MIKELANGELO

D2.21

The final MIKELANGELO architecture

Workpackage	2	Use Case & Architecture Analysis	
Author(s)	Nico Struckmann		USTUTT
	Nadav Har'El		SCYLLA
	Fang Chen, Shiqing Fan		HUAWEI
	Justin Činkelj, Gregor Berginc		XLAB
	Peter Chronz		GWDG
	Niv Gilboa, Gabriel Scalosub		BGU
	Kalman Meth		IBM
	John Kennedy		INTEL
Reviewer	Holm Rauchfuss		HUAWEI
Reviewer	Gregor Berginc		XLAB
Dissemination Level	PU		

Date	Author	Comments	Version	Status
2017-05-23	N.Struckmann	Initial draft	V0.0	Draft
2017-06-01	N.Har'El	Additions for guest OS	V0.1	Draft
2017-06-02	G.Berginc, J.Činkelj	LEET, UNCLOT and generic guest OS sections	V0.2	Draft
2017-06-08	N.Gilboa, G.Scalosub	SCAM Section added	V0.3	Draft

2017-06-09	S.Fan, F.Chen	vRDMA Section added	V0.4	Draft
2017-06-15	K.Meth	ZeCoRx Section added	v0.5	Draft
2017-06-16	J.Kennedy	Snap Section added	V0.6	Draft
2017-06-22	N.Struckmann	HPC section	V0.7	Draft
2017-06-23	P.Chronz	Cloud section	V0.8	Draft
2017-06-26	N.Struckmann	Document ready for review	V1.0	Review
2017-06-29	G.Berginc	Document ready for submission	V2.0	Final

Executive Summary

This report details the final architecture of the MIKELANGELO technology stack. It is comprised of a high-level overview, followed by a thorough presentation of individual components as well as their integrations into common software stacks.

MIKELANGELO project is about improved performance, flexibility and manageability of virtualised infrastructures. The project deals with a wide range of technologies on many different levels of a typical stack, such as cloud and HPC. These involve the hypervisor, the guest operating system and the management layers. On top of that, this holistic approach puts MIKELANGELO into a unique position with the capacity of providing several cross-cutting technologies extending the potential of individual components.

The hypervisor is improved by two components. The IO core manager dynamically balances the allocation of CPU cores dedicated to I/O operations and computation. On the other hand, Zero-Copy Receive proposes to remove unnecessary data transfers found in the existing networking stack of the KVM hypervisor. The guest operating system enhancements introduce significantly improved functional compatibility of the lightweight operating system OSv with Linux and Posix standards. It is also providing a novel programming framework (Seastar) targeting modern asynchronous programming models focusing on scalable computing. Furthermore, MIKELANGELO tackles the cumbersome problem of packaging and managing lightweight virtual machines with a complete toolbox (Lightweight Execution Environment Toolbox, LEET) greatly simplifying the use of application packages and their deployment into various deployments.

On top of these individual improvements, MIKELANGELO provides several cross-level components. The virtual RDMA (vRDMA) promises to provide a paravirtual driver into the virtual machine enabling direct access to the underlying interconnects, such as Infiniband, with very low overhead. UNikernel Cross Level cOmmunication opTimisation (UNCLOT) on the other hand removes entire networking stack of the guest OS offering optimal intra-host communication channel for collocated virtual machines.

The telemetry framework (snap) goes even further by providing invaluable metrics from all the layers of the proposed architecture. All these metrics are used for the evaluation of project results as well as in some high-level management components. Similarly, Side-Channel Attack Monitoring and Mitigation (SCAM) component strengthens the trust of the virtual infrastructure detecting and preventing malicious virtual machines trying to exploit shared hardware to collect sensitive information from colocated virtual machines.

Finally, to demonstrate capabilities of individual components as well as extend the exploitation potential of the MIKELANGELO project, two overarching integrations for Cloud

and HPC, are supplied. vTorque extends the widely used HPC batch-system PBS/Torque and provides the baseline for the integration of several MIKELANGELO components. It comes with management capabilities by the creation of virtual environments and the deployment of batch workloads in such. While MCM and Scotty enhance OpenStack by new scheduling capabilities and continuous integration support, all MIKELANGELO components can also be deployed independently.

Acknowledgement

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

Table of contents

Table of Figures

1 Introduction

This document describes the current status of the final MIKELANGELO architecture as well as the remaining work on the whole software stack and provides complete overview of all components involved and where they are located in the different layers the MIKELANGELO framework. The report details all of its core components, ranging from an enhanced hypervisor and lightweight cloud operating system to cross-layer security, IO optimisations, telemetry and CI components. An approach for the integration into deployment and orchestration frameworks is presented, based on two complementing environments, Cloud and HPC, blurring the boundaries between them. HPC workload manager (PBS/Torque) and a cloud management middleware (OpenStack) are used as industry leaders releasing the potential of individual components. This report accompanies the upcoming public release of the MIKELANGELO technologies and offers some of the more detailed insights into underlying components. Each chapter in this document consists of high-level description of the final architectures addressing the remaining requirements for the various components. These requirements are either internal or external. Internal requirements are those that are identified by the component owner and are part of the initial design (for example, implementation of OSv image composition). External requirements are related to either use case or integration.

The report starts with an overview of the whole stack in this section, succeeded by details for all components. It also highlights the complete integration into the two distinct infrastructures and is finished with a section summarising concluding remarks, outlining the main purpose of the presented architecture design.

The MIKELANGELO architecture is designed to improve the I/O performance in virtualised environments and also to bring the benefits of flexibility through virtualization to HPC systems. These benefits include, besides application packaging and application deployment, also elasticity during the application execution, without losing the high performance in computation and communication provided by HPC infrastructures. Each of the components can be used independently of other, however the benefits sum up combining as much components as possible.

The main advantage of the MIKELANGELO project is that it addresses the whole stack comprising typical infrastructure as a service (IaaS) or, with the support of some of the use cases, even platform as a service (PaaS). It's core development lies in the enhancements made to the KVM hypervisor and improvements, both functional and non-functional, made to a library guest operating system. Both of these components are transparently integrated into commonly used middlewares, PBS/Torque for HPC-like environments and OpenStack for Cloud. Several cross-level components, such as **vRDMA** and **UNCLOT** for improved I/O

performance, snap for holistic telemetry and **SCAM** for greater security improve the overall performance, flexibility and manageability of the two offerings. The diagram in Figure 1 shows a high level overview of these components and their relations, which are further introduced in the remainder of this section.

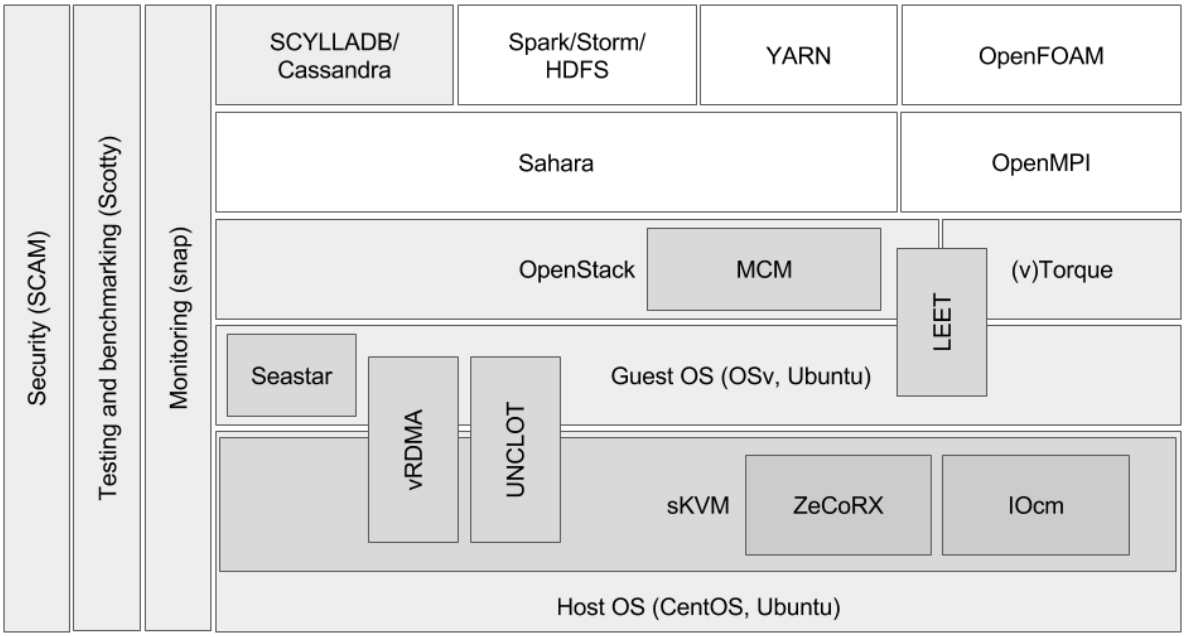

Figure 1: High-level Cloud and HPC-Cloud architecture.

