MIKELANGELO

D2.16

The First OSv Guest Operating System MIKELANGELO Architecture

Workpackage	2	Use Case & Architecture Analysis	
Author(s)	Nadav Har'El		Cloudius Systems
	Shiqing Fan		Huawei
	Gregor Berginc		XLAB
	Daniel Vladušič		XLAB
Reviewer	Nico Struckmann		HLRS
Reviewer	Holm Rauchfuss		Huawei
Dissemination Level	Public		

Date	Author	Comments	Version	Status
2015-09-08	Nadav Har'El	Initial version	V0.1	Draft
2015-09-21	Shiqing Fan	Additions of the vRDMA guest	V0.2	Draft
2015-09-18	Gregor Berginc, Daniel Vladušič	Additions of the application packaging	V0.3	Draft

2015-09-24	Nadav Har'El	Additions of the OSv and Seastar architecture, and the monitoring	V0.4	Draft
2015-09-26	Nadav Har'El	Ready for review	V0.5	In review
2015-09-29	Nadav Har'El, Shiqing Fan, Gregor Berginc, Daniel Vladušič	Updates, according to the review comments	V0.6	In review
2015-09-30	Gregor Berginc	Document ready for submission	V1.0	Final

Executive Summary

The MIKELANGELO project sets out to modify the traditional cloud stack in order to make it easier to run I/O-heavy and compute intensive HPC applications more efficiently and on different hardware platforms - Cloud and HPC environments.

This deliverable describes the architecture of the Guest Operating System - OSv, chosen to be used as part of the overall MIKELANGELO technology stack. We describe its architecture, combined with baseline evaluation (including benchmarks). These show improvement when using OSv as the guest operating system over more traditional operating systems.

Other foci of the document include the additions to the operating system - means to efficiently communicate between different virtual machines through RDMA mechanism, the SeaStar API that significantly improves the speed-ups through novel programming paradigm at the cost of re-programming of the original application and finally, the application packaging system design for OSv, supported by the overview of similar packaging systems.

The deliverable produces the architecture of OSv, the measurements (baseline and speed-ups), designs of the supporting systems and, as this is the first iteration of the deliverable, plans for future work.

Table of contents

Table of Figures

1 Introduction

The MIKELANGELO project builds a new cloud stack with the intention of making it easier to run I/O-heavy and compute intensive High-Performance-Computing (HPC) applications in the cloud, and additionally running these applications more efficiently. The overall architecture of the MIKELANGELO cloud stack includes the following components:

1. The cloud management software, which we described in deliverable D2.19 [1].
2. The hypervisor sKVM, which makes it possible to run multiple virtual machines (VMs, also known as "guests") on one physical machine ("host"). We described the architecture of sKVM in deliverable D2.13 [2].
3. The operating system running on each virtual machine. Its architecture is the topic of this deliverable.
4. The actual application to run on the VMs. We discussed several example applications in the use-case deliverables D2.1 [3], D2.4 [4], D2.7 [5], and D2.10 [6], and in Section 7 we'll also consider additional simple benchmark applications.

In this document, we describe the first architecture of the **guest operating system** layer of the overall MIKELANGELO cloud stack. The term "guest operating system" (or "guest OS") is used for the operating system which runs on each individual VM, and implements the various APIs (Application Programming Interfaces) and ABIs (Application Binary Interfaces) which the various applications need.

Today, most cloud and HPC applications are written to run on Linux, the same OS that was used earlier when running on physical machines. Some of the features that once made Linux desirable on physical machines, such as a convenient single-machine remote administration interface (ssh, config files, etc.) and support for a large selection of hardware, now became irrelevant or even a burden increasing Linux's size, complexity, and boot time. Some of the traditional roles of the OS have become redundant on the cloud, and are now pure overhead: The most prominent example is Linux's support for running multiple processes isolated from one another, and all of them isolated from the kernel. However, the cloud additionally offers isolation between the different VMs, so increasingly, users are deploying separate applications on separate VMs instead of separate processes on the same VM. This makes isolation *inside* a VM redundant, and a performance burden because it slows down context switches, system calls, and other parts of the kernel.

Thus MIKELANGELO replaces the Linux kernel and system libraries by OSv, a new operating system designed especially for running efficiently on virtual machines, and capable of running existing Linux applications with certain limitations (namely, that the application is multi-threaded but not multi process, and that it is compiled as a relocatable executable). Compared to Linux, OSv has a smaller disk footprint, smaller memory footprint, faster boot time (sub-second), fewer run-time overheads, faster networking, and simpler configuration management. In **Section 2**, we describe in detail the architecture of OSv.

In addition to efficiency, a second goal of the MIKELANGELO project is to simplify deployment of applications in the cloud. In **Section 3**, we outline MIKELANGELO's application packaging and image composition mechanisms provided by the MIKELANGELO Package Manager (MPM). It will allow users to quickly and conveniently create new application packages from scratch or composing several application packages (libraries, components) into a target application. The section focuses on a definition of the package structure, metadata and tools provided by MPM with the main goal of building real virtual images suitable for execution in various cloud environments.

While OSv allows running existing Linux applications, in **Section 4** we note that certain Linux APIs, including the socket API, and certain programming habits, make applications which use them inherently inefficient on modern hardware. OSv can (and does) improve the performance of such applications to some degree, but rewriting the application to use new non-Linux APIs can bring even better performance. So in Section 4 we describe a new API, called "Seastar", for writing new highly-efficient asynchronous network applications. We will demonstrate cases where although OSv brings some performance improvements, rewriting the application to use the Seastar API will bring significantly larger improvements.

In **Section 5**, we present the guest-side architecture of the virtualized Remote Direct Memory Access (vRDMA) mechanism. We first described this mechanism in deliverable D2.13 [2], and it requires cooperating modification to both the hypervisor and the guest operating system; In this document we will naturally focus on the guest-side modifications, referring to D2.13 [2] for some host-side details. The basic idea of vRDMA is to speed up communication between different guests by transparently replacing slow IP-based communications with more efficient mechanisms: RDMA (Remote Direct Memory Access) when the guests are on different hosts, or shared memory when the guests are co-located on the same host. Additionally, vRDMA provides efficient access to RDMA primitives directly for applications which can make use of them - including HPC applications which use MPI 2 one-sided communication, or MPI 3 remote memory access (RMA) primitives.

In **Section 6**, we present OSv's built-in monitoring capabilities: OSv provides a REST API (i.e., HTTP) server which allows convenient remote querying of a running guest for various statistics, including traditional Unix-like statistics (such as the list of threads, the VM's uptime, the amount of runtime used by each thread, etc.) and additional OSv-specific low-level event counters such as the frequency of mutex lock attempts, context switches, memory allocations, and many more.

In **Section 7**, we plan the evaluation of guest operating system, and present benchmarks which will help us compare its performance to that of the baseline guest operating system, Linux.

In **Section 8**, we provide conclusions, and **Section 9** we list the external references cited in this document.

2 OSv

MIKELANGELO runs an application on many virtual machines (VMs), also known as *guests* of the hypervisor. VMs on the cloud traditionally run the same operating systems that were used on physical machines, such as Linux, Windows, or *BSD. But the features that made these operating systems desirable on physical machines, are losing their relevance: Examples of such irrelevant features include a familiar single-machine administration interface, the support of multi-user and multiple applications, and the support for a large selection of hardware. At the same time, different features are important for MIKELANGELO: The VM's operating system needs to be fast, small, and easy to administer at large scale.

OSv is a new operating system designed specifically for running a single application on a 64-bit x86 VM. OSv is limited to a single application because the hypervisor already supports isolation between VMs, so an additional layer of isolation inside a VM is redundant and hurts performance. As a result, OSv does not support `fork()` (i.e., processes with separate address spaces) but does fully support multi-threaded applications and multi-core VMs.

The design of OSv, and that of Seastar described later in Section 4, stems from two main goals:

1. **Run existing applications, faster.**
   The goal here is to take unmodified (or only slightly modified) Linux executables, and have them start faster and run faster on OSv than they did on Linux. Naturally, improvement in performance is to be expected only for applications which make heavy use of the operating system, e.g. I/O-intensive applications. Compute intensive applications which do little I/O cannot be accelerated by a better operating system.
   OSv is also smaller than Linux - less code, smaller disk images, and lower memory use.
2. **Provide new APIs for writing even faster applications.**
   Today's Linux APIs - POSIX system calls, socket API, etc. - were formed by decades of Unix legacy, and some aspects of them are inherently inefficient. OSv can improve the performance of applications which use these APIs, but not dramatically. So our second goal is to propose new APIs which will offer applications, that are rewritten to use them, dramatically better performance than unmodified Linux applications. For this, our operating system offers a new library which we call Seastar and is described in section 4 below. In the "Cloud Bursting" use-case we rewrite the Cassandra application using Seastar, and will show how performance will dramatically improve over the unmodified Cassandra. Further, Seastar offers general-purpose APIs

which will be useful to many kinds of asynchronous server applications.

In the rest of this section we will provide a high-level description of the architecture of OSv's core - its kernel and its Linux compatibility. A high level overview of this architecture is presented in Figure 1. Other subsequent sections are dedicated to additional components of the guest operating system: package management (Section 3), the non-Linux API "Seastar" (Section 4), vRDMA (Section 5), and monitoring (Section 6).

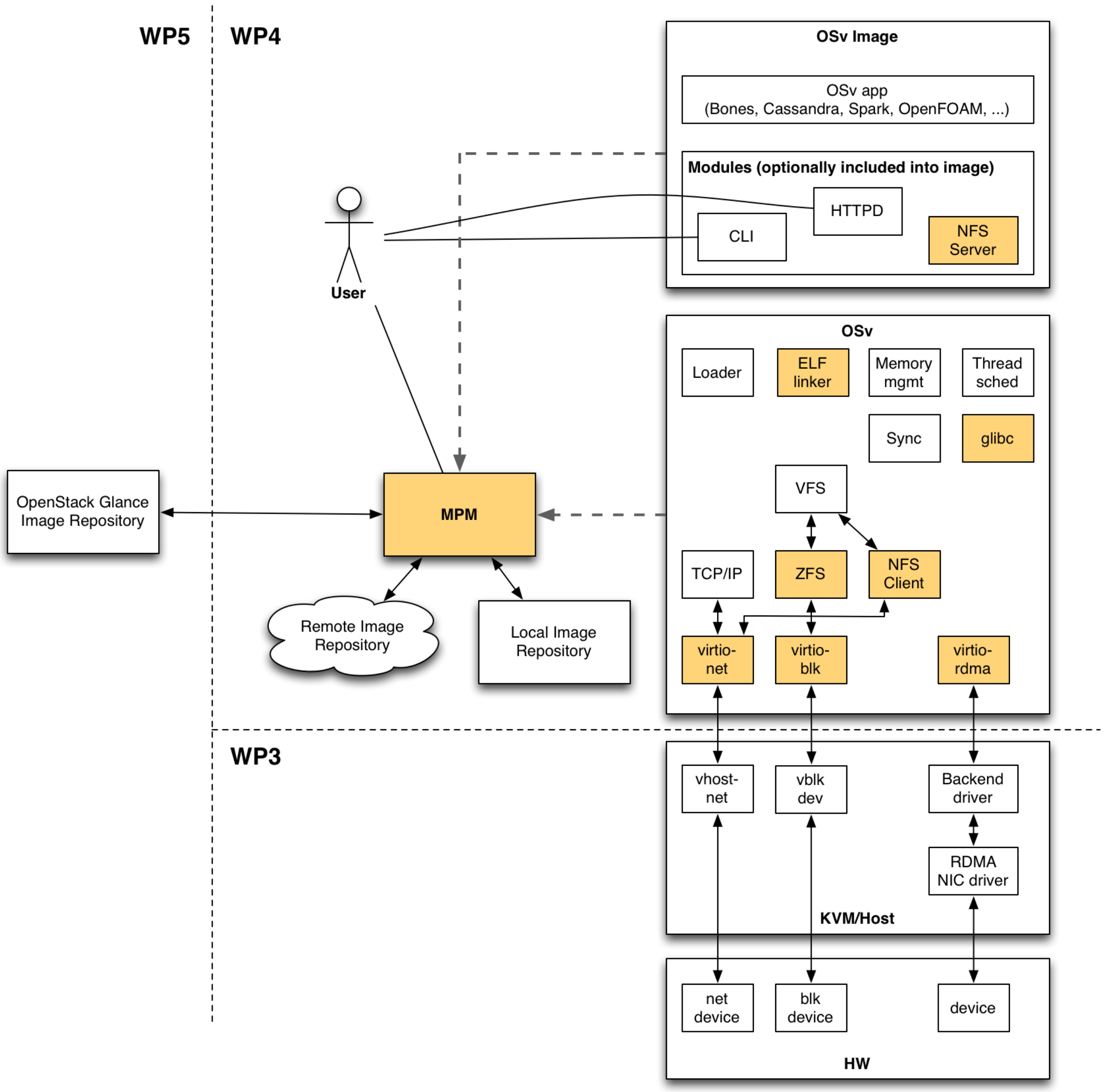

Figure 1: High level architecture of the OSv kernel and its relations to other components of this work package. Orange components are those that will be implemented or improved within MIKELANGELO.

2.1 The Loader

Like all x86 operating systems, OSv's bootstrap code starts with a real-mode boot-loader running on one CPU which loads the rest of the OSv kernel into memory (a compressed kernel is uncompressed prior to loading it). The loader then sets up all the CPUs (OSv fully supports multi-core VMs) and all the OS facilities, and ends by running the actual application, as determined by a "command line" stored in the disk image.

