MIKELANGELO

D4.7
First version of the application packages

Workpackage	4	Guest Operating System
Author(s)	Gregor Berginc	XLAB
	Daniel Vladušič	XLAB
Reviewer	Michael Gienger	HLRS
Reviewer	Joel Nider	IBM
Dissemination Level	PU	

Date	Author	Comments	Version	Status
2015-11-03	Gregor Berginc	Initial document structure	V0.1	Draft
2015-11-20	Gregor Berginc	Description of legacy systems and the changes made to Capstan	V0.1	Draft
2015-12-01	Gregor Berginc	Presentation of MIKELANGELO packages	V0.2	Draft
2015-12-04	Daniel Vladušič	Informal review and improvements	V0.3	Draft
2015-12-09	Gregor Berginc	Final changes and document clean up before the review	V0.4	Review
2015-12-24	Gregor Berginc	Document ready for submission	V1.0	Final

© Copyright Beneficiaries of the MIKELANGELO Project

Executive Summary

Application packaging and package management systems have been around ever since the advent of general purpose (open source) operating systems. Different standards and approaches have emerged trying to solve common pain points of deploying and maintaining packages on target systems. Lately the demand for efficient deployment and management of nearly autonomous application stacks has increased significantly. It has thus become even more apparent that traditional mechanisms for installing and running applications will need to be revised to deliver ready-to-use applications and appliances to Cloud and HPC users.

This deliverable touches this area from the perspective of the lightweight unikernel operating system OSv. Two competing solutions have been provided even before the MIKELANGELO project has started. However, both of them suffer from significant limitations preventing wider adoption by end users. To this end, the application management task of MIKELANGELO strives to extend existing capabilities with a simplified notion of an application package that will in fact relieve the inherent dependency on the OSv operating system. Despite being simplified, the packages of MIKELANGELO Package Manager (MPM) are going to exhibit much more power and flexibility to end users by facilitating the management of dependencies and run-time configuration options. Besides the package management system itself, this deliverable also presents some of the commonly used packages that have already been prepared for MPM.

This report is concluded with the presentation of future plans. We are most interested in improving the flexibility, usability and ease of use from the perspective of both, the application packager and user. Significant effort will also be invested into provisioning of a lightweight interface facilitating integration of the MIKELANGELO package repository with high level management systems, such as OpenStack.

Acknowledgement

The work described in this document has been conducted within the Research & Innovation action MIKELANGELO (project no. 645402), started in January 2015, and co-funded by the European Commission under the Information and Communication Technologies (ICT) theme of the H2020 framework programme (H2020-ICT-07-2014: Advanced Cloud Infrastructures and Services)

© Copyright Beneficiaries of the MIKELANGELO Project

Table of contents

1 Introduction ... 6

2 Current State of Application Packaging for OSv ... 8

3 The MIKELANGELO Package Manager ... 10

3.1 Initial Requirements ... 10

3.2 Application Package Concepts ... 12

3.3 Application and Package Management Workflow ... 14

3.3.1 Simple Workflow ... 14

3.3.2 Advanced Workflow ... 15

4 Application Packages ... 20

4.1 Common Packages ... 20

4.1.1 Bootstrap ... 21

4.1.2 HTTP Server ... 21

4.1.3 Command Line Interface ... 22

4.2 OpenFOAM Application Package ... 22

4.2.1 Building an OpenFOAM Image ... 24

4.2.2 Composing OpenFOAM with MPM ... 25

4.3 Open MPI ... 26

5 Key Performance Indicators ... 27

5.1 KPI4.1 ... 27

5.2 KPI4.2 ... 28

6 Outline and Future Plans ... 30

6.1 Elimination of External Dependencies ... 30

6.2 Extension of Frontend Tools ... 30

6.3 Change of the Underlying Architecture ... 30

6.4 Support for Other Guest Operating Systems ... 31

6.5 Run-time Options ... 31

7 Concluding Remarks ... 33

8 References and Applicable Documents ... 34

© Copyright Beneficiaries of the MIKELANGELO Project

Table of Figures

Figure 1: Architecture of the MIKELANGELO Package Manager. 11

List of Abbreviations

The following is the list of abbreviations use throughout this document.

API	Application Programming Interface
CLI	Command Line Interface
HPC	High Performance Computing
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
MPI	Message Passing Interface
MPM	MIKELANGELO Package Manager
NFS	Network File System
PIE	Position Independent Executable
REST	Representational State Transfer
RPC	Remote Procedure Call
VM	Virtual Machine
VMI	Virtual Machine Image
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

© Copyright Beneficiaries of the MIKELANGELO Project

1 Introduction

An application package is a bundle of one or more components that work together to achieve one common goal. The package may contain an application binary and one or more libraries used by the former. It may also contain one or more configuration files used to alter the behaviour of the application or just some input data used to operate on when started. The application package may further be standalone containing all the necessary bits and pieces for proper operation or comprised of several other application packages.

Whatever the content of the package, the process of building it is of vital importance because the amount of work required to integrate the application package into a novel system, such as OSv, will determine whether the system will succeed or not. Technical superiority of novel systems alone does not guarantee a wider adoption that is mandatory for building, managing and financing a successful product.

ScyllaDB (previously Cloudius Systems) have built the cloud operating system OSv from ground up. OSv already provides most of the standard libraries, such as libc, libm, libpthread etc. However some design decisions limit seamless reuse of application packages from other general purpose systems. Adaptations of the applications are thus inevitable. These may range from recompiling for the target operating system (OSv) to significant patching of the application's source code. Some of the most commonly used applications in cloud environments are already compatible with OSv [1]. For example, memcached [2] caching system and Cassandra [3] database are not only supported in OSv but they also outperform Linux-based execution. These and others have been provided as mostly source-based packages that end users may use to build virtual machine images on their own using one of two existing image building approaches: developer scripts integrated into the OSv source code itself and Capstan, a tool inspired by Docker addressing some of the pain points of the build scripts. Some of these packages have been provided primarily for demonstration purposes, while others may be used in production settings possibly replacing existing applications running on Linux or other systems.