The Hypervisor Architecture, **sKVM**, aims at improving performance and security at the lower layer of the architecture, the hypervisor. Previous work was divided into three separate components: IOcm, an optimization for virtual I/O devices that uses dedicated I/O cores, virtual RDMA (vRDMA) for low overhead communication between virtual machines, and SCAM, a security feature identifying and preventing cache side-channel attacks from malicious co-located virtual machines. Work on IOcm was completed in the second year of the project and its architecture is described in a previous version of this document (D2.20). Updates to vRDMA and SCAM appear in separate sections later in this document. This report also extends the details about the architecture of Zero-copy Receive (**ZeCoRx**) that has initially been introduced in report D3.2. ZeCoRx avoids copying data between host and guest buffers on the receive path.

The guest operating system (or "guest OS") is the operating system running inside each individual VM (virtual machine) of the MIKELANGELO cloud. The guest OS implements the various APIs (Application Programming Interfaces) and ABIs (Application Binary Interfaces) which the applications use. The goals of enhancing the guest operating system within the MIKELANGELO project are to make it easier to run I/O-heavy and HPC (high performance

computing) applications in the cloud, and additionally to run such applications more efficiently than on traditional operating systems.

OSv is an open-source project which was started prior to the MIKELANGELO project by one of the MIKELANGELO partners (ScyllaDB, formerly Cloudius Systems). While OSv was more mature than alternative unikernels, it still needed significant improvements to become useful for MIKELANGELO. The work in WP2 of analyzing OSv's architecture and the various use cases which we intend to run on it, resulted in a set of required improvements to OSv which are being advanced within WP4. This continued development of OSv for MIKELANGELO considers performance, usability and application compatibility, we dedicate a chapter below for more details on the resulting architecture of OSv.

While OSv allows running existing Linux applications, we realized that certain Linux APIs, including the socket API, and certain programming habits, make applications which use them inefficient on modern multi-core hardware. OSv can improve the performance of such applications to some degree, but rewriting the application to use new non-Linux APIs can bring significantly better performance. So in MIKELANGELO we also introduced a new API, called **Seastar**, for writing new highly-efficient asynchronous server applications. One of the four use cases we analyze and implement in WP2 and WP6, the "Cloud Bursting" use case, can particularly benefit from Seastar; This use case is based on the Cassandra distributed database, and reimplementing Cassandra with Seastar resulted in as much as 10-fold performance improvement over regular Cassandra. Seastar, and also the Cassandra reimplementation using Seastar (called "Scylla"), are also released as open source projects. The improvements we are making to Seastar as part of the MIKELANGELO project include improving Seastar's architecture and implementation to make it faster and more useful for MIKELANGELO's specific use case (Cloud Bursting), but also making Seastar more generally useful for more potential MIKELANGELO users by improving Seastar's design, features, and documentation. We will describe the architecture of Seastar in a separate chapter below.

Beyond the benefits of modifying just the guest OS, we can also benefit from possible synergy with the hypervisor, which MIKELANGELO also modifies (as described in the previous section): We can get additional performance benefits from modifying both layers in a cooperative manner. The two cross-layer (guest OS and hypervisor) improvements we are implementing in MIKELANGELO are virtual RDMA (virtualized Remote Direct Memory Access) and UNCLOT (UNikernel Cross Level cOmmunication opTimisation), both already introduced above and have separate sections devoted to them below.

In addition to improving efficiency, another goal of the MIKELANGELO project is to simplify deployment of applications in the cloud. MIKELANGELO initially focused only on application package management with its extensions done primarily to the Capstan open source project.

MIKELANGELO has introduced a completely new way of composing self-sufficient virtual machine images based on OSv. This allows for flexible and efficient reuse of pre-built application packages that are readily provided by the MIKELANGELO consortium. It also builds on best practices on how to approach the package preparation with various packages from the HPC and Big data fields. Since then, MIKELANGELO progressed towards cloud management and application orchestration. This includes the integration of full support for deployment of unikernels onto OpenStack. Application orchestration using container-like interfaces is the last step towards management of lightweight applications on top of heterogeneous infrastructures. All of this has been joined under one main toolbox (**LEET** - Lightweight Execution Environment Toolbox), introduced with Milestone MS5 serving both dissemination and exploitation activities. A separate section will be devoted to LEET below.

On top of these components, MIKELANGELO also delivers components spanning over all the layers of the software stack. This includes snap and SCAM. Snap is a holistic telemetry solution aimed at gathering data from all the layers of the target infrastructures, ranging from the hardware through all layers of the infrastructure software to the applications themselves. Snap furthermore provides flexible telemetry data processing capabilities and a wide range of storage backends. All this provides valuable in-depth insights into running applications by meaningful live statistics.

SCAM on the other hand is a sophisticated security module located at the hypervisor layer. It is able to detect cache side channel attacks from virtual machines trying to steal sensitive information from colocated VMs. The project is demonstrating a particular use case of stealing a private key used for encryption of SSL communication, along with techniques to reliably monitor such attacks and handle them dynamically, i.e. preventing the attacker from stealing the desired information.

Finally, MIKELANGELO uses two commonly used deployment targets: Cloud and HPC. The purpose of integrating the aforementioned components into a common environments is to demonstrate the potential of the individual components, as well as steer the dissemination and exploitation activities. Cloud integration presents the architectural design and gives details on how the integration with OpenStack is achieved. There is the updated CI integration component **Scotty** and also a new component dedicated to live re-scheduling of resources in Clouds, called **MCM.** A tool called **OSmod** that is utilized to modify the host operating system**,** is also introduced.

HPC Integration focuses on the infrastructure and management layer of HPC batch systems. The section presents the integration of all MIKELANGELO components eligible for HPC environments into the widely-spread batch system management software Torque, under the name of **vTorque**. It extends PBS/Torque by virtualization capabilities to handle VM

instantiation, job deployment and VM tear down within the Torque job life cycle in a transparent way to the user.

2 Zero Copy Receive

IOcm (IO Core Manager) was designed to improve IO performance of network traffic from VMs by isolating IO processing on a separate set of cpu cores. The IOcm solution is described in document D2.20, sections 2.2 and in document D3.1 chapter 2. After experimentation, we saw that it improves IO performance most significantly for small packets. For large packets it showed negligible improvement. We observed that our use cases tend to favor large packets, so we did not sufficiently benefit from the performance improvements of IOcm. As a result, we saw the need to develop a new solution to better handle large packets.

In the KVM hypervisor, incoming packets from the network must pass through several objects in the Linux kernel before being delivered to the guest VM. Currently, both the hypervisor and the guest keep their own sets of buffers on the receive path. For large packets, the overall processing time is dominated by the copying of data from hypervisor buffers to guest buffers. The cost of the internal data copy will continue to dominate as external bus speeds improve and approach that of the internal bus.

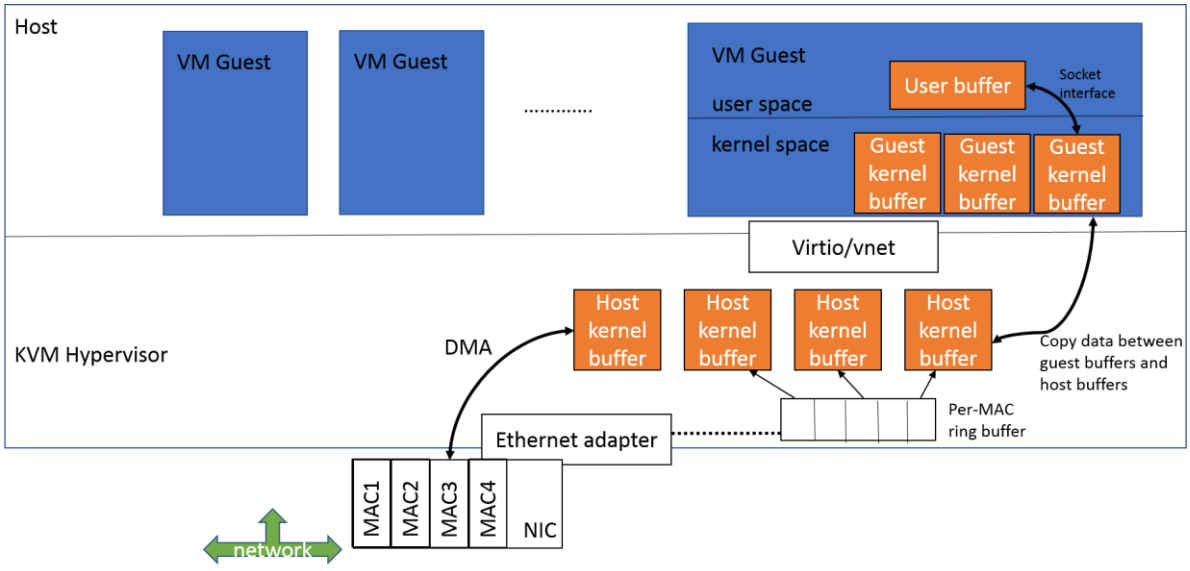

Figure 2. Existing Linux virtio architecture.

Some Linux network drivers support Zero-Copy on Transmit (Tx) messages. Zero-copy Tx avoids the copy of data between VM guest kernel buffers and host kernel buffers, thus improving Tx latency. Buffers in a VM guest kernel for a virtualized NIC are passed through the host device drivers and DMA-d directly to the network adapter, without an additional copy into host memory buffers. Since the Tx data from the VM guest is always in-hand, it is quite straight-forward to map the buffer for DMA and to pass the data down the stack to the network adapter driver.