2.2 Virtual Hardware Drivers

General-purpose operating systems such as Linux need to support thousands of different hardware devices, and thus have millions of lines of driver code. But OSv only needs to implement drivers for the small number of (virtual) hardware presented by the sKVM hypervisor used in MIKELANGELO. This includes a minimal set of traditional PC hardware (PCI, IDE, APIC, serial port, keyboard, VGA, HPET), and paravirtual drivers: kvmclock (a paravirtual high-resolution clock much more efficient than HPET), virtio-net (for network) and virtio-blk (for disk).

As part of the vRDMA plan (see Section 5), we plan to add more drivers to OSv to support infiniband and RDMA, such as virtio-rdma and shared memory module.

2.3 Filesystem

OSv's filesystem design is based on the traditional Unix "VFS" (virtual file system). VFS is an abstraction layer, first introduced by Sun Microsystems in 1985, on top of a more concrete file system. The purpose of VFS is to allow client applications to access different types of concrete file systems (e.g., ZFS and NFS) in a uniform way.

OSv currently has four concrete filesystem implementations: devfs (implements the "/dev" hierarchy for compatibility with Linux), procfs (similarly, for "/proc"), ramfs (a simple RAM disk), and most importantly - the ZFS filesystem.

ZFS is a sophisticated filesystem and volume manager implementation, originating in Solaris. We use it to implement a persistent filesystem on top of the block device or devices given to us (via virtio-blk) by the host.

During this project, we also plan to add the NFS filesystem implementation (i.e., an NFS client), to allow applications to mount remote NFS shared storage, which is a common need for HPC applications.

2.4 The ELF Dynamic Linker

OSv executes unmodified Linux executables. Currently we only support relocatable dynamically-linked executables, so an executable for OSv must be compiled as a shared object ("`.so`") or as a position-independent executable (PIE). Re-compiling an application as a shared-object or PIE is usually as straightforward as adding the appropriate compilation parameters (-fpic and -pic, or -fpie and -pie respectively) to the application's Makefile. Existing shared libraries can be used without re-compilation or modification.

The dynamic linker maps the executable and its dependent shared libraries to memory (OSv has demand paging), and does the appropriate relocations and symbol resolutions necessary to make the code runnable - e.g., functions used by the executable but not defined there are resolved from OSv's code, or from one of the other shared libraries loaded by the executable. ELF TLS (thread-local storage, e.g, gcc's "__thread" or C++11's thread_local) is also fully supported.

The ELF dynamic linker is what makes OSv into a library OS: There are no "system calls" or system-call-specific overheads: When the application calls read(), the dynamic linker resolves this call to a call to the read() implementation inside the kernel, and it's just a function call. The entire application runs in one address space, and in the kernel privilege level (ring 0).

2.5 C Library

To run Linux executables, we needed to implement in OSv all the traditional Linux system calls and glibc library calls, in a way that is 100% ABI-compatible (i.e., binary compatibility) with glibc. We implemented many of the C library functions ourselves, and imported some of the others - such as the math functions and stdio functions - from the musl-libc project [29] - a BSD-licensed libc implementation.

Strict binary compatibility with glibc for each of these functions is a essential, because we want to run unmodified shared libraries and executables compiled for Linux.

2.6 Memory Management

OSv maintains a single address space for the kernel and all application threads. It supports both malloc() and mmap() memory allocations. For efficiency, malloc() allocations are always backed by huge pages (2 MB pages), while mmap() allocations are also backed by huge pages if large enough.

Disk-based mmap() supports demand paging as well as page eviction - these are assumed by most applications using mmap() for disk I/O. This use for mmap() is popular, for example, in Java applications due to Java's heap limitations, and also due to mmap()'s performance superiority over techniques using explicit read()/write() system calls because of the fewer system calls and zero copy.

Despite their advantages, memory-mapped files are not the most efficient way to asynchronously access disk; In particular, cache miss (needing to read a page from disk, or needing to write when memory is low) always blocks a thread, so it requires multiple application threads to context-switch. In Section 4, we explain that for this reason Seastar applications use AIO (asynchronous I/O), not mmap().

2.7 Thread Scheduler

OSv does not support processes, but does have complete support for SMP (multi-core) VMs, and for threads, as almost all modern applications use them.

Our thread scheduler multiplexes N threads on top of M CPUs (N may be much higher than M), and guarantees fairness (competing threads get equal share the CPU) and load balancing (threads are moved between cores to improve global fairness). Thread priorities, real-time threads, and other user-visible features of the Linux scheduler are also supported, but internally the implementation of the scheduler is very different from that of Linux. A longer description of the design and implementation of OSv’s scheduler can be found in [7].

One of the consequences of our simpler and more efficient scheduler implementation is that in OSv, context switches are significantly faster than in Linux: Between 3 to 10 times faster.

2.8 Synchronization Mechanisms

OSv does not use spin-locks, which are a staple building block of other SMP operating systems. The is because spin-locks cause the so-called “lock-holder preemption” problem when used on virtual machines: If one virtual CPU is holding a spin-lock and then momentarily pauses (because of an exit to the host, or the host switching to run a different process), other virtual CPUs that need the same spin-lock will start spinning instead of doing useful work. The “lock-holder preemption” problem is especially problematic in clouds which over-commit CPUs (give a host’s guests more virtual CPUs than there are physical CPUs), but occurs even when there is no over-commitment, if exits to the host are common.

Instead of spin locks, OSv has a unique implementation of a lock-free mutex, as well as an extensive collection of lock-free data structures and an implementation of the RCU (“read-copy-update”) synchronization mechanism.

2.9 Network Stack

OSv has a full-featured TCP/IP network stack on top of the network driver like virtio-net which handles raw Ethernet packets.

The TCP/IP code in OSv was originally imported from FreeBSD, but has since undergone a major overhaul to use Van Jacobson's "network channels" design [13] which reduces the number of locks, lock operations and cache-line bounces on SMP VMs compared to Linux’s more traditional network stack design. These locks and cache-line bounces are very expensive (compared to ordinary computation) on modern SMP machines, so we expect (and indeed measured) significant improvements to network throughput thanks to the redesigned network stack.

We currently only implemented the netchannels idea for TCP, but similar techniques could also be used for UDP, if the need arises.

The basic idea behind netchannels is fairly simple to explain:

- In a traditional network stack, we commonly have two CPUs involved in reading packets: We have one CPU running the interrupt or "soft interrupt" (a.k.a. "bottom half") code, which receives raw Ethernet packets, processes them and copies the data into the socket's data buffer. We then have a second CPU which runs the application's read() on the socket, and now needs to copy that socket data buffer. The fact that two different CPUs need to read and write to the same buffer mean slow cache line bounces and locks, which are slow even if there is no contention (and very slow if there is).
- In a netchannels stack (like OSv's), the interrupt time processing does *not* access the full packet data. It only parses the header of the packet to determine which connection it belongs to, and then queues the incoming packets into a per-connection "network channel", or queue of packets, Only when the application calls read() (or poll(), etc.) on the socket is the TCP processing finally done on the packets queued on the network channel. When the read() thread does this, there are no cache bounces (the interrupt-handling CPU has *not* read the packet's data!), and no need for locking. We still need some locks (e.g., to protect multiple concurrent read()s, which are allowed in the socket API), but fewer than in the traditional network stack design.

2.10 DHCP Client

OSv contains a built-in DHCP client, so it can find its IP address without being configured manually.

3 MIKELANGELO Package Management

Support for running existing applications is one of the fundamental topics that all new systems and frameworks must pay special attention to. Having a great system only demonstrated with a series of laboratory use cases rarely satisfies the needs of the ever increasing demand of the market. This phenomenon has been observed on many occasions in the past, most recently in the mobile platform market (for example, Microsoft struggled hard to increase sales due to poor application ecosystem and BlackBerry lost almost its entire market share because it failed to adjust to changing needs). Platforms backed with powerful application ecosystems have evolved and radically changed the perception of end users bringing once dominating providers to their knees.

OSv supports most of the standard libraries, however due to design decisions of OSv, some limitations have been imposed. This results in required adaptations of the applications, which may range from recompilation for the target operating system OSv to major patching of the application’s source code. Some of the most commonly used applications in cloud environments are already compatible with OSv. For example, memcached [8] caching system and Cassandra [14] database are not only supported in OSv but they also outperform Linux-based execution. Given OSv is an operating system on its own, applications already packaged for other systems, typically do not directly work in it. As already said, they may require (usually minor) patches to source code and their build process. To facilitate the uptake of OSv, a number of commonly used applications [15] have been provided by Cloudius Systems and the community presenting different ways to overcome limitations. Although some of these applications serve primarily for demonstration purposes, the others are already suitable for production use, replacing existing deployments.

To further improve the usefulness and uptake of OSv as the Cloud Operating System, MIKELANGELO project needs to improve the application packaging system for OSv and increase the number of available applications. Having readily available applications will significantly improve an essential part of the MIKELANGELO technology stack.

This part of the document is organised as follows: first, we present the existing ways of application packaging in OSv. Then, we briefly describe the existing solutions on other operating systems and finally, present the initial version of the new OSv packaging system, called MIKELANGELO Package Manager (MPM for short), which uses the currently available requirements (stemming from the MIKELANGELO use-cases) and the patterns from the widely used and proven packaging systems to deliver a robust solution for OSv and MIKELANGELO project.

A note on terminology: in the context of OSv, the application packaging denotes the process of building a package, consisting of application and its context. However, given the OSv package management approaches were built organically, the deployment onto the virtual image is almost always part of the packaging. In our approach, we make the distinction between application packaging and application installation into OSv virtual image.

3.1 Existing OSv Application Packaging

3.1.1 Developer Scripts

OSv source distribution comes with a shell script [16] for building images. The script, called scripts/build, builds the OSv kernel if it hasn't been built before and puts it into a complete image ready for running as a virtual machine in, for example, KVM (using QEMU). The script may take an additional parameter of the form "image=<ModuleName>", which is used to augment the raw OSv kernel image with the module named <ModuleName>. Multiple modules may be requested by separating them with commas, for example "image=<ModuleName1>,<ModuleName2>".

The script only works with local repositories (directories) of application modules which are specified in the config.json file of the OSv source tree, for example:

```
    "repositories": [
        "${OSV_BASE}/apps",
        "${OSV_BASE}/modules",
        "/home/mike/mike-apps"
    ]
```

The above section uses two default locations for applications and modules as well as a third repository of MIKELANGELO related applications which is used internally by the project before applications are published. This allows very coarse and high-level separation of application packages based on their origin.

To keep the repository small, application modules do not contain the actual application code. Instead they are described with a script that acquires the package or code from the Internet and builds a package into a form suitable for inclusion into an OSv based image. The recipe may include external files like static patches that need to be applied to the downloaded source code and provide OSv compliance. An example of a recipe for the OpenFOAM application can be seen in the following listing.

```
#!/bin/sh

VERSION=2.3.1
BASEDIR=$PWD
ROOTFS=$BASEDIR/ROOTFS
SRCDIR=$BASEDIR/OpenFOAM-$VERSION
```

```
# 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

## Patch bashrc to request debug compilation.
#cd $SRCDIR
#patch -p1 < $BASEDIR/patches/debug.patch

# 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

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

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

# 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

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
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

echo "
```

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

Last step in the above listing shows the second fundamental part of the application package, namely the user manifest. This is a simple textual file describing how the application files should be copied into the virtual image. Each line of this file consists of the destination inside the image and the location on host computer building the image. A special marker (**) can be used denoting the folder and its content recursively. Paths are separated with a colon (:).

The user manifest may be static or be generated using the script as in the case above.

The final important part of the package is the definition of the command that should get executed whenever the virtual image is started. This is a Python script that uses a common API provided by OSv describing the module. For example, the following instructs the OpenFOAM virtual image to start the execution of a simpleFoam application passing one additional parameter.

```
from osv.modules import api

default = api.run("--env=WM_PROJECT_DIR=/openfoam /usr/bin/simpleFoam -case
/openfoam/case")
```

When appending more than one package to a virtual image, command lines of individual modules are merged into a single one resulting in all being executed.

The build script is accompanied with an additional run script supporting many of the common options of the QEMU tool. The script is regularly updated with new requirements for the development of the OSv and is ideal for debugging and benchmarking of applications.

A brief summary and evaluation of this method:

- A BASH script (build) executes other scripts (BASH, Python), responsible for downloading, compiling, patching code.
- Each of the user-generated scripts needs to provide user manifest, specifying the files and their locations in the virtual image.
- Possible contextualisation using commands to be executed when virtual image is run.
- Very useful for hacking and testing.
- Very limited robustness (no validation of packages, no dependency management, resolution, etc.).
- Lack of formalized structure and processes to be followed in building OSv complete virtual image.
- Inappropriate for management of larger systems, production environments.

- Application patching and compilation intertwined with application installation into OSv virtual image.

3.1.2 Capstan

Capstan [17] was an attempt to provide an improved application building process. It was inspired by the well-known Docker [18] tool facilitating packaging of applications running in Linux containers. Although Capstan introduced other functionalities described later in this section, its main purpose is to simplify and standardize the structure of packages.