In the deliverable D2.16 The First OSv Guest Operating System MIKELANGELO Architecture [4] we have presented the initial design of the MIKELANGELO Package Manager (MPM) improving the application packaging system targeting OSv in the first phase. The current document builds on the ideas from the previous deliverable, presents the current state of MPM and lays additional plans for future versions. It also describes some of the commonly used pre-built ready to install packages that are a result of task T4.3 Application Packaging. Although still very basic packages, these early releases already outline the target workflow for the packaging envisioned by MIKELANGELO.

© Copyright Beneficiaries of the MIKELANGELO Project

This document is organised as follows. Section 2 deals with application packaging in general, and presents OSv's existing approaches. Section 3 then introduces the modifications of the existing Capstan tool that have been made within task T4.3. Section 4 presents some of the commonly used packages as well as two use-case related packages, OpenFOAM and Open MPI. Section 5 provides a comprehensive evaluation of the Key Performance Indicators as a self-assessment of the current state of the work carried out on the topic. Future plans and ideas are described in Section 6 focusing on the packaging itself as well as integration thereof within Cloud/HPC environment. Last section summarises and concludes the deliverable.

© Copyright Beneficiaries of the MIKELANGELO Project

2 Current State of Application Packaging for OSv

Deliverable D2.16 introduced two existing tools for packaging applications for OSv guest operating system, namely the build scripts and Capstan application. A summary of both tools is briefly presented in this section, along with their major limitations when it comes to broader use.

The major difference between OSv build scripts and Capstan is their intended audience. Build scripts are targeting active developers of the OSv kernel or module maintainers. They are also used for end users requiring the bleeding edge kernel built directly from the source code. Since OSv is changing rapidly, adding support for new functionalities, frequently asked for by end users porting their applications to be OSv compliant, this is a very efficient way of building virtual machine images. However, this technique requires a complete development environment to be setup by the end user which is neither always possible nor desired.

An additional limitation of this approach is the use of a manifest file specifying the structure of directories and files that are to be uploaded into the virtual machine image. The manifest maps local files into their corresponding locations in the image and must be manually crafted by the end user. On the other hand dependent modules must be provided separately in another Python file, named module.py using an API provided by the OSv scripts. Dependencies may be required in two ways:

- **require** will use the given module and copy its files, specified in manifest, onto target virtual machine image;
- **require_running** will also copy all the files of the required module, but will additionally configure this module for starting after the virtual machine is started.

The API also allows one to construct a manifest with Python commands by providing mapping through the provided functions.

In an attempt to alleviate and simplify the building process, Capstan was introduced, aiming for a more user-friendly approach to building virtual machine images based on existing, preconfigured images provided by ScyllaDB in their official repository of base images. Capstan took a similar approach to that of the popular tool for packaging containerised applications, Docker. The build manifest is based on a single file (Capstanfile) specifying the details of how the new virtual machine image should be built:

- the underlying **base image** that can be any of the images provided in a central repository (if the corresponding image is not available in the local repository it is first downloaded and imported for later use)
- the **build** command describes how the package is built from source if necessary. Since existing OSv applications started as packages for OSv build scripts, most of them are

© Copyright Beneficiaries of the MIKELANGELO Project

provided with a Makefile downloading, patching, compiling and packaging these applications. The same command is therefore used to build applications with Capstan.
- the **cmdline** defines the commands that are going to be launched when the virtual machine finishes booting up. The cmdline is stored in the virtual machine image itself
- the **rootfs** provides the content of the application that gets uploaded into virtual machine image. This command is optional, because Capstan looks for a subdirectory "ROOTFS" in the application root directory if it is not provided. Capstan will upload files to the image relatively to the rootfs directory and will preserve any directory structure.

Using Capstan the end user needs to prepare all the files in a special directory. All files from this directory are copied onto the virtual machine, along with all the files that have been provided by the base image. This relieves the end user from having to manually specifying files and their respective locations in the virtual machine image. One of the major disadvantages of Capstan is the fact that base images are always packaged with a fixed partition for user files, which is approximately 10 GB in size. There is no way to create an image of different size using Capstan. Furthermore, building target images consisting of a number of modules would require the user to manually construct the ROOTFS with the contents of all of these modules. This approach is typically complicated and error prone.

Section 3.1 of deliverable D2.16 [4] demonstrates the use of both approaches on a real world example supporting OpenFOAM application for the purposes of the Aerodynamics use case [5].

© Copyright Beneficiaries of the MIKELANGELO Project

3 The MIKELANGELO Package Manager

This section presents the first contribution of the MIKELANGELO project towards a unified and simplified application and package management solution. It is considered first and foremost a preliminary reference documentation of the package manager being developed.

The work started with a thorough analysis and evaluation of the Capstan source code which has revealed that the underlying infrastructure is solid and extensible in a way suitable for adding new capabilities. To this end, the MIKELANGELO Package Manager (MPM) builds upon Capstan to overcome the major limitations of the existing tool briefly presented in the previous section and in report D2.16.

3.1 Initial Requirements

In D2.16 we have presented major requirements for the MPM tool which are described in the following list:

REQ 24: OSv support for contextualisation of VM instances. The most important requirement for the application packaging and OSv itself is the ability to customise the content and behaviour of the application running within virtual machines, i.e. contextualisation of VMs. Contextualisation should be enabled during the image building process as well as during run-time. The former is of particular interest for application packaging, while the latter already is supported by means of an internal extensible HTTP server providing REST API accessing and controlling OSv and applications.

REQ 36: Support environment variables in Capstan. Environment variables are one simple form of contextualisation allowing users to control the behaviour of the application from the outside. This requirement is actually a run-time requirement. Capstan not only allows building of images but also running them in various runtime environments.

REQ 51: Capstan must allow publishing of images to OpenStack Image service. OpenStack has been chosen as the cloud management platform to be used as the basis for the MIKELANGELO stack. One of the identified requirements is to allow direct integration of application packaging with OpenStack services, in particular the image (Glance) and compute service (Nova). This would simplify the deployment of images and instantiation of virtual machines.

REQ 52: Capstan must allow using the OpenStack Image Service as base images to be extended. Capstan provides an internal repository of built images serving as

© Copyright Beneficiaries of the MIKELANGELO Project

the potential base images. The user can either use the official repository or use their own. This requirement would simplify the use of existing base images from users' OpenStack cloud environment augmenting them with additional packages and data.

These high level requirements have been provided by business use cases as well as cloud infrastructure integration. They serve as the basic guideline for the target solution as offered by the MIKELANGELO project. These requirements lead to the design of an architecture comprised of loosely coupled components. The architecture diagram is presented in Figure 1.