Zero-Copy for Receive (Rx) messages is not yet supported in Linux. A number of significant obstacles must be overcome in order to support Zero-Copy for Rx. Buffers must be prepared to receive data arriving from the network. Currently, DMA buffers are allocated by the low-level network adapter driver. The data is then passed up the stack to be consumed. When Rx data arrives, it is not necessarily clear a-priori for whom the data is designated. The data may eventually be copied to VM guest kernel buffers. The challenge is to allow the use of VM guest kernel buffers as DMA buffers, at least when we know that the VM guest is the sole consumer of a particular stream of data.

One solution that has been tried is page-flipping, in which the page with the received data is mapped into the memory of the target host after the data has already been placed in the buffer. The overhead to perform the page mapping (including cache clearing) is significant, and essentially negates the benefit we wanted to achieve by avoiding the copy (Ronciak, 2004)[1].

Our proposed solution requires the introduction of several interfaces that enable us to communicate between the high and low-level drivers to pass buffers down and up the stack, when needed. We also need to deal with the case when insufficient VM guest buffers have been made available to receive data from the network.

Our initial design and implementation concentrate on an end-to-end solution for a single example, but the method is applicable to the general case where a distinct MAC address is identified with a guest VM network interface. We define a VM guest with a macvtap[2] device. On the guest side, the data goes through virtio device. On the host, the data goes through vhost_net, macvtap, macvlan, and the Ethernet adapter device drivers. We describe the proposed changes to these drivers in order to support Zero-Copy Rx.

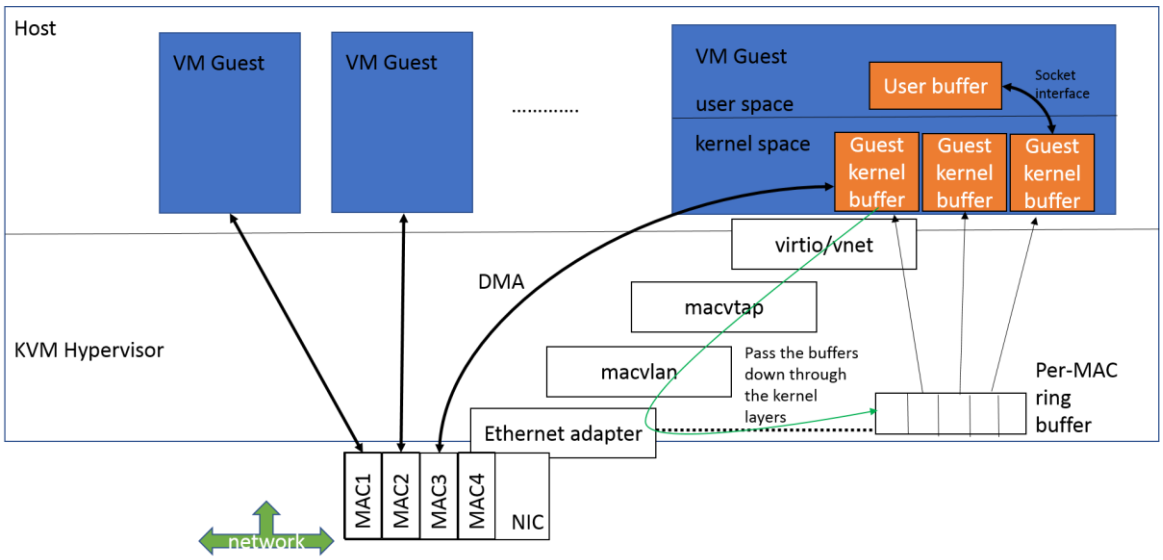

Figure 3. Proposed Zero-Copy Rx architecture.

2.1 Network Device Driver

Each type of driver has a set of operations defined. Typically, only some of the operations are specified, while the others use the default action. The lowest level driver that concerns us is for a network device, corresponding to a physical network adapter. The kernel include file `include/linux/netdevice.h` defines all the possible operations for such a device.

The `netdevice.h` file contains the definition of the structures net_device and net_device_ops. When a device driver is called, it is provided with the net_device structure for the relevant device. (There is usually a separate net_device structure for each instance of a particular type of device.) The net_device structure contains a pointer to its net_device_ops structure, which contains pointers to all the exported functions of the device driver. These functions are available to be called by any other kernel code that obtains a pointer to the net_device structure of the device. An SKB (socket buffer structure) contains a pointer to the net_device structure for the device that is supposed to handle it. This field is set by the function that allocates and then forwards the SKB for processing.

Let's look at an example of a particular Ethernet device, e1000. For the e1000 Ethernet network device, the following operations are defined:

```
static const struct net_device_ops e1000_netdev_ops = {
        .ndo_open                       = e1000_open,
        .ndo_stop                       = e1000_close,
        .ndo_start_xmit                 = e1000_xmit_frame,
        …
};
```

This set of operations is passed to the kernel in the following code sequence:

```
struct net_device *netdev;
…
netdev->netdev_ops = &e1000_netdev_ops;
e1000_set_ethtool_ops(netdev);
netdev->watchdog_timeo = 5 * HZ;
netif_napi_add(netdev, &adapter->napi, e1000_clean, 64);
```

NAPI ("New API") is an extension to the device driver packet processing framework. See (Linux Foundation, 2016)[3] for details. The netif_napi_add interface initializes the context and defines the polling function (e1000_clean) for the device, typically used to process receive events. The polling function is called to check whether any processing is needed on Rx queues.

The ethernet driver (e.g. e1000) typically allocates buffers to receive data that arrives from the network. These buffers are pinned in memory and mapped for DMA. The device driver interacts with the adapter hardware by placing pointers to these buffers in a ring buffer. As

data arrives, the adapter DMAs the data into the next available buffer and marks the buffer as used. An interrupt is generated to inform the driver that data has arrived, which then causes the polling function (e.g. e1000_clean) to be called to process the arrived data. The data buffer is passed up the stack to eventually be copied into the VM guest provided buffers. Every so often, the network adapter device driver adds additional buffers to the ring buffer so as to ensure that there are always enough available buffers to hold arriving data.

In order to support Zero-Copy Rx, we need to make changes to the network adapter device driver (and other drivers in the stack) to allow VM guest buffers to be used to directly receive data from the network via DMA, rather than the adapter device driver allocate and manage the buffers.

2.2 Changes to support Zero copy on Receive

In order to enable Zero-Copy Receive, we need to make changes to various device drivers and mechanisms in the hypervisor code.

- Add support in the virtio device driver to pass individual complete pages, which can then be used for direct DMA.
  In the current virtio implementation, buffers are passed to the hypervisor, typically large enough to hold either an Ethernet packet (about 1500 bytes) or a jumbo packet (64K), depending on configuration. On the other hand, the low-level Ethernet adapter driver (e.g. e1000) allocates individual full pages (4K) to receive data via DMA from the network. We need the virtio driver to allocate full individual pages, so that these can then be passed down to the Ethernet adapter driver to be used for DMA.

- In the network adapter driver, add functions to set individual specified buffers as elements of the Rx ring buffer for DMA.
  In the current implementation, the network adapter device driver allocates the buffers that are placed in the ring buffer to receive network data. We need to introduce in the network adapter driver a function to receive buffers from an external source (e.g. VM guest) to be placed in the receive ring buffer. We also need a mechanism to release these buffers when the device is reset for any reason.

- Add a new function to the struct `net_device_ops` to provide network adapter driver with pages for DMA.
  Each device driver has a set of functions that are exported to be called by other drivers. For a network adapter driver, these functions are specified in the `net_device_ops` struct. We need to add a function to the network adapter driver (e.g. e1000) which enables an upper level driver (e.g. virtio/vnet) to provide a page of memory into which data should be DMA-d. This function is added to the

net_device_ops structure, so that it becomes available to any upper-level driver that wants to call it. All intermediate level drivers must also then support this function call.

- Add flags in various places to indicate whether a buffer and/or transaction involves a zero-copy buffer.
  Current code has various flags to indicate what special performance features are being used, such as buffer merging or checkpoint offloading. We need to add flags in various parts of the code to indicate that a particular IO stream is performing Zero-Copy Receive, in order to provide the proper handling for different types of configurations.

- Add code in the intermediate device drivers to handle the data as zero-copy.
  All device drivers in the call stack must be checked that they support the Zero-Copy Receive function. In particular, the intermediate device drivers (e.g. macvtap, macvlan) must provide a function to pass buffers from the virtio/vnet driver down to the lower level network adapter driver.

2.3 Rx ring buffer and ndo_post_rx_buffer()

In order to support Zero-Copy Rx, we need a way to take VM guest kernel buffers and have the low-level network adapter direct data directly into those VM guest buffers via DMA. The network driver typically has a ring of buffer descriptors, pointing to available buffers into which the adapter performs DMA. Instead of allocating the DMA data buffers in the network adapter driver, we pass down the buffers provided by the VM guest. We add an ndo_post_rx_buffer interface to net_device_ops to allow an upper level component to pass down buffers to the network adapter driver for Rx DMA.

```
ndo_post_rx_buffer (struct net_device *netdev, struct sk_buff skb)
```

There might be several layers of drivers through which the buffer must be passed. For example, we saw in zero-copy Tx code that the buffer is passed through layers macvtap, macvlan, and then finally to the Ethernet device driver. We now have to navigate these layers to post the Zero-Copy Rx buffers. The lowest level driver needs to take the provided buffer, prepare it for DMA, and place it in the Rx ring buffer. Any intermediate drivers must also provide this function, but they simply send the buffer further down the stack.