The structure is specified within a file named Capstanfile [19] allowing the following settings:

- **base**: name of the base OSv image used to augment with this package. It can either be a raw OSv kernel image (e.g. osv-base) or an image with some specific applications (for example cloudius/osv-openjdk for Java applications).
- **build**: denotes the command to be executed when building an image. In most existing applications this results in calling the make command.
- **cmdline**: the command to be executed when the image is started.
- **files**: a map of files that are added to the base image. Each file in this list specifies the location in the target image and the location on the host machine.
- **rootfs**: alternatively to the files option, Capstanfile can specify a path to the root directory on the host computer that is copied into the virtual image. Content of this directory is copied in its entirety to the root directory of the image.

If Capstanfile specifies the files option, it is used in place of rootfs. However, if neither files nor rootfs are specified, Capstan looks for a folder named ROOTFS in the current directory and adds its content to the image. This default is the recommended best-practice, and most applications in the osv-apps collection [15] use it.

An example of a Capstanfile for the OpenFOAM application is presented in the following listing.

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

# 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
```

Besides support for building of virtual images, Capstan also supports running them in various local hypervisors (KVM/QEMU, VirtualBox and VMWare) as well as in the cloud (Google

Compute Engine). However, Capstan has only a very limited support for configuration of the target VM (for example, number of vCPUs, size of block device and RAM).

Furthermore, Capstan also provides a very basic support for using RPM packages in OSv. If rpm-base option is used in Capstanfile, Capstan tries to download the RPM from a fixed base URL. Downloaded file is unpackaged and uploaded into the virtual image without any additional pre-processing such as applying patches or compiling the package. This is thus very limited in sense that most applications are not suitable for execution in OSv out of the box.

One of the major drawbacks of the Capstan is that the image built by Capstan builds a ZFS partition of an inconveniently-fixed size (10 GB) and also deploys modules that are not always required for applications (e.g. HTTP server and shell). The base image thus dictates the size and initial content of the OSv image built based on this image. In contrast, the build script allows users to customise the size of the image by deploying the kernel and two additional tools to create new partitions and upload files to OSv image.

A brief summary and evaluation of this method:

- Formalizes the structure of the packaging process - improves over a BASH build script.
- Improves the management of the user manifest, using files and rootfs concepts.
- Possible contextualisation using commands to be executed when virtual image is run.
- Can be used for building or simply combining and copying of applications into a complete package.
- Very limited robustness (no validation of packages, no dependency management, resolution, etc.).
- Inappropriate for management of larger systems, production environments.
- Always creates the ZFS partition of fixed size (10Gb) and deploys additional tools in the virtual image, which may not be needed at all.
- Application patching and compilation intertwined with application installation into OSv virtual image.

3.2 Other Commonly Used Packaging Solutions

This section briefly introduces some of the package management systems most commonly used in various scenarios and contexts. From a high level perspective all these solutions are very similar in their main principle. They all provide metadata describing the package and its dependencies as well as a recipe for installing or building the content of the package into a final form suitable for use on target system.

Systems below are used in variants of the UNIX operating system. Most of them are proven in real-world deployments and can be thus used as the stripped-down basis of the MPM. Some of them are novel (Ubuntu Snappy) - we believe these must be considered and used in the context of MIKELANGELO - requirements and targeted results.

3.2.1 RPM Package Manager

RPM packaging system has been introduced by Red Hat. It has become the de facto standard in the Linux community, used by almost all commercial Linux distributions (for example Red Hat or SUSE). It has evolved in a standard requiring LSB-compliant systems to support installation of RPM packages. RPM provides many features that make it a popular system for distribution and management. Although not all of them are mandatory for packaging OSv based applications, we describe some of these features in the paragraphs below.

RPM package management, from the perspective of the end user, provides a set of tools facilitating installation of packages in an non-interactive way. Contents (e.g., files and settings) of the installed applications are stored and consistently tracked throughout the entire lifecycle of the package. This greatly simplifies upgrades and uninstallation. RPM also provides tools for validation of installed packages. These tools help diagnosing potential issues in installed systems (for example, part of a package may be compromised due to disk failure or an attacker). Tracked files may also be searched to determine use of files by packages.

One of the most powerful features of RPM is its dependency management. Packages specify a list of prerequisites that must be resolved prior to installing the package itself. The dependency list (eventually, a graph) is extensively used when uninstalling packages guaranteeing that a package is not uninstalled when there exist packages depending on it.

All these features make RPM a tool of choice by system administrators significantly simplifying the workflow and regular processes of maintaining software on Linux systems. However, for the purposes of this document and OSv itself, the more important aspect of RPM system is a set of tools that support downloading of source code, patching the code to meet author’s intentions, automatic code configuration and finally compile the code into executable format. This entire workflow furthermore simplifies management of packages specific for the target environment. For example, an HPC provider may store a number of commonly used packages, all compiled with compiler flags specific for the hardware. RPM describes the recipe for patching the source code and configuring the compiler.

3.2.2 Debian Package Manager

On another part of the Linux spectrum there is a Debian packaging system - Debian Package Manager (dpkg). Debian packages [20] are similar to RPM packages and are also distributed as binary packages, containing executables and supporting files, or source packages containing the source code and a number of patches modifying the original code for target machine. Packages also specify dependencies which are appropriately handled by the package manager. Due to resemblance between RPM and Debian Package Managers, we shall omit an in-depth description of Debian Package Manager.

3.2.3 Homebrew

Homebrew [21] is said to be the missing package manager for OS X. Although OS X comes with useful utilities and packages and also provides its own App Store, common packages, primarily from the Unix world, are not directly available.

Homebrew helps building packages using simple Ruby scripts. It provides a high level API to support package authors with common package installation tasks such as invoking external commands and providing inline patches.

The following is a simple example of a script (formula) for one of the packages (taken from [22]). The script starts with several meta data information (description, homepage, location of the source package and a SHA256 fingerprint allowing Homebrew to validate the download. Additional property head allows package authors to specify where development branches may be downloaded. Dependencies can be specified as mandatory, optional and recommended

- **mandatory** dependencies are always installed
- **optional** dependencies may be installed but are disabled by default
- **recommended** dependencies will be installed by default but may be disabled

Additional switch (:build) can be used to specify a build-time only dependency as shown in the example below where cmake tool is used.

```
class Polarssl < Formula
  desc "Cryptographic & SSL/TLS library"
  homepage "https://tls.mbed.org/"
  url "https://tls.mbed.org/download/mbedtls-2.1.0-gpl.tgz"
  sha256 "b61b5fe6aa33ed365289478ac48f1496b97eef0fb813295e534e0c2bd435dcfc"
  head "https://github.com/ARMmbed/mbedtls.git"

  depends_on "cmake" => :build

  def install
    # "Comment this macro to disable support for SSL 3.0"
    inreplace "include/mbedtls/config.h" do |s|
      s.gsub! "#define MBEDTLS_SSL_PROTO_SSL3", "//#define MBEDTLS_SSL_PROTO_SSL3"
    end

    system "cmake", *std_cmake_args
    system "make"
    system "make", "install"

    # Why does PolarSSL ship with a "Hello World" executable. Let's remove that.
    rm_f "#{bin}/hello"
    # Rename benchmark & selftest, which are awfully generic names.
    mv bin/"benchmark", bin/"mbedtls-benchmark"
    mv bin/"selftest", bin/"mbedtls-selftest"
    # Demonstration files shouldn't be in the main bin
    libexec.install "#{bin}/mpi_demo"
  end

  test do
    (testpath/"testfile.txt").write("This is a test file")
    # Don't remove the space between the checksum and filename. It will break.
    expected_checksum = "e2d0fe1585a63ec6009c8016ff8dda8b17719a637405a4e23c0ff81339148249
testfile.txt"
    assert_equal expected_checksum, shell_output("#{bin}/generic_sum SHA256
testfile.txt").strip
  end
end
```

The install script then takes care of applying changes to the code (inreplace), calling specific commands from the system (system, rm_f, mv etc.) and finally installing the package with the libexec tool.

3.2.4 Snappy Ubuntu

Snappy Ubuntu [23] is a new packaging system for Ubuntu supporting transactional updates. It is targeting clouds and (mobile) devices and provides a very streamlined version of the Ubuntu Core [24]. Applications are installed atomically thus supporting seamless roll back if needed. Applications are sandboxed. Thus they are not allowed to access and modify parts of the system outside its own sandbox, with the aim of providing greater levels of security and easier roll-backs. Snappy package is a compressed file containing all the files that comprise the package and a special meta directory containing package's meta information: name, version, vendor etc. It also describes the list of binaries and services provided by the package as well as required capabilities of the system that are mandatory for proper functioning of a package.

The following listing shows a very simple Snappy package [25].

```
name: xkcd-webserver
version: 0.5
vendor: Snappy Developers <snappy-devel@lists.ubuntu.com>
icon: meta/xkcd.png
services:
 - name: xkcd-webserver
```

```
    start: bin/xkcd-webserver
    description: A fun webserver
    caps:
    - networking
    - network-service
```

3.2.5 Docker

Docker [18] has revolutionised the use of Linux containers. It has started as a layer on top of LXC technology simplifying the preparation of containers. Later it has completely replaced LXC with its own execution environment allowing Docker to introduce new concepts.

Although Docker works with containers rather than full virtual images it is nevertheless valuable from the perspective of the application packaging, as it has already been identified in the MIKELANGELO proposal: Capstan was inspired by the ease of use of Docker. The cornerstone of a Docker container is a single file called Dockerfile comprised of several commands instructing Docker how to build the container. A container typically starts with a command specifying the base container (FROM <container name> command). This will notify Docker that when building a container, it must first download the appropriate base container if it hasn't used it before. One caveat a user must pay special attention to is that when running a container it will use the kernel from the underlying host and not actually the base container specified in this command.

Other commands Dockerfile supports allow users to execute commands at certain points in time. Some of the most important are presented next:

- RUN is used when building a container. It runs any command specified as a parameter and applies it as an additional layer on top of previous one. This means, that after every RUN command, a new layer comprised of previous state and additional changes is created. Docker optimises the storage management so that layers do not escalate in size.
- CMD is executed once a container is started. In contrast to the RUN command which is used during build-time, the CMD can only be issued once. Even if there are two or more CMD instructions, only the last one will prevail. Users are allowed to override the default command when starting a container.
- ENV allows setting environment variables will be used for all succeeding Docker commands as well as in a running container. A powerful feature of the Docker command line tool is the ability to check all available environment variables (docker inspect) which gives the end user the ability to understand which customisation points the container provides.
- ADD and COPY are used to copy files from various sources to the container.
- VOLUME defines a mount point given to the container. It may be provided either by the host itself or by another container.

- LABEL facilitates the description of arbitrary metadata that may be attributed to the container.

In the following paragraphs we present the actual procedure for building of the Docker container, supporting our OpenFOAM use case [6]. Given we are considering Docker containers as the closest so-called relatives to our packages, we are presenting the concrete Dockerfile, used to build the container for the said use case.

It starts by providing the initial container layer which is based on Ubuntu 14.04. It then makes sure that the appropriate DEB source (deb http://www.openfoam.org/download/ubuntu trusty main) is available and runs some "apt-get" commands. Because 'YES' is forced, the installation of all dependent packages is automatic.

As we have seen before, OpenFOAM requires that the environment is properly initialised. Since we are not allowed to add more than one CMD commands in Dockerfile, we decided to request this initialisation as part of user's bashrc script (it should be noted that even if it were possible to add more than one CMD, the end user is always allowed to replace it when running the container).

We then also add the actual OpenFOAM input case and provide a default command executed upor docker start. This could be replaced with a VOLUME or even given to the user to set when specifying the properties of a container to execute.