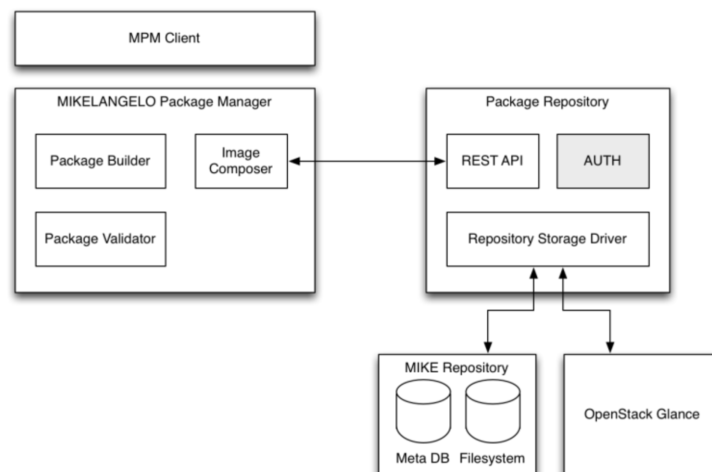

Figure 1: Architecture of the MIKELANGELO Package Manager.

The components of the architecture are briefly discussed next.

MPM Client is the user-facing tool providing access to all the functionalities of MPM through a rich command line interface.

Package Builder is responsible for management of the entire lifecycle of the packages.

Image Composer uses the existing packages and composes them into runnable OSv-based virtual machine images.

Package Validator validates the contents of the package or an application comprised of several packages using a set of pre-defined rules. These will range from simple verification of the metadata to application specific checks, such as missing libraries, binary format etc.

Package Repository stores the packages and virtual machine images and exposes them through the commands available in the REST API. The repository uses a storage driver to enable support for various data stores. MPM internal repository and Glance are the two storage implementations planned.

© Copyright Beneficiaries of the MIKELANGELO Project

In order to fulfil the requirements and implement the application management for OSv, we have initially focused on the concept of application packages and the corresponding workflows end users should abide to use their applications on top of HPC and Cloud infrastructure. Application concepts and the workflows are presented next.

3.2 Application Package Concepts

Discussing with the authors of OSv (ScyllaDB) as well as some of the interested stakeholders, the way package content is defined in existing systems is a rather challenging task for a newcomer to comprehend. Although the structure of the manifest file is very simple, preparation of such file is rather cumbersome and error-prone. Special attention must be paid to the definition of proper file mappings. In some existing OSv-based applications, the manifest has been simplified to specify a reference to a single directory containing the desired structure of files in the virtual machine image.

With MPM we decided go one step further and completely omitted the mandatory manifest file. In it's simplest form, an MPM package is simply **a directory containing all the required content of the package.** This directory is considered as the root of the filesystem of the virtual machine image that is uploaded through a new Capstan *compose* command, described in the section User's Manual below. Compressed files may also be used for more efficient distribution of a package. This package form assumes that the directory contains everything the application needs to be properly executed in OSv allowing experienced technicians to package an all-in-one package for their applications without having to rely on external packages.

On the other hand, this approach severely limits the reuse of common packages that must be packaged for every single application manually. To overcome this limitation, we introduced a package metadata file right within the package's directory. Metadata resides in package's subdirectory called "meta" and the only mandatory part of the package's metadata is actually a package descriptor in meta/package.yaml. The YAML file consists of the following properties

- the **name** of the package: mandatory property of every package;
- the **author** of the package: mandatory name and email of the author;
- optional **version** of the package: the version tag is currently not considered when building a package but helps resolving potential issues; it may be used in future versions;
- the list of **required** package names used when composing the virtual machine image: all required packages must be found and uploaded onto virtual machine image; in future we expect that this list will extended with optional version tags;

© Copyright Beneficiaries of the MIKELANGELO Project

- optional list of **provided binaries** defines executables that are exposed to the user of a package;
- optional list of **autorun** commands specifies binaries and their parameters that are executed immediately after the OSv virtual machine is started.

The YAML file format was chosen in favour of XML or JSON because of its simplicity and readability. YAML omits almost all structural elements making it ideal even for occasional users. Complete parser and validator for reading and writing MPM package metadata descriptors have been implemented providing enough feedback to the user trying to resolve common mistakes.

Section 4 presents examples of package descriptors for some of the commonly used packages as well as initial versions of packages built for MIKELANGELO use cases.

The following listing presents a simple MPM package descriptor for an imaginary package. It describes an artificial package named “MIKELANGELO Package” (name attribute) that was created by the “MIKELANGELO Consortium” (author attribute). Specification of the version, even though it not used at this moment by the MIKELANGELO package management is important for future releases.

```
name: MIKELANGELO Package
author: MIKELANGELO Consortium (info@mikelangelo-project.eu)
version: 0.1

require:
    -   httpserver
    -   nfs

binary:
    -   mike-app: /usr/bin/mike/app
    -   hello: /hello/usr/bin/hello
```

Following these basic data about the package, the package descriptor lists two mandatory requirements that should be uploaded to the virtual machine image during the composition process, i.e. *HTTP* and *NFS* servers. MPM will look for these packages in Capstan’s internal repository if possible, otherwise try to download them from one of the provided remote sources.

Finally, the package specifies two binaries that are exposed to end users, *mike-app* and *hello*. MPM uses this to create symbolic links in OSv in /usr/bin directory to simplify the execution of exposed applications. In future versions, MPM will further use the list of binaries to expose command line options facilitating the execution of commands by dynamically providing information about available binaries and their configuration options. Furthermore, using the list of binaries will facilitate the integration of OSv-based virtual machine images generated on the fly.

© Copyright Beneficiaries of the MIKELANGELO Project

In deliverable D2.16 we have also introduced *package hooks*, which may be used when a package is being built, tested or executed. However none of these hooks have been considered in the initial version and are thus not further elaborated in this section.

3.3 Application and Package Management Workflow

MPM differentiates between two application packaging approaches

- simple form where OSv images are built from directory or compressed file and
- advanced form allowing the composition of OSv images from a number of referenced packages.