The network adapter takes the specified page, maps it for DMA, and adds it into the Rx ring buffer. After data is DMA-d into the buffer, it is removed from the ring buffer and it is unmapped to prevent further DMA use. Eventually (either via a real interrupt or soft interrupt) NAPI identifies that the buffer has been filled. NAPI then wakes up the vhost thread to pass the buffer back up the stack.

2.4 Expected Performance Improvement

We ran some motivating experiments to estimate the expected performance improvement from the Zero-Copy Receive feature. The experiments indicated that we could save as much as 20% in cpu overhead for large packets, if we can avoid the extra data copy. This led us to the decision to progress with a full proof-of-concept implementation of the Zero-Copy Receive feature.

3 OSv

OSv is a new operating system designed specifically for running a single application in a VM. OSv is limited to running a single application (i.e., it is a *unikernel*) 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.

While some of the other unikernels that appeared recently each focus on a specific programming language or application, the goal behind OSv's design was to be able to run a wide range of unmodified (or only slightly modified) Linux executables. As a consequence many dozens of different applications and runtime environments can run today on OSv, including MIKELANGELO's use cases, and much of the work in WP2 and WP4 revolved around improving OSv's compatibility with additional Linux applications and runtime environments .

The MIKELANGELO cloud is based on 64-bit x86 VMs and the KVM hypervisor, so OSv needs to support those. But we wanted OSv to not be limited to that configuration, so today OSv supports most common cloud configurations: both x86 and ARM CPUs (both at 64-bit only) are supported, and so are most common hypervisors: KVM, Xen, VMware, VirtualBox. Support for HyperV is currently being added by an external contributor (who wants to run OSv in Microsoft's Azure cloud).

In the rest of this chapter we will provide a high-level description of the architecture of OSv's core - its kernel and its Linux compatibility. Two separate components that touch both OSv and the hypervisor - vRDMA and UNCLOT - will have separate chapters below. So will the MIKELANGELO's framework for building and deploying OSv-based images - LEET.

The work plan for OSv for the last half year of the project (M30-M36) is mostly to continue to polish and debug the components we already have, focusing on correctly and efficiently supporting additional applications related to, or similar to, the MIKELANGELO use cases.

3.1 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.

3.2 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).

3.3 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 five concrete filesystem implementations: devfs (implements the "/dev" hierarchy for compatibility with Linux), procfs (similarly, for "/proc"), ramfs (a simple RAM disk), ZFS, and NFS.

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.

We added NFS filesystem support (i.e., an NFS client), to allow applications to mount remote NFS shared storage, which is a common requirement for HPC applications.

3.4 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 thread-local storage (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 special overheads for system calls: 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).

OSv's ELF dynamic linker also supports the concept of "ELF namespaces" - loading several different applications (or several instances of the same application) even though they may use the same symbol names. OSv ensures that when an application running in one ELF namespace resolves a dynamically-linked symbol, it is looked up in the same ELF namespace and not in that belonging to the second application. We added the ELF namespaces feature as a response to MIKELANGELO's requirement of running Open MPI-based applications: Open MPI traditionally runs the same executable once on each core, each in a separate process. In OSv, we run those as threads (instead of processes), but we need to ensure that although each thread runs the same executable, they each get a separate copies of the global variables. The ELF namespace feature ensures that.

The ELF namespace feature also allowed us to give each of the Open MPI threads have their own separate environment variables. This works by ensuring that `getenv()` is resolved differently in each of those ELF namespaces.

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

The glibc ABI is very rich - it contains hundreds of different functions, with many different parameters and use cases for each of them. It is therefore not surprising that OSv's reimplementation of those missed some, or implemented some functions imperfectly. As a result, much of the development effort that went in MIKELANGELO into OSv revolved around fixing functions which were either missing or incorrectly implemented, so that we could correctly run additional applications and use cases on OSv. More details about this work and what was improved every year is given in the WP4 series of deliverables.

3.6 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, a page-cache miss - needing to read a page from disk, or needing to write when memory is low - always blocks the running thread, so it requires multiple application threads to context-switch. When we introduce the Seastar library in the following section, we explain that for this reason Seastar applications use AIO (asynchronous I/O), not `mmap()`.

OSv's does not currently have full support for NUMA (*non-uniform memory access*). On NUMA (a.k.a. *multi-socket*) VMs, the VM's memory and cores are divided into separate "NUMA nodes" - each NUMA node is a set of cores and part of the memory which is "closest" to these cores. A core may also access memory which does not belong to its NUMA node, but this access will be slower than access to memory inside the same node. Linux, which has full support for NUMA, provides APIs through which an application can ensure that a specific thread runs only on cores belonging to one NUMA node, and additionally only allocates memory from that node's memory. High-performance applications, including the Open MPI HPC library, make use of these APIs to run faster on NUMA (multi-socket) VMs. OSv does not yet have full support for these APIs, so Open MPI performance on multi-socket VMs suffers as cores use memory which isn't in their NUMA node. To avoid this issue, we recommend that very large, multi-socket VMs be split into separate single-socket (but multi-core) VMs, and shared memory be used to communicate between those VMs (this technique is explained in the UNCLOT section below).

3.7 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 of 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 our paper "OSv — Optimizing the Operating System for Virtual Machines".

One of the consequences of our simpler and more efficient scheduler implementation is that in microbenchmarks, we measured OSv's context switches to be 3 to 10 times faster than those on Linux. However, because good applications were typically written knowing that context switches are slow, and made an effort to reduce the number of context switches, the practical benefit of this speedup in most real-life applications is small.

3.8 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.

3.9 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 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, without reading the packet data. Only when the application calls read() (or poll(), etc.) on the socket, the TCP processing is 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.

3.10 DHCP Client

OSv contains a built-in DHCP client, so it can find its IP address and hostname without being configured manually. For more extensive configuration, we also have cloud-init:

3.11 Cloud-init Client

OSv images can optionally include a "cloud-init" client, a common approach for *contextualization* of VM images, i.e., a technique by which a VM which is running identical code to other VMs can figure out what it is its intended role. Cloud-init allows the cloud management software to provide each VM with a unique configuration file in one of several ways (over the network, or as a second disk image), and specifies the format of this configuration file. The cloud management software can specify, for example, the VMs name, what NFS directory it should mount, what it should run, and so on.

3.12 REST API

OSv images can optionally include an "httpserver" module which allows 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 have a "`memory_malloc`" trace in the beginning of the `malloc()` function, and a "`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 zero 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 for brevity is omitted here - it was already shown in deliverable D2.16.

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. OSv also optionally provides a "swagger" GUI to help a user determine exactly which REST API requests exist, and which parameters they take.

- `/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 was developed so that it can retrieve data from an OSv guest, or multiple OSv guests, using the OSv REST API just described. 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.

4 Seastar

4.1 Introduction

As we noted above, the primary goal of OSv is to run existing Linux software, because most MIKELANGELO use cases required running existing code. Today's Linux APIs - POSIX system calls, socket API, etc. - were formed by decades of Unix and Linux 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 in the development of the guest operating system was to propose new APIs which will offer new applications dramatically better performance than unmodified Linux applications - provided that the application is rewritten to use these new APIs,

In the research paper "OSv — Optimizing the Operating System for Virtual Machines"[4], one of the benchmarks used was Memcached, a popular cloud application used for caching of frequently requested objects to lower the load on slower database servers. 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 a nice speedup that we get just by replacing Linux in the guest by OSv, without modifying the application at all. But we wanted to understand if there is something we could do to get significantly higher performance. 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 the previous section), 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 and 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 the aforementioned OSv paper, 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 was an impressive 4 times better than the original memcached server.

But while the simple "packet filter" API 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 (DPDK is a popular 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 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 can bypass the legacy kernel APIs, e.g., it can directly access 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.