```
FROM ubuntu:14.04
ADD openfoam.list /etc/apt/sources.list.d/

RUN ["apt-get", "update"]
RUN ["apt-get", "install", "-y", "--force-yes", "openfoam231"]

RUN echo ". /opt/openfoam231/etc/bashrc" >> /root/.bashrc

ADD airFoil2D_04 /work/

CMD . /opt/openfoam231/etc/bashrc && simpleFoam -case /work
```

Based on Dockerfile, the container's metadata is stored as an "image". The format of the Docker image metadata is described in [26]. Open Container initiative [27], backed by Linux Foundation a number of sponsors including Docker, Google, Red Hat and Microsoft, is an attempt to make this a truly open format for container-based technologies. This will increase the interoperability of different container technologies and prevent future vendor lock-ins.

3.2.6 Summary of Existing Systems

Packaging systems presented in the previous sections all provide common features summarised in the following list:

- installation of binary packages
- installation from source code, including mechanisms to modify original code for specific needs
- dependency tracking and management
- package versioning
- use of rich metadata facilitating search
- a toolset supporting package generation

Apart from the RPM most of the others are actually very specific to the target systems they are intended to be used in. This renders them difficult to be used directly for the purposes of the OSv packaging system. RPM, on the other hand, has no system-specific requirement which makes it a plausible candidate for packaging OSv applications. However, the sheer complexity of the RPM system and the overhead of capabilities it provides makes it very impractical to use in the context of OSv.

3.3 Design of the MIKELANGELO Package Manager

The initial business requirements gathered from the perspective of internal use cases have shown that at least for this early phase of the project application packaging does not represent a critical requirement for the MIKELANGELO stack. This can be attributed to the fact that all use cases are used to configure their applications manually in their existing environments.

The following is a list of high level requirements that have been identified in the initial requirements gathering process:

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 machine, i.e. contextualisation of VMs. Contextualisation should be enabled during 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 image (Glance) and compute service (Nova). This would simplify the deployment of images and instantiation of virtual machines.

REQ 52: Capstan must allow using OpenStack Image Service as base images to be extended. Capstan provides an internal repository of built images serving as the potential base images. The user can either use the official repository or use her own. This requirement would simplify the use of existing base images from users' OpenStack cloud environment augmenting them with additional packages and data.

All of the aforementioned requirements were targeting the Capstan tool. During the design of the application packaging and review of other tools, these requirements have been evaluated against the goals of MIKELANGELO, the provided requirements and the existing package management systems.

A high level design of the MIKELANGELO package management is presented in the following figure (Figure 2). The end user should only interact with the MPM client tool. This will in turn use the package manager functionalities to validate and build packages. Package, as we will describe in the following subsection, is a compressed file of everything the package author needs for the package to function properly. Package is thus no longer a virtual image.

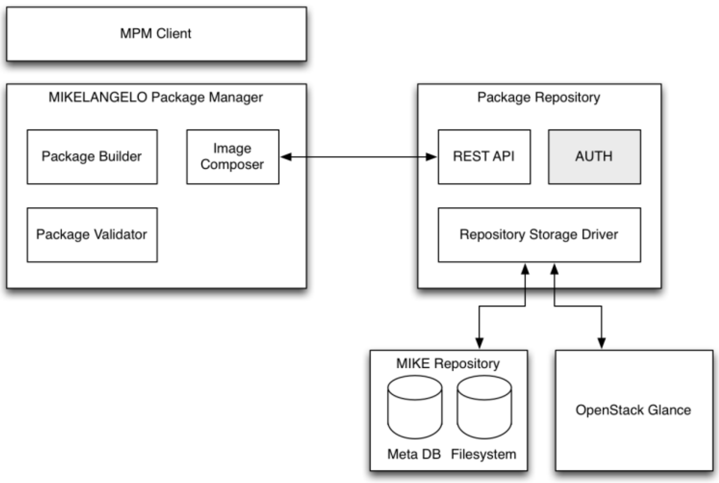

Figure 2: Architecture of the MIKELANGELO Package Manager.

In order to simplify use of packages in live environments, MPM will also support generation of target virtual images through the Image Builder component. These images will be suitable for various virtual environments (QEMU, OpenStack, Amazon etc.).

The following subsections present the crucial parts of the package management.

3.3.1 MIKELANGELO Package Manager Client Command Line Tool

The command line tool of the MIKELANGELO packaging will support package authors to create valid and solid packages suitable for use on top of the MIKELANGELO stack. This section presents some of the main functionalities provided by this tool.

Package initialisation. In previous sections we have presented the structure of a package. The tool will provide a command to instantiate a new package with some basic information (for example name and author). This will provide a basic structure of the package the author will be able to immediately start working on.

Package validation. Validation of a package consists of several tests. First, if the application is a native binary, the validation must verify that it has been compiled into an OSv-compliant binary (for example, shared library). Validation may also verify if dependencies are satisfied (for example all referenced shared libraries are also part of this package) and, if they are not, provide the author with a list of missing dependencies that should be either added into this package or referenced as an external package. Validation will also verify the metadata description of a package and alert the author of any identified issues (for example formatting error in the metadata file).

Package building. Once the author is satisfied with the package content, the tool will also provide a shortcut for building the final package file by compressing the contents of the package directory.

Image composition. Using the package and all its dependencies, the tool will also support generation of virtual machine images in various formats (e.g., QCOW2 or RAW). This will allow integration with OpenStack and HPC environments without any changes to existing systems.

Image testing. If package author provides a testing hook, the tool will also enable testing of the built image.

Other capabilities will be defined in the second and third year once the actual real-world demonstrators will be supported by the MIKELANGELO stack.

3.3.2 Structure of the MIKELANGELO Application Package

As we have already discussed in previous sections, the two existing approaches used in OSv build regular virtual images (for example QCOW2 or RAW). With MPM we propose to change the notion of a package: **MPM Package is a compressed file containing all the required content of the package and additional metadata.**

The content is what needs to be uploaded to the image but only when used by the end user. It can contain arbitrary hierarchy of files. The initial version will have no explicit rules enforcing end users to use specific locations for files of certain type (e.g. binaries, shared libraries, supporting files). However, in future versions we are going to investigate whether additional rules for certain types of files would be reasonable to be more consistent among all potential packages that may be included in the target virtual image. This could be similar to the way different types of files are stored in Linux (/usr/bin, /var, ...). Alternatively, MPM could take approach similar to that of Snappy. All binaries that can be used by package users should be specified explicitly in the metadata. Other files are separated into package root directory in the target image.

A special folder named "meta" should be placed inside a package. This will contain the metadata of the package, such as it's name, description, author and optionally an icon of the package. Part of the package's metadata is also a list of executables and a list of services. Executable is any OSv-compliant binary that is supposed to be executed by end users. Services are OSv-compliant binaries offering specific services to external applications, e.g., a database or an NFS server. Dependencies of the underlying package are also an important part of the package's metadata. Dependencies may provide the OSv kernel, additional required libraries, or complete application modules (HTTP or NFS server, for example).

Metadata folder is ignored when uploading package into the actual virtual image. However it is used as part of the package discovery in the repository as well as when composing images consisting of several packages.

3.3.3 Package Hooks

Hooks are optional actions that can be executed at specific points of the process. For example, Git uses hooks to perform specific actions when user commits files or pushes them into the remote repository. There are typically two reasons for introducing hooks: decentralisation of configuration and modularity/extensibility of the process.

Both are important for MIKELANGELO application packaging due to the fact that most of the actual work related to building the application package and adding them into the target image is done by authors preparing applications compliant with OSv. MPM will thus provide mechanisms and supporting tools to simplify the process for package authors.

Hooks are a special part of the package's metadata. Thus, they must be placed in the "meta/hooks" subdirectory of the package. First version of MPM is going to support the following hooks:

- **meta/hooks/build** may be used to produce the content of the package suitable for uploading into a virtual image. This can include downloading of the original source tree, configuration and building. This hook is executed in the host environment so it can be an arbitrary script (BASH, Python, ...).
- **meta/hooks/run** should specify the command that needs to be executed whenever the virtual machine is started. The hook should also allow specifying additional predefined environment variables.
- **meta/hooks/java-run** specifies the configuration for running Java applications. This consists of the Java Virtual Machine configuration options and the application command line (name of the class and it's arguments). Java must be treated differently from native apps because multiple Java applications running within one OSv instance also run within a single Java Virtual Machine (JVM). All the JVM configuration options must thus be considered and merged into one, meaningful configuration of JVM.
- **meta/hooks/test** may be used to perform any kind of testing facility to validate that the application package is properly configured.

Both meta/hooks/run and meta/hooks/java-run are executed in the host environment updating the startup information of the OSv virtual image. Therefore, they require interaction with the OSv tools from the target image to upload content and specify runtime commands. Two options for these hooks are still being investigated and will be analysed during the development of the application packaging facility. The first option is to design a format with all mandatory and optional options. The other is to provide an API and a supporting library that can be used as part of the packaging process, similarly to the way authors specify the command to be executed on VM boot. We believe that the latter, although being slightly more difficult for non-programmers, is more flexible and robust to user errors.

All hooks are optional and used only if corresponding file is found in the package's metadata. They are executed and used in appropriate phases of the process. There should be no additional files in the hooks directory. If required, they should be placed inside meta/hooks/support directory. The MPM tool will provide functionality to validate the content of the package.

Although there is only small number of hooks planned for the first version of the MIKELANGELO packaging, this mechanism will allow us to extend its capabilities with new features in later stages of the project. One such extension would be a hook specifying the HTTP REST API of the application package and it's implementation (for example, OpenFOAM may introduce an HTTP REST API allowing end users to control the behaviour of the simulation).

The following figure (Figure 3) details the relation of various MPM tools with the aforementioned hooks.

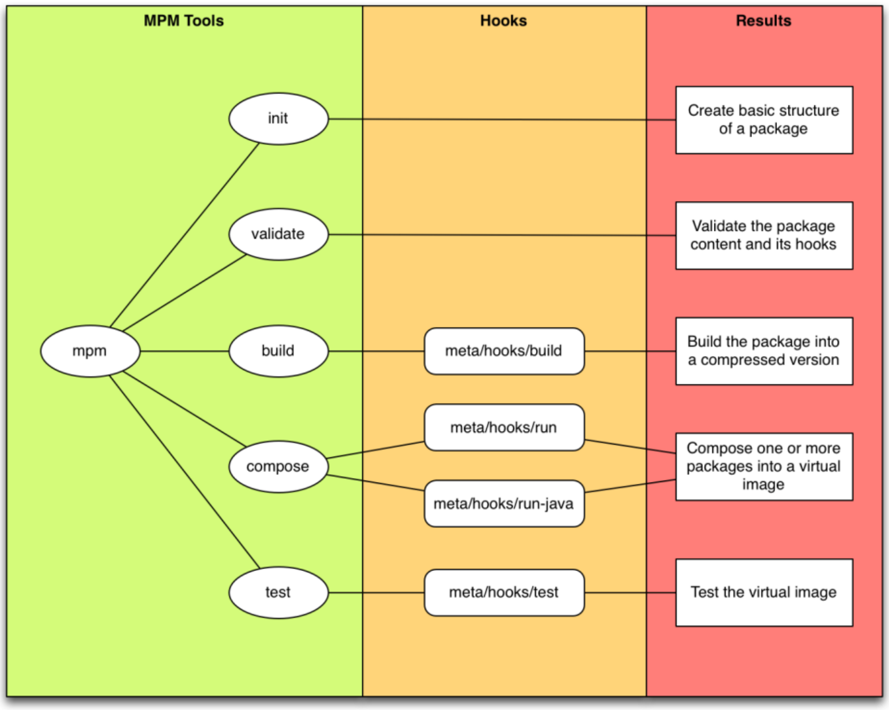
MPM Tools
Hooks
Results
mpm
init
validate
build
compose
test
meta/hooks/build
meta/hooks/run
meta/hooks/run-java
meta/hooks/test
Create basic structure of a package
Validate the package content and its hooks
Build the package into a compressed version
Compose one or more packages into a virtual image
Test the virtual image

Figure 3: Relationships between the tools provided by the MIKELANGELO Package Manager, optional author-provided hooks and results.

3.3.4 Customisation and Configuration of Packages

Application packages may also provide custom configuration switches for controlling the behaviour of the application when executed. These should be specified as part of the package's metadata (for example, a port number at which the service listens for client connections, an NFS mount URI or even just the name of the binary that is to be executed by the VM). Package authors should be able to expose such configuration options to end users. For this, we propose another metadata file meta/opts with a simple structure

- **long option name** represents the long name of the option (e.g. --nfs-mount)
- **short option name** represents the short name of the option (e.g. -n)
- **description** should be provide a brief description of the option for the help message presented to the end user
- **default value** holds the default value for this option if not provided by the end user

At least one of the names should be provided, but preferably both. Description and default values are both optional. However, if the default value is omitted, it should be handled by the application itself.

3.3.5 Integration with OpenStack

OpenStack uses Glance as its central virtual machine image repository. The repository stores virtual images in various widely used file formats (QCOW2, RAW, Amazon images, a Docker container etc.). Virtual images can be uploaded into Glance through Horizon, the central GUI of OpenStack, or via the Glance API.

MPM will support building at least QCOW2 and RAW images resulting in packaged applications that can directly be stored in Glance and executed on OpenStack Cloud. However, the packaging of applications for OSv will be much more flexible. Thus, we are going to investigate different ways supporting integration of packages in their raw format (compressed file) directly into the Glance. Once requested by Nova, Glance will serve these packages and provide tools allowing on-the-fly generation of target images.

One of the things task T4.3 in collaboration with work package 5 will investigate is whether such approach is reasonable when executed in real-world environments (for example, the Big Data use case). One of the crucial aspects of on-the-fly generation of images is whether it can outperform booting standard images due to its smaller size.

3.4 Analysis of Existing OSv Applications

OSv applications [15] is a dedicated repository, hosted at Github, where community provides their modifications to some of the most popular applications used in cloud applications. These include Apache Spark, Cassandra NoSQL database, memcached, Jetty application container and MySQL to name just a few. These applications do not contain any sources or binaries. Instead, they provide scripts that acquire the code and prepare it for upload onto virtual image when requested by the OSv build script or Capstan.

Although each application has its own way of building it, they share some common ideas. Basically all use Makefile which is automatically invoked by the OSv build script. Most commonly used steps in the Makefile are the following:

- set local and environment variables
- download files from the internet and unpack compressed files
- file and directory manipulations (e.g. copy, move, create directory)
- patch files with changes necessary to build compliant binaries
- compile application sources, typically using Makefile