These two approaches are presented in the following subsections. Both workflows have already been fully integrated into a modified Capstan tool that is going to be available as part of the public release of the MIKELANGELO stack.

3.3.1 Simple Workflow

This workflow is the simplest possible way to build an OSv-based virtual machine image. The only requirement is to have access to the bare OSv kernel (the *launcher* image) and a dedicated directory where all the other contents of the image resides.

The default launcher image is downloaded automatically by the Capstan tool from a remote repository on the first use. Capstan does not impose any requirements on the launcher image. It is simply a virtual machine image that serves as the base for other images that augment it. This means that it is perfectly fine to manually build the OSv kernel from source code and import the launcher image into the local Capstan repository for later use.

To compose the target virtual machine image one only has to package all files necessary in a single directory hierarchy. To illustrate, imagine a simple native application with the following structure:

```
.
├── lib
│   └── libhello.so
├── main.cc
└── Makefile
```

This project contains one C++ source file that depends on another shared library (libhello.so). Makefile compiles the main.cc and links it with the given library to create the main executable (main.so) in the root of the project’s directory. The final package structure is:

```
.
├── lib
│   └── libhello.so
├── main.cc
```


© Copyright Beneficiaries of the MIKELANGELO Project

```
├── main.so
└── Makefile
```

Capstan now provides a convenient method to compose a virtual machine for this project. The compose command requires two parameters

- **image name** as it will appear in Capstan's local repository. Although simple names are allowed, the suggested way is to use virtual grouping to image names as this will lead to a nicely organised repository. For example if the image is named mike/simple-image, Capstan will first create group's directory (mike) and put the image name (simple-image) into that directory.
- **path to image contents** is the root of the complete hierarchy of directories and files that are to be uploaded to virtual machine image. For the very simple cases, the path may also point to a single file.

Based on this, the above project can be used to compose a virtual machine image using the following command

```
$ capstan compose mike/hello-osv ~/mike/apps/hello-osv/
Importing mike/hello-osv...
Adding /Makefile...
Adding /lib...
Adding /lib/libhello.so...
Adding /main.cc...
Adding /main.o...
Adding /main.so...
```

This clones the default OSv launcher image and uploads all the files onto the image. All files are copied to the image relatively to the source directory:

- main.so to /main.so because it is in the root source directory
- libhello.so to /lib/libhello.so because it is in the lib subdirectory

To test the image, use the existing *run* command and specify the command to execute (using Capstan's -e command line option).

```
$ capstan run mike/hello-osv -e "/main.so MIKELANGELO"
Created instance: mike-hello-osv
OSv v0.24-16-g4235192
eth0: 192.168.122.15
Hello, MIKELANGELO
```

3.3.2 Advanced Workflow

Advanced workflow supports more complex real-world scenarios where applications and virtual machine images are comprised of several other packages. These can be commonly used packages provided by a remote repository (e.g. HTTP server, NFS server, Java) or internal packages (for example OpenFOAM base application as used by the corresponding use case).

© Copyright Beneficiaries of the MIKELANGELO Project

All phases of this workflow are presented next.

Management of remotes

The publicly available version of Capstan assumes that only a single remote repository is available. This repository is maintained by ScyllaDB and only contains a small number of base Capstan images. These are already pre-packaged with different subsets of modules and a user partition of fixed size. It is not possible to use this repository for storing other images and packages.

MPM extends management of remote repositories to allow more than one source to be used. This provides the necessary flexibility to end users creating their own public or private repositories of packages.

Initialisation of the package

As described in Section 3.2 the MPM package is a directory with a package descriptor. The descriptor is a file called package.yaml residing in a meta subdirectory of the package. To facilitate the initialisation of the package, MPM provides a convenient method that creates this package descriptor with the basic information, such as name, author and version, already stored in a suitable format. The user is allowed to modify the package descriptor and add other properties, such as other packages required for proper operation of the VM and the binaries and services the packages provides.

The package may be initialised either prior to creating the package contents or after the content has already been defined. This gives users the flexibility and ease of use as not to impose any unnecessary requirements to the packaging workflow.

The following command shows an example of a package initialisation

```
$ capstan package init -l "My first package" --author="MIKELANGELO Consortium" -v 0.1 my-package
```

This creates the package directory my-package and also the package descriptor in my-package/meta/package.yaml with the following content

```
name: My first package
author: MIKELANGELO Consortium
version: "0.1"
```

Alternatively, if the user omits the final argument (my-package in the example above), it will only create the package descriptor (meta/package.yaml) in the current directory.

© Copyright Beneficiaries of the MIKELANGELO Project

Creating package contents

The package content is any hierarchy of files located under the root of the package directory. The root of the package is simply the directory containing the package's meta directory. This initial version of MPM has no specific requirements for the organisation of files. Nevertheless, the best practice is to use well-known locations from the Unix/Linux world:

- Put all your binaries into the /usr/bin directory. If a user is explicitly listing binaries in the package descriptor, this rule may be disregarded as MPM will make sure that binaries are exposed in a common way.
- Put all your libraries into /usr/lib directory
- Configuration options should be placed inside /etc directory.

There are no recommendations on the application specific data as they may reside anywhere on the guest's filesystem. For example, in the case of the OpenFOAM application we have decided to put input files into /foam/case.

MPM provides a helpful command to inspect the content of the target package. If the following command is executed in the package's root directory, it will create a subdirectory *capstan-pkg* in the current directory with files from this and all required packages.

```
$ capstan package collect
```

To show an example of this command, a slightly modified example from the previous section will be used. It uses the same two binaries but adheres to the above best practices:

```
.
├── meta
│   └── package.yaml
└── usr
    ├── bin
    │   └── main.so
    └── lib
        └── libhello.so
```

This package requires the HTTP server which is specified in the package's descriptor:

```
name: My first package
author: MIKELANGELO Consortium
version: "0.1"

require:
 - httpserver
```

After using Capstan's "package collect" command, the hierarchy in capstan-pkg is the following. Lines starting with "#" are comments. Note that some of the files have intentionally been skipped to maintain a reasonable length.