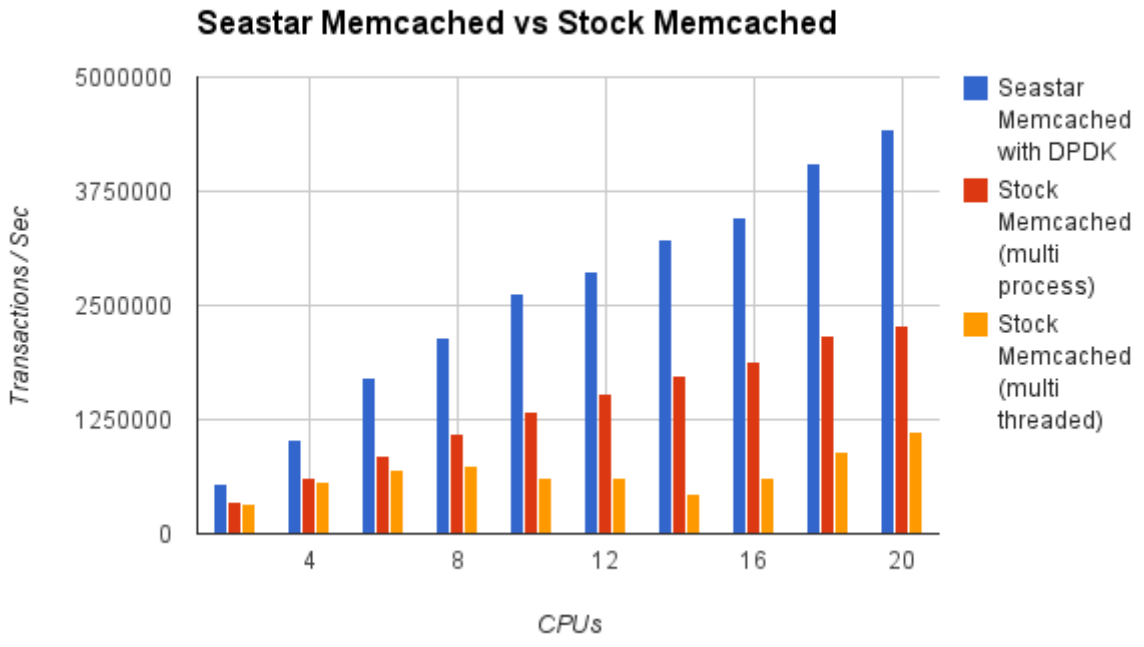

Figure 4. Performance of stock memcached (orange) vs Seastar reimplementation of memcached (blue), using TCP and the memaslap[5] workload generator - for varying number of cores The red bars show a non-standard memcached deployment using multiple separate memcached processes (instead of one memcached 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.

Applications that want to use Seastar will need to be rewritten to use its new (and very different) APIs. This requires significant investment, but also comes with significant rewards: The creator of the Seastar library, ScyllaDB, spent the last two years reimplementing the popular Cassandra distributed database in C++ and Seastar, and the result, "Scylla" (which, like Seastar and OSv, is released as open source[6]), has much higher throughput than Cassandra: Independent benchmarks[7] by Samsung showed Scylla to have between 10 to 37 times higher throughput than Cassandra on a cluster of 24-core machines in different workloads, In fact, the Scylla distributed database performs so well that it has become ScyllaDB's main product, making the company highly dependent on Seastar's exploitation. For this reason, ScylllaDB has been investing into Seastar additional efforts beyond what is being funded by the MIKELANGELO project, and plans to continue developing Seastar even after the project ends.

We use Scylla, the Seastar-based reimplementation of Cassandra, in the "Cloud Bursting" use case. But our goal is for Seastar to be a general-purpose APIs which will be useful to many kinds of asynchronous server applications, which are often run on the cloud. As such, we are making an effort of providing a rich, well-balanced, and well documented API, and also writing a tutorial on writing Seastar applications (a draft of which was already included in D4.4 and D4.5). At the time of this writing, we know of at least two other companies besides ScyllaDB which based their product on Seastar, and more are considering doing this.

4.2 Seastar architecture

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 drastically reduce 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.

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 higher 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++ 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 I/O") 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 large 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.

4.3 Additional Seastar components

Above we explained a few key parts of Seastar's architecture, including its sharded design, futures and continuations, asynchronous disk I/O, and the user-space TCP/IP stack. But Seastar includes many more components, which we will briefly survey now. Work on polishing and documenting all of these components is ongoing, but the CPU scheduler will be receiving particular attention and improvements in the upcoming months in an effort to continue reducing the latency of ScyllaDB and other Seastar applications.

1. **Reactor:** The Seastar "reactor" is Seastar's main event loop; A reactor thread runs on each of the cores dedicated to the application, polls for new events (network activity, completed disk I/O, expired timers, and communication between cores) to resolve the relevant futures, and runs continuations on the ready queue (continuations attached to futures that have been resolved).

   The Seastar reactor is **not** preemptive, meaning that each continuation runs to completion or until it voluntarily ends. This means that application code in each continuation does not need to worry about concurrent access to data - nothing can run on this core in parallel to the running continuation, and the continuations on other cores cannot access this core's memory anyway (thanks to the share-nothing design, more on that below).

   When idle, the reactor can either continue polling or go to sleep; The former approach improves latency, but the latter allows lower power consumption. The user can choose one of these options, or let Seastar choose between them automatically based on the load.

   In any case, Seastar applications are meant to monopolize the CPU cores they are given, and not time-share them with other applications. This comes naturally in the cloud, where the VM runs a single application anyway.

2. **Memory allocation:** Seastar divides the total amount of memory given to it between the different cores (reactor threads). Seastar is aware of the NUMA configuration, and gives each core a chunk of memory it can most efficiently access. The C library's `malloc()` (and related standard C and C++ allocator functions) is overridden by Seastar's own memory allocator, which allocates memory from the current shard's memory area. Memory allocated on one shard may only be used by the same shard. Seastar has limited facilities for allowing one shard to use memory allocated by another shard, but generally these are not to be used, and shards should communicate via explicit message passing as described below.

3. **Message passing between shards:** Seastar applications are sharded, or *share-nothing*, and shards do not normally read from each other's memory because, as we already explained, doing that requires slow locks, memory barriers and cache-line bouncing. Instead, when two shards want to communicate, they do this via explicit message passing, implemented internally over shared memory. The message passing API allows running a piece of code (a lambda) on a remote shard. This code is run by the reactor running on the remote shard - as explained above, without risk of parallel execution of continuations.

4. **Log structured allocator (LSA):** Seastar's `malloc()/free()` described above is more-or-less identical to the traditional one, except the fact that it uses a different memory pool for each core. Perhaps the biggest downside of the `malloc()/free()` API is that it causes fragmentation - it is possible (and after a long run, fairly likely) that allocating and freeing of small objects will prevent allocation of a larger object, even though we might still have plenty of unused memory. Because the user of the allocated object may save pointers to them, the memory allocator is not allowed to move them around to fix fragmentation.

   Garbage Collection is a common way to solve this problem, because part of the work of a garbage-collector is to compact (i.e., move) the objects in memory, so it can free up large contiguous areas of memory. However, GC comes with an additional anti-feature: It needs to **search** where the garbage (freed) objects are, and this search is where most of GC's disadvantages come from (e.g., pauses, and need for plenty of spare memory).

   Seastar's LSA (log-structured allocator) aims to be the best of both-worlds: Like `malloc()/free()`, memory is explicitly allocated and freed (in modern C++ style, automatically by object construction/destruction), so we always know exactly which memory is free. But additionally, LSA memory may move around (or automatically be evicted), and the LSA APIs allow tracking where an object moved, or prevent movements temporarily.

5. **I/O scheduler:** One of the key requirements that arose in the "Cloud Bursting" use case was to ensure that performance does not deteriorate significantly during a period of cluster growth. When a Cassandra cluster grows, the new nodes need to copy existing data from the old nodes, so now the old nodes use their disk for both streaming data to new nodes, and for serving ordinary requests. It becomes crucial to control the division of the available disk bandwidth between these two uses. The I/O scheduler allows the application to tag each disk access with an I/O class, for example

a "user request" vs. "streaming to new node", and can control the percentage of disk bandwidth devoted to each class.

6. **I/O tune:** Seastar's disk API is completely asynchronous and future-based just like everything else in Seastar. This means that an application can start a million requests (read or write) to disk almost concurrently, and then run some continuation when each request concludes. However, real disks as well as layers above them (like RAID controllers and the operating system), cannot actually perform a million requests in parallel; If you send too many, some will be performed immediately and some will be queued in some queue invisible to Seastar. This queuing means that the last queued request will suffer huge latency. But more importantly, it means that we can no longer ensure the desired I/O scheduling, because when a new high-priority request comes in, we cannot put it in front of all the requests which are already queued in the OS's or hardware's queues, beyond Seastar's control.

   So clearly Seastar should not send too many parallel requests to the disk, and it should maintain and control an input queue by itself. But how many parallel requests should it send to the lower layers? If we send too few parallel requests, we might miss out on the disk's inherent parallelism: Modern SSDs, as well as RAID setups, can actually perform many requests in parallel, so that sending them too few parallel requests will reduce the maximum throughput we can get in those setups.

   Therefore we developed "IOtune", a tool that runs on the intended machine, tries to do disk I/O with various levels of parallelism, and discovers the optimal parallelism. The optimal parallelism is the one where we get the highest possible throughput, without significantly increasing the latency. This is the amount of parallelism which the disk hardware (and RAID controllers, etc.) can really support and really perform in parallel. After discovering the optimal parallelism, IOtune writes this information to a configuration file, and the Seastar application later reads it for optimal performance of Seastar's disk I/O.

7. **CPU scheduler:** The I/O scheduling feature which we described above goes a long way to ensure that when the system needs to run two unrelated tasks (such as the data-streaming and the request-handling mentioned above) one of them does not monopolize all the resources. However, in some cases ScyllaDB has work in which there is very little disk I/O but significant computation - and in such cases the I/O scheduler is not enough and we need an actual CPU scheduler.