- use other system utilities (e.g. install, find, echo)
- generate usr.manifest file used to control the content that’s uploaded to the image

4 The Seastar API

In the first research paper on OSv [7], one of the benchmarks used was Memcached [8], a popular cloud application used for caching of frequently requested objects to lower the load on slower database servers. We will continue to use this benchmark for the MIKELANGELO work as well (see Section 7 below). Memcached demonstrated how an unmodified network application can run faster on OSv than it does on Linux - a 22% speedup was reported in the paper.

22% is an impressive speedup that we get just by replacing Linux in the guest by OSv, without modifying the application. Additional small speed-ups may be achieved by continuing to optimize OSv, and this is one part of our plan. The other involves an in-depth analysis of the kernel and applications understanding what is actually causing major bottlenecks. Overcoming these bottlenecks may result in 2-time, 4-time or even higher speedups.

When we profiled memcached on OSv, we quickly discovered two performance bottlenecks:

1. **Inefficiencies inherent in the Posix API**, so OSv cannot avoid them and still remain Posix compatible: For example, in one benchmark we noticed that 20% of the memcached runtime was locking and unlocking mutexes - almost always uncontended. For every packet we send or receive, we lock and unlock more than a dozen mutexes. Part of OSv's performance advantage over Linux is that OSv uses a "netchannel" design for the network stack reducing locks (see Section 2), but we still have too many of them, and the Posix API forces us to leave many of them: For example, the Posix API allows many threads to use the same socket, allows many threads to modify the list of file descriptors, to poll the same file descriptors - so all these critical operations involve locks, that we cannot avoid. The socket API is also synchronous, meaning that when a send() returns the caller is allowed to modify the buffer, which forces the network code in OSv to not be zero-copy.
2. **Unscalable application design:** It is not easy to write an application to scale linearly in the number of cores in a multi-core machine, and many applications that work well on one or two cores, scale very badly to many cores. For example memcached keeps some global statistics (e.g., the number of requests served) and updates it under a lock - which becomes a major bottleneck when the number of cores grow. What might seem like an acceptable solution - lock-free atomic variables - is also not scalable, because while no mutex is involved, atomic operations, and the *cache line bounces* (as different CPUs read and write the same variable), both become slower as the number of cores increase. So writing a really scalable application - one which can run on (for example) 64 cores and run close to 64 times faster than it does on a single

core - is a big challenge, one with which the traditional Linux APIs rarely help with, and therefore most applications are not as scalable as they should be - which will become more and more noticeable as the number of cores per machine continues to increase.

In [7], we tried an experiment to quantify the first effect - the inefficiency of the Posix API. The subset of memcached needed for the benchmark was very simple: a request is a single packet (UDP), containing a “GET” or “PUT” command, and the result is a single UDP packet as well. So we implemented in OSv a simple “packet filter” API: every incoming ethernet packet gets processed by a function (memcached’s hash-table lookup) which immediately creates the response packet. There is no additional network stack, no locks or atomic operations (we ran this on a single CPU), no file descriptors, etc. The performance of this implementation (as reported in [7]) was an impressive 4 times better than the original memcached server!

But while the simple “packet filter” API of [7] was useful for the trivial UDP memcached, it was not useful for implementing more complex applications, for example applications which are asynchronous (cannot generate a response immediately from one request packet), use TCP or need to use multiple cores. Fast “packet filter”-like APIs are already quite commonly used, and DPDK [9] is a good example, and are excellent to implement routers and similar packet-processing software; But they are not really helpful if you try to write a complex, highly-asynchronous network applications of the kind that is often used on the cloud - such as a NoSQL database, HTTP server, search engine, and so on.

For the MIKELANGELO project, we set out to design a new API which could answer both above requirements: An API which new applications can use to achieve optimal performance (i.e., the same level of performance achieved by the “packet filtering API” implementation), while at the same time allows the creation of complex real-life applications: The result of this design is *Seastar*:

- Seastar is a C++14 library, which can be used on both OSv and Linux. Because Seastar bypasses the kernel for most things, we do not expect additional speed improvements by running it on OSv - though some of OSv’s other advantages (such as image size and boot time) may still be relevant.
- Seastar is not part of the kernel, but is nevertheless an integral part of the MIKELANGELO guest operating system, needed by new applications that choose to use it to go beyond the improvement OSv can offer to unmodified applications. For example, in the “Cloud Bursting” use case we will show how a re-implementation of Cassandra using Seastar will provide a performance boost much greater than we could get by running Cassandra on OSv.
- Seastar is designed for the needs of complex asynchronous server applications of the type common on the cloud - e.g., NoSQL databases, HTTP servers, etc. Here

"asynchronous" means that a request usually triggers a cascade of events (disk reads, communication with other nodes, etc.) and only at a later time can the reply be composed.

- Seastar provides the application the mechanisms it needs to solve both performance bottlenecks mentioned at the top of this section - achieve optimal efficiency on one core, as well as scalability in the number of cores. We'll explain how Seastar does this below.
- Seastar bypasses the legacy kernel APIs. Notably, it directly accesses the network card directly using DPDK. Seastar provides a full TCP/IP stack (which DPDK does not).

We've reimplemented memcached using Seastar, and measured 2 to 4-fold performance improvement over the original memcached as well as near-perfect scalability to 32 cores (something which the "packet filter" implementation couldn't do). Figure 4 below for more details.

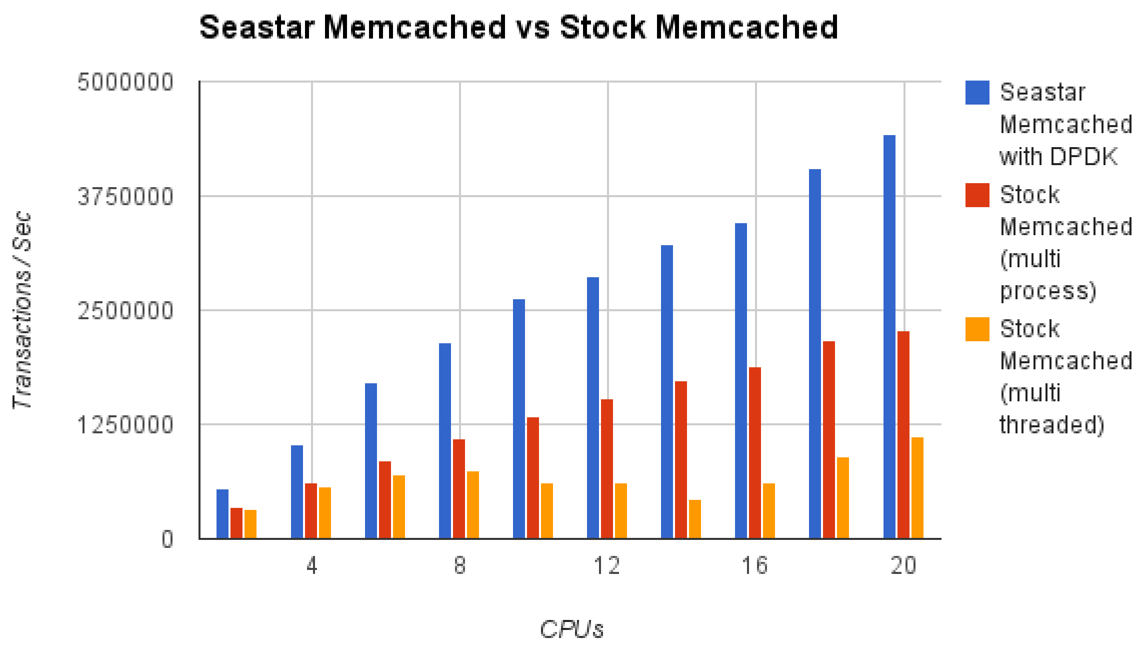

Figure 4: Performance of stock memcached (orange) vs Seastar reimplementation of memcached (blue), using TCP and the memaslap [12] workload generator - for varying number of cores The red bars show a non-standard memcached deployment using multiple separate memcached processes (instead of one memcached with multiple threads); Such a run is partially share-nothing (the separate processes do not share memory or locks) so performance is better than the threaded server, but still the kernel and network stack is shared so performance is not as good as with Seastar.

How can an application designed to use Seastar be so much faster than one using more traditional APIs such as threads, shared memory and sockets? The short answer is that modern computer architecture has several performance traps that are easy to fall into, and Seastar ensures that you don't by using the following architecture:

1. **Sharded ("share nothing") design**
   Modern multi-core machines have shared memory, but using it incorrectly can decimate an application's performance: Locks are very slow, and so are processor-provided "lock-free" atomic operations and memory fences. Reading and writing the same memory object from different cores significantly slows down processing compared to one core finding the object in its cache (this phenomenon is known as "cache line bouncing"). All of these slow operations already hurt one-core performance, but get progressively slower as the number of cores increases, so it also hurts the scaling of the application to many cores.
   Moreover, as the number of cores increases, multi-core machines inevitably become multi-socket, and we start seeing NUMA (non-uniform memory access) issues. I.e., some cores are closer to some parts of memory - and accessing the "far" part of memory can be significantly slower.
   So Seastar applications use a **share-nothing** design: Each core is responsible for a part (a "**shard**") of the data, and does not access other cores' data directly - if two cores wish to communicate, they do so through message passing APIs that Seastar provides (internally, this message passing uses the shared memory capabilities provided by the CPU).
   When a Seastar application starts on N cores, the available memory is divided into N sections and each core is assigned a different section (taking NUMA into account in this division of memory, of course). When code on a core allocates memory (with malloc(), C++'s new, etc.), it gets memory from this core's memory section, and only this core is supposed to use it.
   The following figure (Figure 5) shows the difference between traditional stack and the one introduced by Seastar.

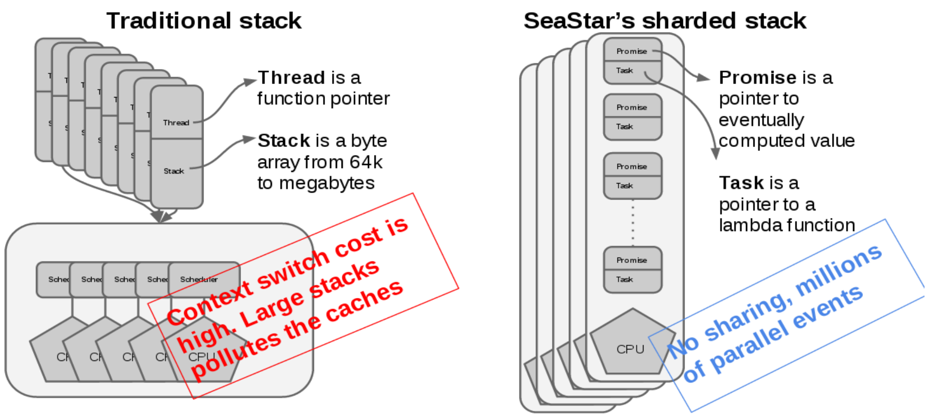
Traditional stack
Thread
Stack
Thread is a function pointer
Stack is a byte array from 64k to megabytes
Scheduler
CPU
Context switch cost is high. Large stacks pollutes the caches
SeaStar's sharded stack
Promise
Task
Promise is a pointer to eventually computed value
Task is a pointer to a lambda function
CPU
No sharing, millions of parallel events

Figure 5: Comparison of traditional and Seastar's shared stack.

2. **Futures and continuations, not threads**
   For more than a decade, it has been widely acknowledged that high-performance server applications cannot use a thread per connection, as those impose huge memory consumption (thread stacks are big) and significant context switch overheads. Instead, the application should use just a few threads (ideally, just one thread per core) which each handles many connections. Such a thread usually runs an "event loop" which waits for new events on its assigned connections (e.g., incoming request, disk operation completed, etc.) and processes them. However, writing such "event driven" applications is traditionally very difficult because the programmer needs to carefully track the complex state of ongoing connections to understand what each new event means, and what needs to be done next. Seastar applications also run just one thread per core. Seastar implements the **futures and continuations** API for asynchronous programming, which makes it easier (compared to classic event-loop programming) to write very complex applications with just a single thread per core. A **future** is returned by an asynchronous function, and will eventually be fulfilled (become ready), at which point a **continuation**, a piece of non-blocking code, can be run. The continuations are C++14 lambdas, anonymous functions which can capture state from the enclosing code, making it easy to track what a complex cascade of continuations is doing. We explain Seastar's futures and continuations in more detail below.
   The future/continuation programming model is not new and has been used before in various application frameworks (e.g., Node.js), but before Seastar, it was only partially implemented by the C++14 standard (std::future). Moreover, Seastar's implementation of futures are much more efficient than std::future because Seastar's implementation uses less memory allocation, and no locks or atomic operations: A future and its continuation belong to one particular core, thanks to Seastar's sharded design.
3. **Asynchronous disk I/O**
   Continuations cannot make blocking OS calls, or the entire core will wait and do nothing. So Seastar uses the kernel's AIO ("asynchronous") mechanisms instead of the traditional Unix blocking disk IO APIs. With AIO, a continuation only starts a disk operation, and returns a *future* which becomes available when the operation finally completes.
   Asynchronous I/O is important for performance of applications which use the disk, and not just the network: A popular alternative (used in, for example, Apache Cassandra) is to use a pool of threads processing connections, so when one thread blocks on a disk access, a different thread gets to run and the core doesn't remain idle. However, as explained above, using multiple threads has non-negligible performance overheads, especially in an application (like Cassandra) which may need

to do numerous concurrent disk accesses. With modern SSD disk hardware, concurrent non-sequential disk I/O is no longer a bottleneck (as it was with spinning disks and their slow seek times), so the programming framework should not make it one.

4. **Userspace network stack**
   Seastar can optionally bypass the kernel's (Linux's or OSv's) network stack and all its inherent inefficiencies like locks, by providing its own network stack: Seastar accesses the underlying network card directly, using either DPDK (on Linux or OSv) or virtio (only supported on OSv). On top of that, it provides a full-featured TCP/IP network stack, which is itself written in Seastar (futures and continuations) and correspondingly does not use any locks, and instead divides the connections among the cores; Once a connection is assigned to a core, only this core may use it. The connection will only be moved to a different core if the application decides to do so explicitly).
   Importantly, Seastar's network stack supports multiqueue network cards and RSS (receive-side steering), so the different cores can send and receive packets independently of each other without the need for locks and without creating bottlenecks like a single core receiving all packets. When the hardware's number of queues is limited below the number of available cores, Seastar also uses software RSS - i.e., some of the cores receive packets and forward them to other cores.

4.1 A Taste of Seastar: Futures and Continuations

A complete tutorial of the Seastar API is beyond the scope of this deliverable, but we do plan to write one later as part of the project: We have already released Seastar as open source (http://seastar-project.org), and an open-source library would not be very useful if it is not well documented. Here, just to give the reader a taste of the Seastar API, we will have a partial look at how futures and continuations are used by a Seastar application.

Futures and continuations, which we will introduce now, are the building blocks of asynchronous programming in Seastar. Their strength lies in the ease of composing them together into a large, complex, asynchronous program, while keeping the code fairly readable and understandable.

A **future** is a result of a computation that may not be available yet. Examples include:

- a data buffer that we are reading from the network
- the expiration of a timer
- the completion of a disk write
- the result of a computation that requires the values from one or more other futures.

The type `future<int>` variable holds an integer that will eventually be available - at this point might already be available, or might not be available yet. The method `available()` tests if a value is already available, and the method `get()` gets the value. The type `future<>` (with empty brackets) indicates something which will eventually complete, but not return any value.

A future is usually returned by a **promise**, also known as an **asynchronous function**, a function or object which returns a future and arranges for this future to be eventually resolved. One simple example is Seastar's function `sleep()`:

```
future<> sleep(std::chrono::duration<Rep, Period> dur);
```

This function arranges a timer so that the returned future becomes available (without an associated value) when the given time duration elapses.

A **continuation** is a callback (typically a C++ lambda) to run when a future becomes available. A continuation is attached to a future with the `then()` method. Here is a simple example:

```
#include "core/sleep.hh"
#include <iostream>

void f() {
    std::cout << "Sleeping... " << std::flush;
    using namespace std::chrono_literals;
    sleep(1s).then([] {
        std::cout << "Done.\n";
        engine_exit();
    });
}
```

In this example we see us getting a `sleep(1s)` future, and attaching to it a continuation which prints a message and exits. The future will become available after 1 second has passed, at which point the continuation is executed. Running this program, we indeed see the message "Sleeping..." immediately, and one second later the message "Done." appears and the program exits.

So far, this example was not very interesting - there is no parallelism, and the same thing could have been achieved by the normal blocking POSIX `sleep()`. Things become much more interesting when we start several `sleep()` futures in parallel, and attach a different continuation to each. Futures and continuation make parallelism very easy and natural:

```
#include "core/sleep.hh"
```

```
#include <iostream>

void f() {
    std::cout << "Sleeping... " << std::flush;
    using namespace std::chrono_literals;
    sleep(200ms).then([] { std::cout << "200ms " << std::flush; });
    sleep(100ms).then([] { std::cout << "100ms " << std::flush; });
    sleep(1s).then([] { std::cout << "Done.\n"; engine_exit(); });
}
```

Each `sleep()` and `then()` call returns immediately: `sleep()` just starts the requested timer, and `then()` sets up the function to call when the timer expires. So all three lines happen immediately and `f` returns. Only then, the event loop starts to wait for the three outstanding futures to become ready, and when each one becomes ready, the continuation attached to it is run. The output of the above program is of course:

```
$ ./a.out
Sleeping... 100ms 200ms Done.
```

`sleep()` returns `future<>`, meaning it will complete at a future time, but once complete, does not return any value. More interesting futures specify a value of any type (or multiple values) that will become available later. In the following example, we have a function returning a `future<int>`, and a continuation to be run once this value becomes available. Note how the continuation gets the future's value as a parameter:

```
#include "core/sleep.hh"
#include <iostream>

future<int> slow() {
    using namespace std::chrono_literals;
    return sleep(100ms).then([] { return 3; });
}

void f() {
    slow().then([] (int val) {
        std::cout << "Got " << val << "\n";
        engine_exit();
    });
}
```

The function `slow()` deserves more explanation. As usual, this function returns a future immediately, and doesn't wait for the sleep to complete, and the code in `f()` can chain a continuation to this future's completion. The future returned by `slow()` is itself a chain of futures: It will become ready once sleep's future becomes ready and then the value 3 is returned.

This example begins to show the convenience of the futures programming model, which allows the programmer to neatly encapsulate complex asynchronous operations. slow() might involve a complex asynchronous operation requiring multiple steps, but its user can use it just as easily as a simple sleep(), and Seastar's engine takes care of running the continuations whose futures have become ready at the right time.

Ready futures: A future value might already be ready when then() is called to chain a continuation to it. This important case is optimized, and *usually* the continuation is run immediately instead of being registered to run later in the next iteration of the event loop.

This optimization is done *usually*, though sometimes it is avoided: The implementation of `then()` holds a counter of such immediate continuations, and after many continuations have been run immediately without returning to the event loop (currently the limit is 256), the next continuation is deferred to the event loop in any case. This is important because in some cases we could find that each ready continuation spawns a new one, and without this limit we can starve the event loop. It important not to starve the event loop, as this would starve continuations of futures that weren't ready but have since become ready, and also starve the important **polling** done by the event loop (e.g., checking whether there is new activity on the network card).

`make_ready_future<>` can be used to return a future which is already ready. The following example is identical to the previous one, except the asynchronous function `fast()` returns a future which is already ready, and not one which will be ready in a second as in the previous example. The nice thing is that the consumer of the future does not care, and uses the future in the same way in both cases.

```
#include "core/future.hh"
#include <iostream>

future<int> fast() {
    return make_ready_future<int>(3);
}

void f() {
    fast().then([] (int val) {
        std::cout << "Got " << val << "\n";
        engine_exit();
    });
}
```

Capturing state in continuations: We've already seen that Seastar *continuations* are lambdas, passed to the `then()` method of a future. In the examples we've seen so far, lambdas have been nothing more than anonymous functions. But C++11 lambdas have one more trick up their sleeve, which is extremely important for future-based asynchronous programming in Seastar: Lambdas can **capture** state. Consider the following example:

```
#include "core/sleep.hh"
#include <iostream>

future<int> incr(int i) {
    using namespace std::chrono_literals;
    return sleep(10ms).then([i] { return i + 1; });
}

void f() {
    incr(3).then([] (int val) {
        std::cout << "Got " << val << "\n";
        engine_exit();
    });
}
```

The asynchronous function `incr(i)` takes some time to complete (it needs to sleep a bit first...), and in that duration, it needs to save the `i` value it is working on. In the early event-driven programming models, the programmer needed to explicitly define an object for holding this state, and to manage all these objects. Everything is much simpler in Seastar, with C++11's lambdas: The *capture syntax* `[i]` in the above example means that the value of `i`, as it existed when `incr()` was called, is captured into the lambda. The lambda is not just a function - it is in fact an *object*, with both code and data. In essence, the compiler created for us automatically the state object, and we neither need to define it, nor to keep track of it (it gets saved together with the continuation, when the continuation is deferred, and gets deleted automatically after the continuation runs).

One implementation detail worth understanding is that when a continuation has captured state and is run immediately, this capture incurs no runtime overhead. However, when the continuation cannot be run immediately (because the future is not yet ready) and needs to be saved till later, memory needs to be allocated on the heap for this data, and the continuation's captured data needs to be copied there. This has runtime overhead, but it is unavoidable, and is very small compared to the parallel overhead in the threaded programming model (in a threaded program, this sort of state usually resides on the stack of the blocked thread, but the stack is much larger than our tiny capture state, takes up a lot of memory and causes a lot of cache pollution on context switches between those threads).

In the above example, we captured `i` *by value* - i.e., a copy of the value of `i` was saved into the continuation. C++ has two additional capture options: capturing by *reference* and capturing by *move*.

Using capture-by-reference in a continuation is almost always a mistake, and would lead to serious bugs. For example, if in the above example we captured a reference to `i`, instead of a copy to it,

```
future<int> incr(int i) {
    using namespace std::chrono_literals;
    return sleep(10ms).then([&i] { return i + 1; });   // The "&" here is wrong.
}
```

this would have meant that the continuation would contain the address of `i`, not its value. But `i` is a stack variable, and the `incr()` function returns immediately, so when the continuation eventually gets to run, long after `incr()` returns, this address will contain unrelated content.

Using capture-by-*move* in continuations, on the other hand, is valid and very useful in Seastar applications. By **moving** an object into a continuation, we transfer ownership of this object to the continuation, and make it easy for the object to be automatically deleted when the continuation ends. For example, consider an ordinary (synchronous) function taking a `std::unique_ptr`.

```
int do_something(std::unique_ptr<T> obj) {
     // do some computation based on the contents of obj,
     // let's say the result is 17
     return 17;
     // here obj goes out of scope so the compiler delete()s it.
}
```

By using `unique_ptr` in this way, the caller passes an object to the function, but tells it the object is now its exclusive responsibility - and when the function is done with the object, it should delete the object. How do we do the same in an asynchronous function? We end up needing to use the `unique_ptr` in a continuation. The following won't work:

```
future<int> slow_do_something(std::unique_ptr<T> obj) {
    using namespace std::chrono_literals;
    return sleep(10ms).then([obj] {
         return do_something(std::move(obj))}); // WON'T COMPILE
}
```

The problem is that a `unique_ptr` cannot be passed into a continuation by value (this is that the "`[obj]`" is trying to do), as this would require copying it, which is forbidden because it violates the guarantee that only one copy of this pointer exists. We can, however, *move* `obj` into the continuation:

```
future<int> slow_do_something(std::unique_ptr<T> obj) {
    using namespace std::chrono_literals;
    return sleep(10ms).then([obj = std::move(obj)] {
        return do_something(std::move(obj))});
}
```

Here the use of `std::move()` causes obj's move-assignment is used to move the object from the outer function into the continuation. C++11's notion of move (*move semantics*) is similar to a shallow copy, followed by invalidating the source copy (so that the two copies do not co-exist, as forbidden by `unique_ptr`). After moving obj into the continuation, the top-level function can no longer use it (in this case this is of course fine, because we return anyway).

The `[obj = ...]` capture syntax we used here is new to C++14. This is the main reason why Seastar requires C++14, and does not support older C++11 compilers.

5 The virtio-rdma Component

As a part of the RDMA virtualization design that we described in Deliverable 2.13 (Hypervisor Architecture, [2]), virtio-rdma is the major component on the guest OS. It presents itself as an Ethernet device on the guest and takes over the control of communications that are generated by the guest application.

Figure 6 is a general architecture of guest OS that is using virtio-rdma for inter-VM communication. Basically, two types of inter-VM communication are supported: shared memory for VMs on the same host, and virtualized RDMA for VMs on different hosts. virtio-rdma shares part of the route information from the host in order to know whether the communication endpoint is on the same host or on a remote host, and then to decide which protocol to use.

On the other hand, virtio-rdma may be used by guest applications that are implemented with socket or RDMA verbs API. It uses different RDMA virtualization mechanisms to support socket and RDMA verbs API, which were presented in detail in the Hypervisor deliverable D2.13. For guest applications that are implemented with RDMA verbs API but still involve inter-VM communication on the same host, shared memory will not be used as it is not implemented for the RDMA communication stack. However, for such case, RDMA device will take care of the shared memory communication by using the shared RDMA memory regions among guest, host, and the RDMA device.

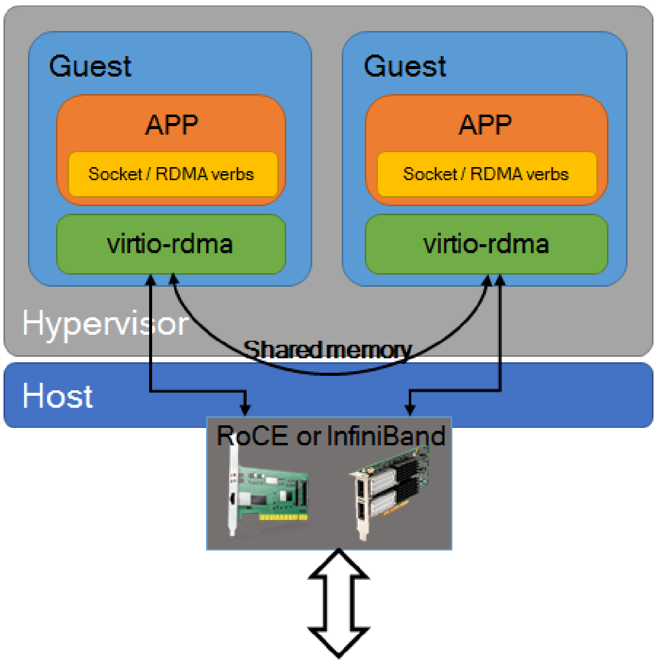

Figure 6: Overview of virtio-rdma component in guest OS.

The RDMA hardware that may be used for this design are InfiniBand and RoCE (RDMA over Converged Ethernet). The RDMA device drivers are installed on the host to support the RDMA calls that are passed by virtio-rdma.

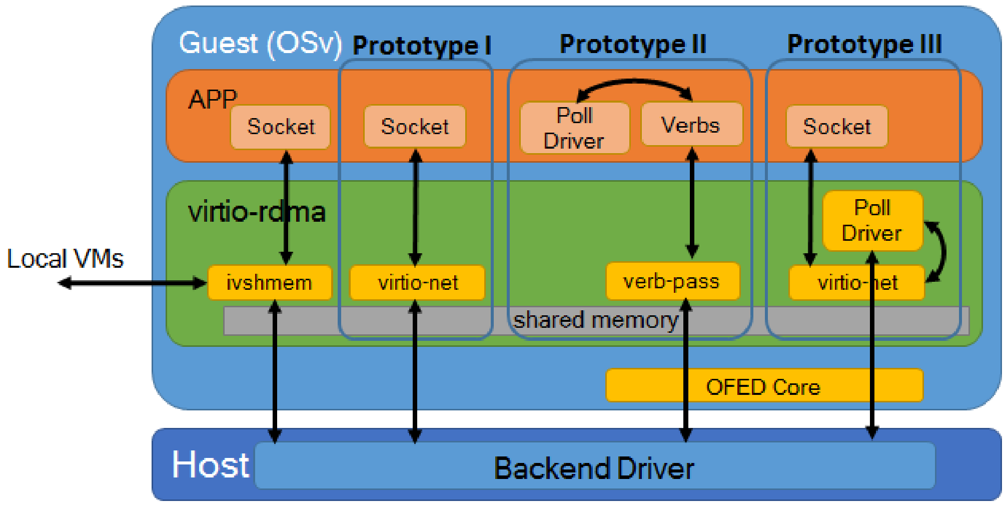

Figure 7: OSv architecture with virtio-rdma component.

virtio-rdma contains several sub modules to support different communication protocols and guest applications, as shown in Figure 7. It also requires specific guest OS support, i.e. the Open Fabric Enterprise Distribution (OFED) [5] core driver, in order to perform direct operation on RDMA memory regions. Open Fabrics is an organization who provides the core library for RDMA standard, and other device manufactures, such as Mellanox [6], normally provide device drivers based on this core library. This core library needs to be supported on OSv later for prototype II (see D2.13, Hypervisor Architecture [2]) next year.

In D2.13 [2], we described three prototypes of RDMA virtualization design. The corresponding part on OSv of the three prototypes are shown in Figure 7. On the left side, ivshmem (a.k.a. Nahanni) [30, 31] module is used for shared memory communication, which is available for all RDMA virtualization prototypes. It shares part of the host main memory for host-guest and guest-guest communication. The ivshmem support on OSv is missing at moment, and it is planned to be implemented in the second year of MIKELANGELO project.

Prototype I has the simplest design on the guest system, which doesn't involve much new implementation. As described in D2.13, the RDMA and poll driver are implemented on the host. The polling results will be sent to guest through vhost-user channel. Vhost-user is a new implementation based on the kernel vhost in the latest versions of QEMU. It works in the user space and uses kernel vhost to initialize the necessary resources that are shared between the processes in the user space. However, most of the communications are taking place in the user space. In this prototype, virtio-rdma is capable of sending and receiving individual L2 packets by multiplexing two other device drivers which do the real work, i.e. the real virtio-net and the ivshmem.

In prototype II, the guest application is directly implemented with RDMA verbs API and a poll driver, which will actively poll the RDMA event queues for completion events of the communication. The RDMA calls, such as memory allocation/release, preparing work requests and communication buffers, are directly performed on the RDMA memory regions by virtio-rdma with the help of the OFED core driver on the guest. Other RDMA calls, such as endpoint management, start/stop data transmission, will be passed through verb-pass module directly to the host, and then the RDMA actions will be performed with the kernel RDMA driver on the host.

In prototype III, socket API is used in guest application, with a poll driver implemented inside virtio-rdma. The difference between prototype I and III is that, for prototype III, the poll driver will directly access the RDMA memory regions without sending any messages to the host through a dedicated communication channel like prototype I does, thus prototype III reduces the amount of messages and notifications between guest and host dramatically.

Generally, prototype II should have the best performance of the three, and prototype III should have better performance than prototype I. These three prototypes are supposed to support different types of use cases and guest applications, and they will be one of the main achievements by the end of MIKELANGELO project.

As prototype I has less implementation effort, it is planned as a starting point for the first year of the MIKELANGELO project, and its performance measurements will be used for comparison with the other two prototypes later on. Several new interfaces have to be implemented for integrating virtio-rdma on the host side (i.e. via QEMU). Table 1 lists the necessary interfaces that will be used to initialize and finalize the virtio-rdma for the guest OS.

virtio_rdma_start	virtio_rdma_validate_features
virtio_rdma_reset	virtio_rdma_get_config
virtio_rdma_instance_init	virtio_rdma_set_config
virtio_rdma_class_init	virtio_rdma_set_status
virtio_register_types	virtio_rdma_guest_notifier_pending
virtio_rdma_get_features	virtio_rdma_guest_notifier_mask
virtio_rdma_bad_features	virtio_rdma_load
virtio_rdma_set_features	virtio_rdma_save

Table 1: Basic virtio-rdma interfaces for Prototype I.

6 Monitoring OSv

OSv images can optionally include an "httpserver" module which can be used to enable remote monitoring of an OSv VM. "httpserver" is a small and simple HTTP server that runs in a thread, and implements a REST API, i.e., an HTTP-based API, for remote inquiries or control of the running guest. The reply of each of these HTTP requests is in the JSON format.

The complete REST API is described below, but two requests are particularly useful for monitoring a running guest:

1. "**/os/threads**" returns the list of threads on the system, and some information and statistics on each thread. This includes each thread's numerical id and string name, the CPU number on which it last ran, the total amount of CPU time this thread has used, the number of context switches and preemptions it underwent, and the number of times it migrated between CPUs.
   The OSv distribution includes a script, `scripts/top.py`, which uses this API to let a user get "top"-like output for a remote OSv guest: It makes a "`/os/threads`" request every few seconds, and subtracts the total amount of CPU time used by each thread in this and the previous iteration; The result is the percentage of CPU used by each thread, which we can now sort and show the top CPU-using threads (like in Linux's "top"), and some statistics on each (e.g., similar subtraction and division can give us the number of context switches per second for each of those threads).
2. "**/trace/count**" enables counting of a specific tracepoint, or returns the counts of all enabled tracepoints.
   OSv's tracepoints are a powerful debugging and statistics mechanism, inspired by a similar feature in Linux and Solaris: In many places in OSv's source code, a "trace" call is embedded. For example, we might have a "`memory_malloc`" trace in the beginning of the `malloc()` function, or "`sched_switch`" trace when doing a context switch. Normally, this trace doesn't do anything - it appears in the executable as a 5-byte "NOP" (do-nothing) instruction and has almost immeasurable impact on the speed of the run. When we want to enable counting of a specific tracepoint, e.g., count the number of sched_switch events, we replace these NOPs by a jump to a small piece of code which increments a per-cpu counter. Because the counter is per-cpu, and has no atomic-operation overhead (and moreover, usually resides in the CPU's cache), counting can be enabled even for extremely frequent tracepoints occurring millions of times each second (e.g., "`memory_malloc`") - with a hardly noticeable performance degradation of the workload. Only when we actually query the counter, do we need to add these per-cpu values to get the total one. The OSv distribution includes a script, `scripts/freq.py`, which uses this API to

enable one or more counters, to retrieve their counts every few seconds, and display the frequency of the event (subtraction of count at two different times, divided by the time interval's length). This script makes it very convenient to see, for example, the total number of context switches per second while the workload is running, and how it relates, for example, to the frequency of `mutex_lock_wait`, and so on. The list of tracepoints supported by OSv at the time of this writing includes over 300 different tracepoints, and is shown here:

access_scanner
add_read_mapping
app_destroy
app_join
app_join_ret
app_main
app_main_ret
app_request_termination
app_request_termination_ret
app_termination_callback_added
app_termination_callback_fired
async_timer_task_cancel
async_timer_task_create
async_timer_task_destroy
async_timer_task_fire
async_timer_task_insert
async_timer_task_misfire
async_timer_task_remove
async_timer_task_reschedule
async_timer_task_shutdown
async_worker_fire
async_worker_fire_ret
async_worker_started
async_worker_timer_fire
async_worker_timer_fire_ret
callout_init
callout_reset
callout_stop
callout_stop_wait
callout_thread_dispatching
callout_thread_retry
callout_thread_waiting
callout_thread_waking
clear_pte
condvar_wait
condvar_wake_all
condvar_wake_one
drop_read_cached_page
drop_write_cached_page
elf_load
elf_lookup
elf_lookup_addr
elf_unload
epoll_create
epoll_ctl
epoll_ready
epoll_wait
function entry
function exit
pool_alloc
pool_free
pool_free_different_cpu
pool_free_same_cpu
remove_mapping
sampler_tick
sched_idle
sched_idle_ret
sched_ipi
sched_load
sched_migrate
sched_preempt
sched_pull
sched_queue
sched_sched
sched_switch
sched_wait
sched_wait_ret
sched_wake
sched_yield
sched_yield_switch
synch_msleep
synch_msleep_expired
synch_msleep_wait
synch_wakeup
synch_wakeup_one
synch_wakeup_one_waking
synch_wakeup_waking
tcp_input_ack
tcp_output
tcp_output_cant_take_inp_lock
tcp_output_error
tcp_output_just_ret
tcp_output_resched_end
tcp_output_resched_start
tcp_output_ret
tcp_output_start
tcp_state
tcp_timer_tso_flush
tcp_timer_tso_flush_err
tcp_timer_tso_flush_ret
thread_create
timer_cancel
timer_fired
timer_reset
timer_set
tso_flush_cancel
tso_flush_fire
tso_flush_sched
vfs_isatty
vfs_isatty_err
vfs_isatty_ret
vfs_link
vfs_link_err
vfs_link_ret
vfs_lseek
vfs_lseek_err
vfs_lseek_ret
vfs_lstat
vfs_lstat_err
vfs_lstat_ret
vfs_mkdir
vfs_mkdir_err
vfs_mkdir_ret
vfs_mknod
vfs_mknod_err
vfs_mknod_ret
vfs_open
vfs_open_err
vfs_open_ret
vfs_pread
vfs_pread_err
vfs_pread_ret
vfs_pwrite
vfs_pwrite_err
vfs_pwrite_ret
vfs_pwritev
vfs_pwritev_err
vfs_pwritev_ret
vfs_readdir
vfs_readdir_err
vfs_readdir_ret
vfs_rename
vfs_rename_err
vfs_rename_ret
vfs_rmdir
vfs_rmdir_err
vfs_rmdir_ret
vfs_stat
vfs_stat_err
vfs_stat_ret
vfs_statfs
vfs_statfs_err
vfs_symlink
vfs_symlink_err
vfs_symlink_ret
vfs_truncate
vfs_truncate_err