© Copyright Beneficiaries of the MIKELANGELO Project

```
# Placeholder for the available virtual devices
├── dev
# System-wide configuration
├── etc
│   ├── fstab
│   ├── hosts
│   └── krb5.conf
# HTTP server provides a service which is started automatically on system boot.
├── init
│   └── 30-auto-00
├── libenviron.so
# This is the main HTTP server binary
├── libhttpserver.so
# These are some libraries that are part of the bootstrap package. More on this in
# the deliverable.
├── libtools.so
├── libuutil.so
├── libzfs.so
├── proc
├── tmp
# The following two tools provide ability to make ZFS partition and open file server
# facilitating file uploads.
├── tools
│   ├── cpiod.so
│   └── mkfs.so
├── usr
│   ├── bin
# main.so is the main binary of out application package.
│   │   └── main.so
# HTTP server provides many additional libraries.
│   ├── lib
│   │   ├── ...
│   │   ├── libcrypto.so.10
│   │   ├── libcrypto.so.1.0.1k
│   │   ├── libgcc_s.so.1
│   │   ├── libgssapi_krb5.so.2
# This is the library our main application depends upon.
│   │   ├── libhello.so
│   │   ├── ...
# HTTP server also provides other application specific data.
│   └── mgmt
│       ├── api
│       │   └── listings
│       │       ├── api.json
│       │       ├── app.json
│       ├── ...
```

Composing the package

The last step in this workflow is the actual composition of a virtual machine image. For this purpose Capstan provides another package related sub-command

```
$ capstan package compose mpm-test/my-app
```

that uses the contents and the package descriptor from the current directory, composes a complete virtual machine image and imports it into Capstan's local repository. It uploads exactly the same files as those collected by the "package collect" command.

© Copyright Beneficiaries of the MIKELANGELO Project

Running the Virtual Machine

Since the composed application package is self-sufficient virtual machine image stored within Capstan's local repository it may be started using existing Capstan run command, for example:

```
$ capstan run mpm-test/my-app
```


© Copyright Beneficiaries of the MIKELANGELO Project

4 Application Packages

This section describes the MPM packages currently available. These packages serve for demonstration purposes and are all publicly available at MIKELANGELO's Open Data repository [8].

4.1 Common Packages

The OSv source tree provides a number of modules that are commonly used in virtual machine images built by end users. These modules add functionality to the core OSv kernel and may be used for enabling additional requirements the application may have or for debugging of virtual machine images. Some of the commonly used modules are the following:

- **HTTP Server** offers a lightweight HTTP server. Primary reason for the HTTP server is to expose OS capabilities over a powerful RESTful API.
- **CLI** provides the command line interface allowing users to enter an interactive shell in the guest OS and inspect the content of the user ZFS partition. It furthermore allows direct access to the API, which can be used for debugging purposes.
- **Cloud init** supports initialisation of the virtual machine when the virtual machine image is started. If the cloud-init module is enabled in the virtual machine image, it will read configuration options from various options (sources) and apply them to the running VM before executing the command line provided to the machine.
- **Java** provides the Java runtime allowing execution of one or more java applications within an OSv instance. The biggest limitation of this module is that all java programs in one OSv instance share the same configuration of the virtual machine. A similar module is based on OpenJDK java.

While OSv build scripts already support integration of these modules into VMs, the current version of publicly available Capstan does not offer support for this. Instead, it provides pre-built virtual machine images that already have one or more of these modules integrated into virtual machine images. These images predefine the size of the image which cannot be modified which makes them impractical to use.

Based on the current requirements from MIKELANGELO's use cases, some of these modules have already been packaged for MPM. These modules have existed even before the MIKELANGELO project. The following packages thus present the meaning of each of these modules as well as present their package descriptors to familiarise the reader with the new concepts.

© Copyright Beneficiaries of the MIKELANGELO Project

4.1.1 Bootstrap

Bootstrap contains tools, libraries and configuration files that are required for proper operation of the virtual machine running OSv-based applications

- a placeholder for /dev: this directory does not contain any of the files because the system will create suitable device files as necessary
- system-wide configuration in /etc: it contains default hosts file for local hostname resolution and the static configuration of available filesystems (fstab). Since it is not yet possible to create arbitrary mount points, fstab will only list one user partition, mounted at root.
- bootstrap tools facilitate the partitioning (mkfs) and uploading of source files onto the VMI (cpiod)
- various libraries supporting the above tools and also common requirements of the kernel and user applications.

The package descriptor of the bootstrap package contains only the basic information.

```
name: OSv Bootstrap
author: MIKELANGELO (info@mikelangelo-project.eu)
version: 0.23-24-gc60331d
```

Other packages do not need to explicitly require the bootstrap package because it is included automatically in every composed VMI.

4.1.2 HTTP Server

The Hypertext Transfer Protocol (HTTP) server exposes a RESTful API of the OSv guest. It is available as a submodule of the OSv source tree and facilitates management and monitoring of the guest directly from the host or any other place, including a higher level cloud management. The HTTP server is furthermore important for debugging purposes because it allows execution of commands without rebuilding and restarting the virtual machine.

The package descriptor of the HTTP server package is presented next.

```
name: OSv HTTP REST Server
author: MIKELANGELO (info@mikelangelo-project.eu)
version: 0.23-24-gc60331d

autorun:
 - /libhttpserver.so
```

Although the HTTP server uses numerous other shared libraries (e.g. crypto, SSL, YAML), we have decided to provide it as a single package, mostly because current application requirements did not list these libraries at a finer level.

© Copyright Beneficiaries of the MIKELANGELO Project

The descriptor above also presents the first real example of a package that provides an application started immediately after the OSv kernel boots. Capstan currently does not provide an option to prevent services from starting automatically, but future versions will include additional package-wide options allowing users to build images with packages that can be toggled off when necessary instead of having to rebuild the entire virtual machine image.

4.1.3 Command Line Interface

The Command Line Interface (CLI) builds upon the HTTP server and provides a simple shell to the user of the virtual machine. All functions of the shell are provided by the HTTP server. Therefore, its package descriptor specifies httpserver as its only requirement.

```
name: OSv Command Line Interface
author: MIKELANGELO (info@mikelangelo-project.eu)
version: 0.23-24-gc60331d

require:
  - httpserver

binary:
  cli: /cli/cli.so
```

CLI also exposes a binary (cli) that points to the actual shared object (/cli/cli.so).

4.2 OpenFOAM Application Package

The purpose of this section is to present the steps required to make an open source application suitable for running in OSv. This section is based on OpenFOAM because it is the first application package provided specifically for one of the use cases of the MIKELANGELO project. Deliverable D2.10[5] presented OpenFOAM and some of the required changes necessary to the OSv kernel that have already taken place during the initial requirements analysis phase of the project.