   We explained above how the reactor has a queue of ready continuations which need to run; With the CPU scheduler, we have several of these queues, each belonging to a different "scheduling group". The application author tags continuations as belonging

to a particular scheduling group, and the reactor's CPU scheduler chooses which scheduling group's continuation(s) to run next, while considering fairness and latency requirements. The latency requirements are especially important, and are one of Seastar's strong points: For example, we do not want continuations belonging to some background operation to run for 100ms, while continuations belonging to client request handling just wait (and incur a large user-visible latency).

8. **SEDA**: Somewhat related to the CPU scheduler is Seastar's support for batching of continuations inspired by the well-known SEDA ("staged execution event-driven architecture") concept:

   The fact that Seastar works with short continuations instead of coarse threads contributes to its higher throughput and lower latency, but introduces a new problem: The CPU constantly jumps between different pieces of code, the instruction cache is not large enough to fit them all, and execution speed (instructions per cycle) becomes suboptimal. To solve this, Seastar provides a mechanism whereby the application can ask to batch certain continuations together. For example - instead of running a certain continuation immediately, Seastar might wait until 100 instances of the same continuation are ready - and then run them one after another. An important part of this Seastar mechanism is that it ensures the latency required by the application: A continuation will not be delayed for batching by more than that desired latency, and, conversely - we cannot collect so many continuations into one batch so that when we run this batch, everything hangs for too long a time. The end result of Seastar's SEDA mechanism is a delicate balance between high execution efficiency (high instructions/cycle) and low latencies.

9. **TLS** (transport layer security): In addition to providing an asynchronous, zero-copy, interface to the TCP/IP protocol suite, Seastar also gives the same interface to secure TLS (formerly known as SSL) connections.

10. **RPC:** Seastar runs on a single machine, but many Seastar applications, including the Scylla distributed database which we use for the "Cloud Bursting" use case, offer a distributed service and therefore need convenient primitives for communicating between different machines running the same application. So Seastar offers RPC capabilities, with which the application running on one machine can call a normal-looking function returning a future value, while the Seastar seamlessly communicates with the remote machine, runs the function there, retrieves the result, and resolves the previously-returned future with that result. Seastar's RPC also includes advanced features typically needed by distributed applications, such as serialization of data,

negotiation (allowing different versions of the application to communicate with each other) and compression.

11. **Seastar threads**: Those are not actual OS-level threads, but rather a mechanism to allow thread-like programming in Seastar: Code running in a "Seastar thread" has a stack like in normal threads; it can wait for a future to become resolved (by calling its get() method), and when the future resolves, the code continues to run from where it left off. The main overhead of Seastar threads compared to regular continuations is the memory taken up by the stacks, so we normally use Seastar threads for pieces of code where we have tight control over the number of times they run in parallel.

12. **System calls:** In some cases, a Seastar application cannot avoid making a blocking system call. For example, to delete a file the unlink() system call must be used - Linux (or OSv) do not provide an asynchronous version of this system call. But Seastar application code must never block, as this can block the single reactor thread running on this core, and leave it idle. Therefore Seastar provides a mechanism of "converting" a blocking system call to a regular Seastar future. This is implemented by having a separate pool of threads dedicated to running the blocking system calls (and those threads will indeed block). Because these system calls are not frequently needed, the thread-based implementation does not negatively impact the performance of the application.

13. **Various utility functions:** Seastar provides a large number of utility functions for making future-based programming more approachable. It has support for loops and iterations, semaphores (for limiting the number of times that a particular continuation may run in parallel), gates (for ensuring that all code related to some service completes before the service is shut down), pipes, input and output streams, and more.

14. **Asynchronous file I/O on buggy filesystems:** we already mentioned AIO above, but there is an extra complication: Some filesystems - such as the popular ext4 on Linux - have buggy support for asynchronous I/O. The key requirement of asynchronous I/O is that such calls never block - they must always return immediately, and let the caller know later when the operation completed. Yet, on Linux ext4, some operations do block - e.g., while a file's size is being extended and new blocks need to be allocated. This is clearly a bug in Linux, but one which Seastar had to work around. The solution involves knowing which operations do block, and avoid them or make sure they rarely happen and run them in the syscall thread pool. With these fixes, Seastar provides a true asynchronous experience even on ext4.

15. **Exception handling:** Seastar futures can resolve to either a value (we provide an example in the next section) or an *exception*. Seastar provides various mechanisms for handling exceptions which happen asynchronously, and should abort a chain of continuations, be handled, or both. We also made a significant effort to ensure that exception handling is efficient, but because of locks used by current implementations of the compiler, they should not be overused.

16. **Object lifetime management:** Asynchronous applications need to worry about object lifetime management - e.g., an object which was created for the needs of a particular request, must be kept alive during the handling of this request (otherwise, we will crash), but must be destroyed when the request completes (or we'll have a memory leak). Seastar provides various tools for object lifetime management, including optimized implementation of shared_ptr (to not use atomic operations, because thread-safety is not needed by Seastar applications), a do_with() function for ensuring that an object lives until a future is resolved, temporary_buffer for moving around temporary data which is automatically freed when no longer needed, and more.

17. **HTTP server:** Seastar includes an HTTP server implementation (written, of course, in Seastar), which can be used by the application for various things - e.g., implementing a REST API.

18. **Logger, metrics and monitoring:** Seastar also includes mechanisms for logging messages and errors, and collecting metrics (counters, sums, etc.), and for reporting them to various collection mechanisms such as Collectd or Prometheus - and through them also to MIKELANGELO's Snap.

5 LEET - Lightweight Execution Environment Toolbox

Over the course of the MIKELANGELO project, the package and application management has matured significantly. It was initially named MIKELANGELO Package Manager (MPM) to indicate its broader meaning. However, with progression of the project, it became harder to present the (exploitable) results under this name. Namely, we were introducing functionalities far outreaching the initial plan of plain package management, as defined in T4.3. We have therefore decided to extend the internal and external exploitation strategies and rebrand everything related to the deployment, management, reconfiguration of OSv-based unikernels under a single umbrella term - Lightweight Execution Environment Toolbox, LEET.

The work initially started mainly as a narrow WP4-specific (Guest Operating System) effort which was later expanded into cross work package effort, including WPs WP5 and WP6. At the very core of the application management are enhancements made to the Capstan[8] tool initially created by Cloudius Systems (now called Scylla DB). MIKELANGELO project introduced the concept of application packages along with their composition into self-contained runnable virtual machine images - unikernels. The package management then evolved integrating the OpenStack infrastructure services (including the WPs WP5 and WP6). This integration automated the deployment and management of unikernels to the users who are now able to use OSv-based applications independent of the deployment target: either locally or in the cloud (private or public OpenStack, Amazon EC2 and Google GCE). This has been initially achieved by implementing OpenStack provider for Capstan, and has later been replaced by integrating Capstan with the UniK[9] tool. Integration with UniK gave significant visibility to the MIKELANGELO project in the unikernel community due to our collaboration with the upstream community.

Following subsections briefly describe current state of technologies dealing with the management of applications and presents high-level plans for the implementation of additional features until the end of the project. It consists of three parts: package management, cloud management and application orchestration.

5.1 Package management

As described in the previous deliverables (for example D4.5 - D4.5 OSv – Guest Operating System – intermediate version[10]) the core of the package and application management tools of the MIKELANGELO project is based on Capstan. Package management and the ability to easily compose lightweight virtual machine images is the foundation of the overall application management in the MIKELANGELO project. To understand the core ideas behind package management in similar projects, a detailed analysis of existing tools for building and managing OSv-based virtual machine images, as well as some other unikernels like rumprun

and MirageOS, has been performed. This led to the design of the initial architecture that has since then regularly been updated based on the new findings.

Because OSv allows reuse of precompiled libraries and applications, we wanted to design the architecture that revolves around prebuilt packages. This approach enables fast and simple composition of OSv images, removing the need for lengthy and error prone re-compilation of source code from the process. This architecture (Figure 5) was initially presented in D2.16[11] and assumes three layers. The bottom layer represents the package repository storing application packages and their metadata. Because multiple repository providers are envisioned, an abstraction of a package repository is required. Primary package repository is an internal one (Local Repository in the diagram below) serving both as a temporary place for packages and virtual machines as well as local environment for execution of OSv unikernel.

The middle layer is the core of the package manager and provides a set of functionalities for building, validating and composing application packages and runnable virtual machine images. The topmost layer (Client) is a user-facing component that allows interaction with the remaining two layers. It is currently implemented as a command line interface (CLI).