```
in_lltable_lookup
in_lltable_lookup_fast
inpcb_free
inpcb_ref
inpcb_rele
jvm_balloon_close
jvm_balloon_fault
jvm_balloon_free
jvm_balloon_move
jvm_balloon_new
map_arc_buf
memory_free
memory_huge_failure
memory_malloc
memory_malloc_large
memory_malloc_mempool
memory_malloc_page
memory_mmap
memory_mmap_err
memory_mmap_ret
memory_munmap
memory_munmap_err
memory_munmap_ret
memory_page_alloc
memory_page_free
memory_realloc
memory_reclaim
memory_wait
mmu_vm_fault
mmu_vm_fault_ret
mmu_vm_fault_sigsegv
msix_interrupt
msix_migrate
mutex_lock
mutex_lock_wait
mutex_lock_wake
mutex_receive_lock
mutex_send_lock
mutex_try_lock
mutex_unlock
net_packet_handling
net_packet_in
net_packet_out
pcpu_worker_item_end_wait
pcpu_worker_item_invoke
pcpu_worker_item_set_finished
pcpu_worker_item_signal
pcpu_worker_item_wait
pcpu_worker_sheriff_started
poll
poll_drain
poll_err
poll_ret
poll_wake
unmap_arc_buf
vfs_access
vfs_access_err
vfs_access_ret
vfs_chdir
vfs_chdir_err
vfs_chdir_ret
vfs_chmod
vfs_chmod_err
vfs_chmod_ret
vfs_close
vfs_close_err
vfs_close_ret
vfs_dup
vfs_dup3
vfs_dup3_err
vfs_dup3_ret
vfs_dup_err
vfs_dup_ret
vfs_fallocate
vfs_fallocate_err
vfs_fallocate_ret
vfs_fchdir
vfs_fchdir_err
vfs_fchdir_ret
vfs_fchmod
vfs_fchmod_ret
vfs_fchown
vfs_fchown_ret
vfs_fcntl
vfs_fcntl_err
vfs_fcntl_ret
vfs_fstat
vfs_fstat_err
vfs_fstat_ret
vfs_fstatfs
vfs_fstatfs_err
vfs_fstatfs_ret
vfs_fsync
vfs_fsync_err
vfs_fsync_ret
vfs_ftruncate
vfs_ftruncate_err
vfs_ftruncate_ret
vfs_futimens
vfs_futimens_err
vfs_futimens_ret
vfs_getcwd
vfs_getcwd_err
vfs_getcwd_ret
vfs_ioctl
vfs_ioctl_err
vfs_ioctl_ret
vfs_truncate_ret
vfs_unlink
vfs_unlink_err
vfs_unlink_ret
vfs_utimensat
vfs_utimensat_err
vfs_utimensat_ret
vfs_utimes
vfs_utimes_err
vfs_utimes_ret
virtio_add_buf
virtio_blk_make_request_readonly
virtio_blk_make_request_seg_max
virtio_blk_read_config_blk_size
virtio_blk_read_config_capacity
virtio_blk_read_config_geometry
virtio_blk_read_config_ro
virtio_blk_read_config_seg_max
virtio_blk_read_config_topology
virtio_blk_read_config_wce
virtio_blk_req_err
virtio_blk_req_ok
virtio_blk_req_unsupp
virtio_blk_strategy
virtio_blk_wake
virtio_disable_interrupts
virtio_enable_interrupts
virtio_kicked_event_idx
virtio_net_fill_rx_ring
virtio_net_fill_rx_ring_added
virtio_net_rx_packet
virtio_net_rx_wake
virtio_net_tx_failed_add_buf
virtio_net_tx_no_space_calling_gc
virtio_net_tx_packet
virtio_net_tx_packet_size
virtio_net_tx_xmit_one_failed_to_pos
t
virtio_wait_for_queue
vring_get_buf_elem
vring_get_buf_finalize
vring_get_buf_gc
vring_get_buf_ret
vring_update_used_event
waitqueue_wait
waitqueue_wake_all
waitqueue_wake_one
xen_irq
xen_irq_exec
xen_irq_exec_ret
xen_irq_ret
```

Beyond these two useful REST API requests, OSv supports many more requests, overviewed here. Note that this overview omits a lot of important information, such as the parameters that each request takes, or the type of its return value. For the full information, please refer to the modules/httpserver/api-doc/listings directory in OSv's source distribution [28]. OSv also optionally provides a "swagger" GUI to let a user use these requests by filling a form, instead of remembering the request's URL and parameters.

- `/api/batch`: Perform batch API calls in a single command. Commands are performed sequentially and independently. Each command has its own response code.
- `/api/stop`: Stopping the API server causing it to terminate. If the API server runs as the main application, it would cause the system to terminate.
- `/app`: Run an application with its command line parameters.
- `/env`: List environment variables, return the value of a specific environment variable, or modify or delete one - depending if the HTTP method used is GET, POST, or DELETE respectively.
- `/file`: Return information about an existing file or directory, delete one, create one, rename one, or upload one.
- `/fs/df`: Report filesystem usage of one mount point or all of them.
- `/hardware/processor/flags`: List all present processor features.
- `/hardware/firmware/vendor`
- `/hardware/hypervisor`: Returns name of the hypervisor OSv is running on.
- `/hardware/processor/count`
- `/network/ifconfig`: Get a list of all the interfaces configuration and data.
- `/network/route`
- `/os/name`
- `/os/version`
- `/os/vendor`
- `/os/uptime`: Returns the number of seconds since the system was booted.
- `/os/date`: Returns the current date and time.
- `/os/memory/total`: Returns total amount of memory usable by the system (in bytes).
- `/os/memory/free`: Returns the amount of free memory in the system (in bytes).
- `/os/poweroff`
- `/os/shutdown`
- `/os/reboot`
- `/os/dmesg`: Returns the operating system boot log.
- `/os/hostname`: Get or set the host's name.
- `/os/cmdline`: Get or set the image's default command line.
- `/trace/status`, `/trace/event`, `/trace/count`, `/trace/sampler`, `/trace/buffers`: Enable, disable and query tracepoints.

The full-stack MIKELANGELO Instrumentation and Monitoring system has been designed with a flexible plugin architecture. An OSv monitoring plugin is being developed that will be able to retrieve all useful data from an OSv guest using the OSv REST API described. Indeed, the OSv monitoring plugin may gather data from multiple OSv instances. The available monitoring data, including thread and tracepoint data as well as hardware configuration, can be discovered at runtime and only the specific data of interest captured, processed and published to the monitoring back-end.

7 Evaluation Baseline

The new guest operating system components described in this document - OSv, Seastar and virtio-rdma - are poised to improve the performance of various cloud and HPC workloads, including the MIKELANGELO use cases [3, 4, 5, 6]. However, each of these complete workloads is complex and difficult to continuously benchmark as part of the ongoing development effort of the guest operating system. For this purpose, we want to choose several simple benchmark which each is simple to install and test repeatedly, and each stresses one particular area that the new guest OS is supposed to improve.

In the following subsections, we chose three benchmarks that we will strive to improve throughout the project:

1. One simple benchmark which represents an HPC workload which uses RDMA (*NetPIPE*).
2. Two simple benchmarks which represents an I/O-intensive TCP/IP server (*Netperf* and *Memcached*).

The first benchmark will primarily demonstrate the improvements provided by RDMA-verbs virtualization in virtio-rdma. The other benchmarks will demonstrate the improvements from three different components of the guest operating system: from virtio-rdma's support for unmodified socket applications, from OSv's faster kernel and network stack, and from rewriting an application to Seastar.

The baseline of both benchmark will be running the unmodified benchmark application on the Linux guest operating system.

7.1 Benchmarking virtio-rdma

7.1.1 Benchmarks Used

The following is the list of benchmarks packages that will be used to measure performance of RDMA-related components of MIKELANGELO. The list will be revised throughout the MIKELANGELO project.

- NetPerf [10], a benchmark that can be used to measure various aspect of networking performance, e.g. latency, bandwidth on TCP or UDP. It will help get the communication performance of inter-VM communication using shared memory and RDMA virtualization solutions.
- NetPIPE [11], a protocol independent network performance evaluator. It performs ping-ping test between two processes either over network or SMP with increasing

message sizes. This benchmark has full support of MPI-2 application, thus is commonly used to evaluate performance of a HPC environment.

7.1.2 Testbed

The testbed system consists of two servers, each running 2 virtual machines. These two virtual machines are able to communicate through Ethernet (Intel I217-LM e1000) or InfiniBand (Mellanox ConnectX-3 FDR) network interconnects between the hosts or through shared memory inside the host.

The server is a HP workstation with a 4-core single node (Intel Core i5-4590 3,30GHz) processor, and 4GB RAM memory with the host OS being Ubuntu 14.04 LTS.

The host hypervisor is KVM (3.18.0)/QEMU (2.3.0), and the VMs run guest OS the Ubuntu 14.04.2 server edition, each configured with four vCPU (assigned with four cores on the host) and 2GB of memory.

An additional HPC testbed, provided by HLRS, is described in D2.19 "The first MIKELANGELO architecture" [1]. It will be used to further evaluate the performance gains.

7.1.3 Experimental Methodology

- For inter-VM communication on different hosts, we run single VM on 2 to 16 hosts. The hosts are connected via Ethernet and InfiniBand network. Then we compare the network performance in following network modes:
  - TCP,
  - Ethernet over InfiniBand,
  - IP over InfiniBand,
  - RoCE,
  - InfiniBand.
- For inter-VM communication on the same host, we run 2 to 8 VMs.

7.2 TCP/IP Benchmark

The purpose of this benchmark is to evaluate the performance improvement that MIKELANGELO as a whole, and its various separate components, can bring to TCP-based applications such as network servers. Such applications could benefit from virtio-rdma's speed-ups for unmodified socket applications (including special support for communicating VMs on the same host), from OSv's faster kernel and network stack, and from rewriting an application to Seastar.

- Netperf [10], already mentioned above, can measure some aspects of the network performance of unmodified applications using TCP through the socket API. Netperf measures the maximum TCP throughput in long streams ("TCP_STREAM" benchmark) as well as latency ("TCP_RR").
- Memcached [8] is a popular cloud application used for caching of frequently requested objects and lowering the load on slower database servers. With short requests and responses, which are typical, memcached's load is different from Netperf above in that it emphasizes not one long TCP stream, but rather processing a huge number of separate connections from many concurrent clients. The memcached server will be loaded by the memaslap [12] load generator and benchmark tool, which can simulate a configurable memcached workload from many clients, and measure the resulting throughput (transactions per second).

8 Key Takeaways

- This deliverable describes the architecture of the guest operating system - the operating system running on each VM in the MIKELANGELO cloud.
- This architecture improves the performance of existing cloud and HPC applications, originally written for Linux, with two major components, OSv and vRDMA:
    - **OSv** is a new operating system designed specifically for running a single application in a cloud VM.
      OSv is limited to a single application, with a single process but potentially many threads. OSv can run existing Linux executables if they are relocatable (i.e., a shared object (".so") or a PIE), and also use existing Linux shared libraries. Other executables will need to be re-compiled from source to be relocatable.
    - **vRDMA** speeds up the communication between different guests which communicate with the traditional socket API or with the RDMA verbs API:
        - For applications using the socket API, vRDMA transparently replaces the slow IP-based communication with more efficient mechanisms: RDMA (Remote Direct Memory Access) when the guests are on different hosts, or shared memory when the guests are co-located on the same host.
        - For applications which use the RDMA verbs API, such as those that use MPI 2's one-sided communication or MPI 3's RMA, vRDMA provides an efficient implementation of these RDMA verbs, again over RDMA or shared memory, as appropriate.
- OSv also improves the agility of the application deployment, by reducing boot time and image size, by offering HTTP-based monitoring and control, and by proposing a new workflow for composing these application images on-the-fly from pre-compiled components using the MIKELANGELO Package Manager.
- Additionally, a new API, "Seastar", is proposed for new I/O-intensive asynchronous applications such as network servers. Seastar can significantly improve the performance of such applications over existing applications which use traditional APIs like sockets and threads.
    - Seastar applications are more efficient and more scalable (to many-core and many-socket machines) because they avoid slow and unscalable locks and atomic operations, and instead use a sharded (share-nothing) design.
    - Seastar uses a future-and-continuations programming model to allow writing complex asynchronous applications, not just simple packet processors.
- We then introduced the monitoring capabilities built directly into OSv. These provide a thorough insight into the behaviour of the application and the kernel itself and are

well suited for the overarching MIKELANGELO monitoring facility integrated as part of work package 5.

- We conclude simple benchmarks that are going to be used initially to measure the performance improvements of MIKELANGELO’s guest operating system, compared to the baseline operating system Linux. These benchmarks measure the performance of TCP-based network applications, as well as RDMA-verbs-based HPC applications.

9 References and Applicable Documents

[1] "The first MIKELANGELO architecture", Deliverable D2.19 of the MIKELANGELO project (August, 2015).
[2] "The first sKVM hypervisor architecture", Deliverable D2.13 of the MIKELANGELO project (August, 2015).
[3] "First Cancellous bone simulation Use Case Implementation strategy", Deliverable 2.1 of the MIKELANGELO project (August, 2015).
[4] "First Cloud-Bursting Use Case Implementation Strategy", Deliverable 2.4 of the MIKELANGELO project (August, 2015).
[5] "The First Virtualised Big Data Use Case Implementation Strategy", Deliverable 2.7 of the MIKELANGELO project (August, 2015).
[6] "The First Aerodynamic Map Use Case Implementation Strategy", Deliverable 2.10 of the MIKELANGELO project (August, 2015).
[7] "OSv - Optimizing the Operating System for Virtual Machines", A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, V. Zolotarov. 2014 USENIX Annual Technical Conference (USENIX ATC 14) 1, 61-72
[8] "memcached - a distributed memory object caching system", http://memcached.org/
[9] "DPDK, the Data Plane Development Kit", http://dpdk.org/
[10] "Netperf", http://www.netperf.org/netperf/
[11] "NetPIPE - a Network Protocol Independent Performance Evaluator", http://bitspjoule.org/netpipe/
[12] "Memaslap - a load generation and benchmark tool for memcached servers", http://docs.libmemcached.org/bin/memaslap.html
[13] "Speeding up Networking", Van Jacobson and Bob Felderman, Linux.conf.au 2006, http://www.lemis.com/grog/Documentation/vj/lca06vj.pdf
[14] The Apache Cassandra project, http://cassandra.apache.org/
[15] OSv Applications, https://github.com/cloudius-systems/osv-apps
[16] OSv scripts for building images, https://github.com/cloudius-systems/osv/blob/master/scripts/build
[17] Capstan, http://osv.io/capstan/
[18] Docker, https://www.docker.com
[19] Capstanfile documentation, https://github.com/cloudius-systems/capstan/blob/master/Documentation/Capstanfile.md
[20] Debian packages, https://www.debian.org/doc/manuals/debian-faq/ch-pkg_basics.en.html
[21] Homebrew, http://brew.sh
[22] Example homebrew forumula https://github.com/Homebrew/homebrew/blob/master/Library/Formula/polarssl.rb
[23] Snappy Ubuntu, https://developer.ubuntu.com/en/snappy/tutorials/build-snaps/
[24] Ubuntu Core, https://developer.ubuntu.com/en/snappy/

[25] Example of Ubuntu Snappy application, http://bazaar.launchpad.net/~snappy-dev/snappy-hub/snappy-examples/files/head:/python-xkcd-webserver/

[26] Docker Image Specification, https://github.com/docker/docker/blob/master/image/spec/v1.md

[27] Open Container Initiative, https://www.opencontainers.org

[28] OSv HTTP Server API Listing, https://github.com/cloudius-systems/osv/tree/master/modules/httpserver/api-doc/listings

[29] Musl C library, http://www.musl-libc.org/

[30] Gordon, A. W., & Lu, P. (2011). Low-Latency Caching for Cloud-Based Web Applications. Work, 456, 2–8.

[31] Libraries, A. (2011). Shared-Memory Optimizations for Virtual Machines.