In order to be able to run OpenFOAM based applications on top of OSv we had to make some minor changes to the build system. The following code listings present the common steps that a package maintainer needs to do when building a package from source code. Although the process is specific to OpenFOAM it presents one of possible ways to build an OSv compatible application.

```
#!/bin/sh

VERSION=2.4.0
BASEDIR=$PWD
ROOTFS=$BASEDIR/ROOTFS
SRCDIR=$BASEDIR/OpenFOAM-$VERSION
```


© Copyright Beneficiaries of the MIKELANGELO Project

```
# Check whether we need to download the tarball
if [ ! -f OpenFOAM-$VERSION.tgz ]; then
    wget http://downloads.sourceforge.net/foam/OpenFOAM-$VERSION.tgz
fi
# Extract the target version
tar zxf OpenFOAM-$VERSION.tgz
```

We start by setting some variables that will be used throughout the entire process. The script then checks whether an appropriate package exists and extracts it into a local directory.

```
# Set OpenFOAM's environment variables required to build the package. You should change
this to cshrc if
# that's the shell you are using.
export FOAM_INST_DIR=$BASEDIR
. $SRCDIR/etc/bashrc

# First, compile the wmake used to build OpenFOAM sources.
cd $SRCDIR/wmake/src
make
```

OpenFOAM uses wmake for compilation which we build in the code listing above. No changes are necessary here because wmake will be executed on the host machine.

```
# Patch the solidMixtureProperties library
cd $SRCDIR
patch -p1 < $BASEDIR/patches/solidMixtureProperties-dependency.patch
```

This is the first interesting part. One of OpenFOAM's core libraries does not specify dependencies properly resulting in a shared object that has no reference to the other library. With this patch, we have fixed this by adding the missing dependency making sure the build system uses it when compiling and linking shared objects.

```
# Make the OpenFOAM library
cd $SRCDIR/src
./Allwmake
```

This builds the core OpenFOAM library using wmake that leads us to the crucial part. The following patch first alters the build system by setting compiler and linker flags:

- compiler: -fpie
- linker: -pie -shared

Using these flags, the resulting application will be compiled and linked as Position Independent Executable (PIE) shared object which can be loaded and executed by the by OSv loader.

```
# Apply the patch changing WMake options to position independent executables.
cd $SRCDIR
patch -p1 < $BASEDIR/patches/pie.patch

# Go and build the simpleFoam
```

```
cd $SRCDIR/applications/solvers/incompressible/simpleFoam
wmake
```

The only remaining step is to collect all the necessary files and organise them according to our needs. The following example already adheres to the best practices presented in previous section.

```
cd $BASEDIR
mkdir -p $ROOTFS/usr/lib
mkdir -p $ROOTFS/usr/bin
mkdir -p $ROOTFS/openfoam

cp $SRCDIR/platforms/$WM_OPTIONS/bin/simpleFoam  $ROOTFS/usr/bin

ldd $SRCDIR/platforms/$WM_OPTIONS/bin/simpleFoam | grep -Po '(?<=> )/[^ ]+' | sort | uniq
| grep -Pv 'lib(c|gcc|dl|m|util|rt|pthread|stdc\+\+).so' | xargs -I {} install  {}
$ROOTFS/usr/lib
# Also install libfieldFunctionObjects.so as it is not linked from the simpleFoam
# (these libraries are loaded dynamically by the simpleFoam application)
install $SRCDIR/platforms/$WM_OPTIONS/lib/libfieldFunctionObjects.so $ROOTFS/usr/lib
install $SRCDIR/platforms/$WM_OPTIONS/lib/libforces.so $ROOTFS/usr/lib

# Copy the configuration files and scripts to the image.
cp -r $SRCDIR/etc $ROOTFS/openfoam
```

When using OSv's build scripts, we can also automate generation of the manifest file with the following command.

```
echo "
/usr/bin/simpleFoam: ${ROOTFS}/usr/bin/simpleFoam
/usr/lib/**: ${ROOTFS}/usr/lib/**
/openfoam/etc/**: $ROOTFS/openfoam/etc/**
" > usr.manifest
```

4.2.1 Building an OpenFOAM Image

The script from the previous section builds OpenFOAM and puts all files necessary into a single directory (ROOTFS), thus greatly simplifying the generation of virtual machine image. Indeed, in order to build a package using build scripts, one only needs to call the following commend to build the image.

```
$ $OSV_HOME/scripts/build image=OpenFOAM
```

Since this is part of the OSv kernel, this will also check whether the kernel and the loader image needs to be rebuilt first.

To build the image using Capstan, the following Capstanfile may be used

```
# Use the OSv base image with only OSv kernel and HTTP server.
base: cloudius/osv-base
```


© Copyright Beneficiaries of the MIKELANGELO Project

```
# This is the command line that will be executed once the VM is started.
cmdline: /usr/bin/simpleFoam -case /openfoam/case

# This command will build the package. Since it is built from sources, we can use make.
build: make

# Explicitly name the folder containing all the files the application needs.
rootfs: ROOTFS
```

followed by Capstan's build command.

```
$ capstan build OpenFOAM
```

This will put the contents of ROOTFS directory into the virtual machine image.

4.2.2 Composing OpenFOAM with MPM

In its simplest form, MPM does not require any package descriptor allowing one to put the contents of a given directory directly into the target virtual machine image:

```
$ capstan compose OpenFOAM /path/to/OpenFOAM/ROOTFS
```

However, all of the above ways of building OpenFOAM suffer from a common issue: if one needs to change the input to OpenFOAM simulation or only use OpenFOAM as a basis to run their own simulations, this would require rebuilding of the OpenFOAM base image.

This is where the benefit of application packages in MPM becomes clear. The package descriptor for OpenFOAM is rather minimalistic but it allows composition of more complex images on top of it.

```
name: OpenFOAM Base
author: MIKELANGELO Consortium (info@mikelangelo-project.eu)
version: 2.4.0
```

For example, one might compose an image using the following package descriptor

```
name: OpenFOAM Case 1
author: OpenFOAM User (of-user@example.com)

require:
 - httpseerver
 - openfoam

service:
 - /usr/bin/openfoam -case /openfoam/case
```

This will automatically compose a virtual machine image using the contents of the current directory as well as all the files of required packages. Specifying the service, it will automatically start the given command as soon as the OSv kernel finishes booting. Given the

© Copyright Beneficiaries of the MIKELANGELO Project

current directory contains OpenFOAM's input case in subdirectory openfoam/case, it will execute a simulation based on the initial conditions.

