Subscribe URL http://xmpp.org/extensions/xep-0060.html
[8] Cozens S 2005 Advanced Perl Programming O'Reilly Series (O'Reilly Media) ISBN 9780596004569
[9] Ian Paterson P S A 2010 XEP-0206: XMPP Over BOSH URL http://xmpp.org/extensions/xep-0206.html
[10] Larsen D T 2012 Long-term preservation of analysis software environment - to appear in Proc. of the 19th Int. Conf. on Computing in High Energy and Nuclear Physics (CHEP’12)

Notes
1 Conary software package manager from rPath - http://wiki.rpath.com/wiki/Conary
2 QEMU image utilities are part of QEMU open source machine emulator and virtualizer - http://www.qemu.org/
3 AMD Tapper - An open-source infrastructure for all aspects of testing including Operating Systems and Virtualization - http://developer.amd.com/zones/opensource/AMDTapper/Pages/default.aspx
4 Ejabberd - an open-source Jabber/XMPP instant messaging server- http://www.ejabberd.im/
5 POE is a portable multitasking and networking framework for any event loop - http://poe.perl.org/
6 Archipel - XMPP based orchestrator - http://archipelproject.org/
7 Scencha Touch - Mobile App Development Platform - http://www.sencha.com/products/touch
8 Apache CloudStack is an open-source, large-scale virtual machine network management infrastructure - http://cloudstack.org/
9 Chirp is a user-level file system for collaboration across distributed systems such as clusters, clouds, and grids - http://nd.edu/~ccl/software/chirp/