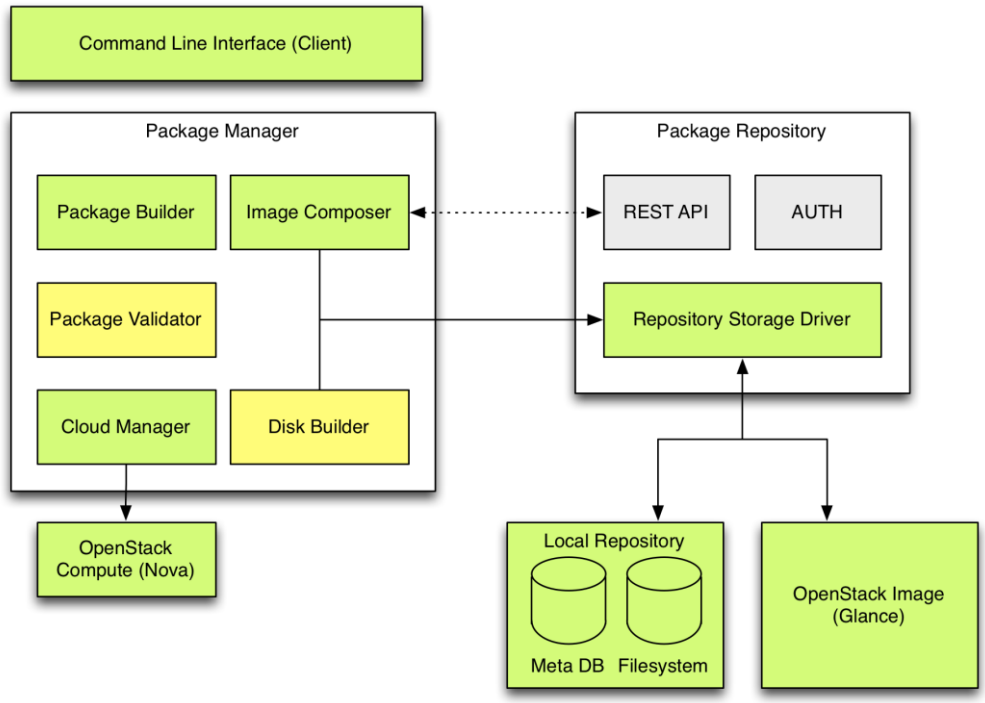

Figure 5. Initial architecture of the MIKELANGELO Package Management. Components marked with green color represent the main development focus.

The diagram has been updated based on the current status, briefly presented in the following list:

- Initial version of Package Validator has been put in place. The main purpose of this component is to allow users to simply validate that the content of the required packages has been uploaded successfully. This involves the ability to run a simple application, pre-configured by the application manager, and testing against the expected results, namely it is a variant of unit and integration testing for application packages. In most cases the expected result is given as an expected output printed by the application. For example, in case of Node.js runtime package, the test could contain a simple JavaScript script, a run configuration specifying the exact command, and a text that is echoed by the script. OpenFOAM package could check that the invocation of the application indeed prints the name of the application solver used in the image.
- The cloud integration has been finalised. This included the integration with OpenStack image service as well as introduction of an abstract cloud manager and implementation for OpenStack provider. Together, this allows users to seamlessly deploy unikernels to OpenStack. This is reflected in the Openstack Nova and Glance components.
- Introduction of Disk Builder. This is current work in progress. This main goal of this component is to simplify the creation of ZFS disk images with specific content. By default, package manager composes a self-contained virtual machine image. However, it has become evident that end-users are typically interested in ability to have a single VM with just the core operating system and runtime environment, while the data, configuration and application scripts or binaries are provided through external volumes attached to the main instance. Since creating ZFS disk images is a rather cumbersome task, we started working on a component that would create such images of specified size and upload the required content (this would not include the kernel and accompanying libraries). Further details on the use of this component are outlined in the following subsections.

Further to these new enhancements, several updates to the existing requirements have also been made since the last release (MS4, M24). Below is a list of these updated requirements with their initial descriptions and the comments in their implementation and progress at MS5 and M30.

1) **Elimination of external dependencies (initial requirement).** Capstan currently still relies on QEMU/KVM to compose and configure target virtual machine images. Currently this is not a mandatory requirement because users are still using Capstan in controlled environments, but the requirement is very important for external users. **M30 comments:** This requirement has not been addressed yet and it has not been decided whether to proceed with the implementation. The main reason for this is that it has already been shown that these dependencies can easily be overcome by

wrapping VM image composition into Docker containers. As task T4.3 is also responsible for the preparation of all relevant application packages (either used by MIKELANGELO use cases or are relevant to the broader communities) we have been automating the creation of the packages by wrapping build recipes into such containers allowing repeatable composition of all packages[12].

2) **Change of the underlying architecture (initial requirement).** While we were implementing initial support for the chosen cloud provider, it became even more evident that the current architecture of the Capstan tool is not suitable for supporting alternative providers nor for providing services to external integrators (for example HPC integration). Because our main target so far was simplification from the end user's perspective, we are planning to address this requirement in the next iteration.
   M30 comments: Although we initially planned to expose the packaging mechanisms via API, we decided to keep the current architecture as the integration point is fairly small and can be handled easily with the approach.
3) **Support for other guest operating systems (initial requirement).** We have reevaluated this requirement with other potential systems, in particular unikernel systems. It became evident that there are significant differences in the way images are composed and/or compiled. Consequently, we are postponing this requirement for now. In D6.1 [6] we also mention a new project (Unik [19]) that has been trying to address this. We are following the project closely, and in discussions with the team behind it about potential collaboration.
   M30 comments: This requirement has been covered by the integration of our tools into UniK project. Several attempts have been made to introduce other unikernels, however none of them proved suitable for the requirements of the MIKELANGELO project. The other unikernel approaches are too specific (not general purpose enough) or cannot be packaged as efficiently as OSv.
4) **Run-time options (initial requirement).** This requirement is going to be addressed in the next iteration. It is important to allow users to work with OSv-based applications as if they were simple processes. During the comprehensive benchmarking of OSv applications it was observed on several occasions that it was impossible to understand the application without looking at the actual code. To this end we are going to expand the package metadata capabilities allowing package authors to provide reasonable documentation as well as the intended use cases (for example, default commands).
   M30 comments: We have enhanced the ability to customise the run configuration by allowing default values of predefined configurations to be overridden from the command line interface.
5) **Finalise cloud integration.** The initial target is to fully support OpenStack integration with more features available out-of-the-box. When we move from application

packages, into runnable instances it is essential that their lifecycle can be managed from a central place. OSv applications are not typical in that the user can seamlessly connect to a remote machine to reconfigure it. Besides improving support for image and compute services, additional services are going to be integrated: networking and storage. This is going to be addressed in the next iteration.

M30 comments: This requirement has been addressed. Further details given in the following subsection.

6) **Abstraction of cloud provider.** We believe that relying on a single cloud framework will limit the exploitation potential of the application packaging. To this end we are already planning to abstract the concept of a cloud provider. An implementation for Amazon AWS (and potentially OpenNebula) is going to be provided with this improvement.

   M30 comments: Same as previous requirement.

7) **Package hub.** Users are currently required to download and install the package repository locally. For the time being the number of packages and consequently the size of the repository is not large, so a central hub of all packages has not been necessary. With an increasing number of applications and packages, such a centralised hub will simplify the workflow acquiring only packages required by the end user.

   M30 comments: We have introduced a package hub, publicly available at MIKELANGELO Package repository[13]. The Capstan tool is extended with the querying and automatic downloading of required packages.

8) **Package usage description standardization.** A package author has currently no standard means of providing package usage description where she could describe how to properly use the package. Package authors need a way to describe: (i) configuration files that must/can be provided, (ii) each configuration set purpose, (iii) parameters description and (iv) package limitations. In the future we should consider introducing a new file (e.g. /meta/doc.yaml) where package description would be stored.

   M30 comments: This is work in progress. We have added support for run configuration documentation, however we regularly get feedback from internal or external users about missing built-in documentation system. The final release will thus include package documentation that will be exposed to package users.

9) **Package versioning.** In report D4.5 we have also introduced a requirement for allowing more comprehensive versioning of the packages. Even though version has been introduced in the initial release of the tool, it was not used at all yet. We are still considering practical value for this as the current package repository is still maintainable.

In summary, the outlook for the last phase of the package management tool is to become even more user friendly and introduce some functionalities that have been identified while implementing the internal use cases as well as some other applications that are using unikernels. These requirements are frequently verified with the community and are now mainly targeting the application orchestration presented in section 5.3.

5.2 Cloud Management

Management in cloud environment has already been implemented before milestone MS4. The main contribution in this area was the implementation of OpenStack connector into Capstan and UniK tools. Both implementations were presented in detail in report D4.5 (OSv - Guest Operating System – intermediate version)[14]. This report also outlined some of the potential functionalities cloud management tools foresaw important, when bringing unikernels closer to the cloud management.

In the meantime the UniK project got significantly delayed because the whole team behind the project left Dell EMC. XLAB has discussed this transition and the effects it will have on the community with the UniK team lead. Albeit assuring the project will be preserved, it has now stagnated for several months. To this end we have decided to postpone the implementation of the described features. For completeness and in case the project is revived, we are listing these requirements next. The descriptions are taken from the aforementioned report D4.5 and updated with the current status.

Further integration of OpenStack provider. Should the UnIK project be revived, we plan to add support for block storage attachments. This will allow to bootstrap numerous small OSv-based unikernels and attach data and configurations on the fly[15].

Use UniK Hub as a central repository for Capstan packages. UniK platform has introduced a central repository called UniK Hub where users are allowed to store their unikernels. Since Capstan is already providing it's own centrally managed package repository, we are going to consider benefits of integrating that into UniK Hub. One potential problem with the UniK Hub is that it is dependent on the infrastructure provider (for example, OpenStack has a different hub than VMware and Amazon Web Services). Since Capstan builds images that can be suited to other providers, it may be better to focus on improving Capstan's package repository capabilities than integrating it with a Hub that may not exploit all of its potential.