The HTTP server is not required by the OpenFOAM application. However, it is very useful for debugging purposes, either during the simulation execution or afterwards to collect the final results.

4.3 Open MPI

The previous section described the process of building the OpenFOAM [6] application package without support for parallel execution. OpenFOAM, as already described in deliverable D2.10 [5] is using an abstraction layer for the underlying framework and Open MPI [7] is currently the only production-ready implementation of this abstraction.

In cooperation between WP4 and WP5 we have already provided an initial support for parallel execution of multiple threads[1] within OSv. To this end, we have also provided a package for Open MPI which includes all the client-side libraries required. In order to use the Open MPI package, we can use the following package descriptor to support parallel execution:

```
name: OpenFOAM Parallel
author: MIKELANGELO Consortium (info@mikelangelo-project.eu)
version: 2.4.0

require:
 - openmpi
```

Extensive description of the changes that were necessary in the Open MPI itself is presented in report D5.4 [10].

[1] OSv, being a unikernel, does not support execution of multiple processes, but it fully supports multiple threads. However, all the threads are executed within a single space which means that all threads running the same application share global variables and environment variables. This has been addressed in T4.2 supporting multiple copies of these data to be available to different threads.

© Copyright Beneficiaries of the MIKELANGELO Project

5 Key Performance Indicators

From the inception of the MIKELANGELO project [9], one of the most significant promises has been to constantly monitor key performance indicators (KPIs) from all the different perspectives. KPIs focusing on the application packaging are related to the following objective of the project:

Objective O4: To provide application packaging, improving portability and deployment of virtualized applications.

The specific KPIs, as described in the project's Grant Agreement are:

- **KPI4.1**: A system for appropriately packaging of applications in MIKELANGELO OSv is provided.
- **KPI4.2**: A relative improvement of efficiency [time] between deploying a packaged and non- packaged application.

Assessment of both of these KPIs based on the current version of MPM is presented in the following subsections.

5.1 KPI4.1

This KPI deals with suitability of the packaging system and ease of use. As we have seen in previous sections, OSv provided two packaging systems (build scripts and Capstan) even before the MIKELANGELO started. However, both of them suffer from certain limitations that make them impractical for a wider adoption by the community.

We have based the MIKELANGELO Package Manager on top of Capstan because it has a solid codebase and fewer dependencies on the OSv source tree than build scripts. Capstan already provides the basic infrastructure necessary to build images and manage remote and local repository of virtual machine images.

The preliminary version of MPM, released as an integral part of this deliverable, already demonstrates some initial improvements of the baseline, in particular with respect to ease of packaging an application. MPM allows users to use their favourite tools to prepare the content of the package and use MPM only to compose the target virtual machine. MPM also provides supporting tools (package initialisation, collection of package contents etc.) simplifying the process even further but does not require the use of these tools.

Some of the limitations of the current version of MPM are the following

© Copyright Beneficiaries of the MIKELANGELO Project

- expressiveness of the package descriptor: the initial version provides only the basic package definition and dependency management (for example, even without some basic versioning)
- runtime management: users are already able to provide custom commands when booting the VM, however this requires writing the command line into the image's header actually altering the VMI within the repository
- volume management: it's currently impossible to use external volumes from within virtual machines. It is frequently necessary to explicitly separate applications from data.
- dependencies on external tools: MPM still requires QEMU for some rudimentary tasks, such as manipulation of image header (definition of partitions, boot command).

All of these limitations are going to be addressed in future releases as discussed in the next section.

5.2 KPI4.2

This release of MPM does not deal with relative improvement of the performance yet. Instead, it focuses on the design and implementation of a more flexible interface specifically targeting non-technical end users. Nevertheless, the following table shows average runtimes for some of the packages. The MPM data column also shows relative performance of MPM compared to build scripts and Capstan.

	Build scripts	Capstan	MPM
HTTP Server	3.15 s	3.44 s	6.93 s (2.2, 2.0)
CLI	3.30 s	3.59	6.84 s (2.1, 1.9)
OpenFOAM	30.93 s	7.97 s	8.44 s (0.3, 1,1)

The table shows that for simple packages (modules) build scripts outperform both Capstan and MPM. Build scripts are highly optimised to frequent recompilation of the kernel and modules making little redundant steps during the building of the image. These modules are also pre-packaged within Capstan base images (cloudius/osv-base and cloudius/osv) explaining the reason why Capstan builds images with base modules faster than MPM. The former has these modules already built into one of base images and thus only copies the base image into target image. On the other hand, the latter (MPM) needs to clone the launcher image and then upload all the required files to target image, including the bootstrap package.

Finally, building an image containing the OpenFOAM application is considerably slower using build scripts than Capstan or MPM. In this case, the OpenFOAM application uses a custom

© Copyright Beneficiaries of the MIKELANGELO Project

made Makefile that calls OpenFOAM's internal build system to recompile source code if necessary. Although no changes have been made to OpenFOAM sources, just checking the entire source tree takes nearly half a minute for build scripts. However, once OpenFOAM is compiled, there is no reason for checking whether recompilation of the source code is necessary. Capstan and MPM are thus using binaries and all other configuration files and simply upload them onto target images. Again, MPM performs slightly slower because it has to upload the bootstrap package which is not necessary in Capstan. We consider this a sensible trade-off between the raw performance and the flexibility and ease of use.

This KPI will be thoroughly evaluated within WP6 with a cooperation of all the use cases.

© Copyright Beneficiaries of the MIKELANGELO Project

6 Outline and Future Plans

Up until now, this document is focused on the current state of the MIKELANGELO Package Management which is currently available as a public preview version. In the following subsections we briefly touch improvements that are planned for the next phase of the project. These are tackling specific challenges in the MPM itself. Within other deliverables of Work Package 5 additional ideas are presented targeting the integration of MPM into chosen Cloud and HPC frameworks.

6.1 Elimination of External Dependencies

One of the most challenging limitations of the current MPM version is its inherent use of QEMU to perform some of the low level tasks, such as updating the image partition table and startup command. In order to use MPM (or Capstan), the end user must have QEMU tools installed on the host system. This is neither convenient for end users nor platform independent and will be addressed in the first official demonstrator in M18. In order to completely eliminate this requirement we are going to implement additional bootstrapping tools. These tools will serve as agents for front end tools and will be fully integrated into MPM through a well-defined API.

6.2 Extension of Frontend Tools

Frontend tools are exposed through a well-defined command line interface. The current version of this interface favours simplicity over the expressive power which is ideal for users getting started with the packaging of OSv applications. However, advanced users require more power and flexibility over the result of the packaging and composition processes. To this end, we are planning to extend the command line interface with new commands and optional parameters to these commands. An additional analysis and potentially also a survey will be conducted prior to the implementation of the new interface.

6.3 Change of the Underlying Architecture

MPM is based on Capstan that has a very basic architecture. It is comprised of a command line interface, a set of high level commands and several utility functions. The only user accessible part is the command line interface which resembles the interfaces of other popular solutions, like Docker and Juju. Every CLI command or subcommand is backed by a dedicated command in the backed logic facilitating the comprehension of the components. Finally, utility functions provide generic tasks that are shared across the commands.

Integration of such a system is cumbersome as it allows only the following two ways:

© Copyright Beneficiaries of the MIKELANGELO Project

1. Direct integration of code. That would require that the integrated system is also written in Go (programming language used in Capstan/MPM) or other languages that can directly use products produced by the Go compiler (for example C++).
2. Use of command line interface. CLI is well-formed and can easily be automated for simple tasks. However, such an approach is difficult to maintain as the interface changes because there is no inherent versioning support. It is also sometimes difficult to share erroneous states consequently requiring a lot of glue integrating the external tool into the main process.

Thus, due to the requirements of the HPC and, in particular, Cloud integration process, we are considering alternative options that will facilitate loosely coupled integration of MPM functionalities into a high level workflow. In recent time, the idea of microservices is used in nearly all similar solutions. Instead of having one monolithic tool, the underlying architecture is split into a multitude of smaller, manageable pieces exposing commonly acceptable interfaces, such as RESTful or RPC (Remote Procedure Call). Command line interface, which is the frontend for end users, is just one of the potential consumers of the commands provided as microservices. External systems and processes are free to use standard client-side libraries and tools guaranteeing proper management of the transport layer.

6.4 Support for Other Guest Operating Systems

Finally, we are also considering support for other operating systems. Although all these systems are already accompanied with at least one application packaging system, we believe that the notion of a self-contained and composable application may facilitate exploitation of an application management solution. There should be no technical reason that a package, created for OSv, for example OpenFOAM, would not work on Linux. In the context of HPC applications and users this will provide a tremendous boost to the potential reach of the MIKELANGELO project. By removing an inherent dependency on one specific guest operating system with some potentially severe limitations, we might be able to build an ecosystem for packages and applications that are suitable for inclusion in the arbitrary application market.

6.5 Run-time Options

One of the benefits of having application packages instead of complete virtual machine images is that they may be altered in various ways during the composition. However, applications typically provide several optional (command line) parameters allowing users to change the behaviour of the application. For example, an application may provide a verbose flag for debugging purposes and an MPI application specifies the number of processes that should work in parallel. Although additional VMI can easily be constructed using different configuration options, this would soon lead to an unmanageable repository of images.

© Copyright Beneficiaries of the MIKELANGELO Project

To solve this we are going to extend the package descriptor with available run-time options. This will allow application maintainers to explicitly expose some of the run-time options to end users. These options will be automatically injected into the command line interface and consequently presented in the command descriptor. For example, when the user will execute

```
$ capstan run help openfoam
```

they will get the list of available command line options, such as the number of processes and the location of the shared storage where the input data resides.

© Copyright Beneficiaries of the MIKELANGELO Project

7 Concluding Remarks

This deliverable describes the current state of the MIKELANGELO Package Manager (MPM) and application packages. The primary mission of MPM is to combine the best of existing build scripts, available to OSv developers, and Capstan, a virtual machine image builder. MPM is going to excel at flexibility and ease of use from the perspective of an end user interested in exploring the OSv guest operating system and building OSv-based applications and appliances.

Despite being an early release, MPM already demonstrates preliminary improvement of flexibility through few well-designed package management tools supporting the user in preparation of a package suitable for use within an independent virtual machine image. To showcase a proof of concept we have built several packages commonly used in OSv-based virtual machine images. We have used these packages to compose a package suitable for a slightly simplified execution of the aerodynamics use case. This package uses common packages and augments them with a ready to use OpenFOAM appliance that can be executed in any environment.

The MIKELANGELO Package Manager is going to be significantly refined in the future releases. We are going to address some of the major limitations as well as extend it with some completely new features that will support proliferation of the number of packages, applications and, consequently, end users.

© Copyright Beneficiaries of the MIKELANGELO Project

8 References and Applicable Documents

[1] OSv apps repository, https://github.com/cloudius-systems/osv-apps/

[2] OSv memcached application, https://github.com/cloudius-systems/osv-apps/tree/master/memcached

[3] OSv Cassandra application, https://github.com/cloudius-systems/osv-apps/tree/master/cassandra

[4] MIKELANGELO Report D2.16 The First OSv Guest Operating System MIKELANGELO Architecture, http://www.mikelangelo-project.eu/deliverables/deliverable-d2-16/

[5] MIKELANGELO Report D2.10 The First Aerodynamic Map Use Case Implementation Strategy, https://www.mikelangelo-project.eu/deliverables/deliverable-d2-10/

[6] OpenFOAM home page, http://www.openfoam.org

[7] Open MPI home page, http://www.open-mpi.org

[8] MIKELANGELO Application Packages, http://opendata.mikelangelo-project.eu/public.php?service=files&t=48c868ca95a115db9c9db629e890cebf

[9] MIKELANGELO Project, http://mikelangelo-project.eu

[10] MIKELANGELO Report D5.4 First report on the Integration of sKVM and OSv with HPC, https://www.mikelangelo-project.eu/deliverables/deliverable-d5-4/

© Copyright Beneficiaries of the MIKELANGELO Project