Support cloud orchestration templates. This requirement discussed the ability to allow application orchestration directly from the UniK tool. We are going to abandon this approach in favour of enhancing the application orchestration as part of the Kubernetes integration. We believe this will enable much better reuse of unikernel-based applications. Sheer size of

the Kubernetes community also guarantees that our integration efforts will actually be used by a broader community.

5.3 Application Orchestration

Orchestration of OSv-based unikernels is one of the challenges that hasn't been addressed extensively in MIKELANGELO until now. The application management focused on being able to simplify the creation of self-contained virtual machines executing a single process and deploying them to the cloud. Limited efforts towards orchestrating the services were made in experiments using OpenStack Heat[16] orchestration. Albeit working, it proved a bit cumbersome and very specific to the underlying infrastructure vendor. In collaboration with DICE[17] H2020 project we analysed Cloudify[18] project which abstracts the underlying orchestration engine by using the TOSCA[19] standard. At the time this meeting was held (~M27) Cloudify focused almost exclusively on OpenStack as well so we again decided not to go along this path.

Even in the initial project proposal, three years ago, we stated that using unikernels should be as simple as using containers. Therefore, we were also closely following the Kubernetes[20] project throughout the entire MIKELANGELO project. The Kubernetes system gives its users power over deployment, scaling and management of containers (for example based on Docker). Using its open format, it allows one to configure the whole application stack that is automatically deployed and configured on the fly. Each service is deployed in a separate container and all are properly interconnected to be able to exchange information.

Earlier in this year (2017) Mirantis published an initial (pre-alpha) release of a plugin for Kubernetes that provisions virtual machines based on Kubernetes specification. This plugin, Virtlet[21], extends Kubernetes Container Runtime Interface (CRI)[22] and ensures that service specification is provisioned as a virtual machine just as if it were a container. This offers tremendous power to the users who are now able to choose the optimal deployment target through the same well-known interface. For example, an application can consist of an Oracle Database and a set of services interacting with the database and with each other. To achieve optimal performance and isolation, the database should be deployed into a dedicated virtual machine, while the services themselves can be deployed as containers. Virtlet allows for such a mix through the same interface, offered by Kubernetes.

MIKELANGELO has been in close contact with Mirantis from the very beginning to discuss the plans and potential collaboration. The initial plan was to evaluate the technology to assess whether it is useful in the context of unikernels. To this end, we have extended the microservice application[23] we have been developing. This application serves as a showcase for the whole application package management stack. Based on this, MIKELANGELO became the first external source code contributor to the Virtlet project. MIKELANGELO was also

invited to present the results in the official Kubernetes sig-node meeting[24]. This contribution exposes virtual machine logs to Kubernetes management layer. The reasoning behind the contribution lies in the fact that contrary to the Linux guests, OSv does not have the ability to connect to the machine and examine the logs. Virtlet provided access to the console of the VM, however this does not contain the log that was written before the console has been opened. It furthermore provides plain text logs which are not understood by Kubernetes and are thus not exposed to the users through its interfaces.

The following figure shows a schema of the logging contribution that we have designed and implemented. The main goal was to be completely compliant with the existing architecture while keeping the logging independent of the other components of Kubernetes and Virtlet.

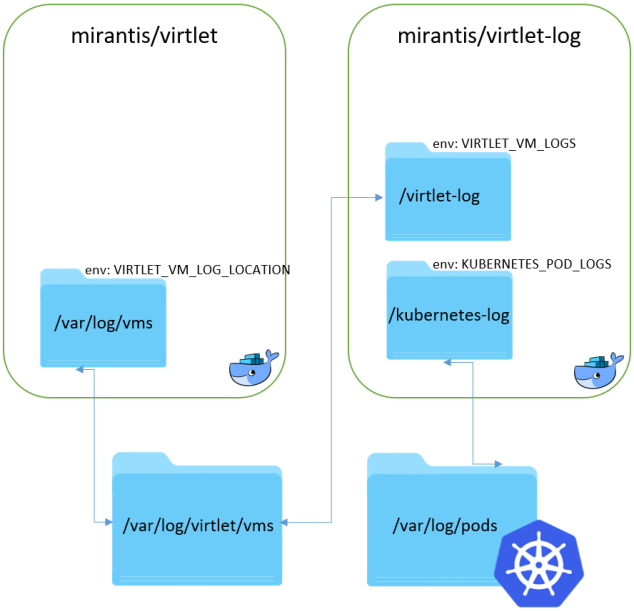

Figure 6: Logging schema introduced into Kubernetes Virtlet.

In case logging is enabled, the virtual machines are configured to redirect all messages that are printed to the serial console into a log file stored in `/var/log/virtlet/vms`. This directory is then mounted into a special *virtlet-log* container, which is responsible for monitoring for changes in this directory. Whenever a new log file is identified, a separate log worker is created, which will then follow the log file for changes and convert it to a log file compliant with Kubernetes/Docker (this file basically contains a single JSON per line, consisting of a timestamp and the actual log message).

Albeit this contribution is rather basic, it serves as the foundation of two important pillars. First, by exposing logs to the end users, it allows seamless execution of OSv-based services on top of Kubernetes. Users no longer need to build their instances for debugging purposes as they will get all the necessary information directly from the management layer.

Second, it has positioned MIKELANGELO project as an external contribution to an open source project, sponsored by one of the biggest open source contributors to OpenStack and Kubernetes. This will also benefit the Virtlet project when considering the incubation into the Kubernetes ecosystem.

The plan for the remaining phases of the project is to bridge more gaps between the containers, Linux-based guests and OSv-based guests. The following is the list of features that have been identified so far, but as the integration is quite new, this might change to address the crucial needs.

Enable unikernel contextualisation. The Kubernetes Virtlet still relies only on the NoCloud[25] cloud-init data source. Albeit OSv has added support for NoCloud, it supports a custom exchange format. The decision was made while adding contextualisation support into HPC integration where it was important only to have an integration possible. With Virtlet, it is now required to adhere the specification and thus support the meta- and user-data to be provided as ISO9660 volume. At the time of writing this report, alternatives are being investigated, e.g. to instead integrate OSv's format into Virtlet, however this might over complicate the integration. One of the main reasons why being able to use contextualisation features from Kubernetes is vital, is the ability to set the command to run within OSv instance. At the moment, Kubernetes is only able to provision unikernel as it is meaning that each service has to have its own VM image even if they differ in nothing but boot command. With full cloud-init support, however, a single VM image with several run configuration will be required.

Volume attachments support. In containers, various custom data (e.g. configuration, data, scripts) are frequently exported to the container as external volumes, attached to them. Virtlet is already adding support for volume attachments, however in case of OSv, getting the volume to work properly with its Virtual File System is not trivial. This is because ZFS is the only local filesystem known to OSv. Integrating additional filesystems may be too complex, so we are considering adding additional VM composition phase. This would allow one to build OSv-compliant VM images in advance and attach them dynamically to the deployed instances. Reconfiguration, data initialisation and business logic implementation would thus be separated from the main unikernel containing only the enabling runtime environment.

5.4 Application Packages

The following is the list of pre-built application packages. Some packages are merely updated to the latest version, while others are completely new. The packages are hosted in a public MIKELANGELO package repository[26]. These packages can be used directly using the tools from LEET.

OSv launcher (OSv kernel)
A bootable image containing the OSv kernel and the tools required to compose the target application image. This image is inherently included in all application images whenever they are composed. A bootstrap package accompanies the launcher further providing system-wide libraries that are required for proper operation of the OSv-based VM (for example, a library for the ZFS file-system [15]).

HTTP Server
The package contains the REST management API for the OSv operating system. It allows users to query and control all kinds of information about the OS.

Command Line Interface (CLI)
Provides a simple interactive shell for OSv virtual machines, mostly useful for debugging purposes. The CLI primarily offers a user-friendly interface to the HTTP server. Therefore, it automatically includes the HTTP Server package.

Open MPI[27]
This package contains libraries and tools required to run MPI-based applications. It is intended as a supporting package for HPC applications that need the MPI infrastructure: either just Open MPI libraries or also the *mpirun* command for launching parallel workloads.

OpenFOAM Core[28]
This package includes the base libraries, tools and supporting files that are required by arbitrary applications based on OpenFOAM toolkit.

OpenFOAM simpleFoam
OpenFOAM solver application, used extensively by the Aerodynamics use case. The package only contains a single binary (application) and references the core package above for the remaining functionalities. This solver is has been validated and evaluated most thoroughly in the MIKELANGELO project.

OpenFOAM solvers
OpenFOAM comes with additional solvers that can be used in different simulation scenarios. As part of our efforts, we have created packages for the following solvers[29]: pimpleFoam, pisoFoam, porousSimpleFoam, potentialFoam, rhoporousSimpleFoam and rhoSimpleFoam.

Java[30]
The package contains an entire Java Virtual Machine and supporting tools for running arbitrary applications on top of OSv. It has been successfully tested with several applications such as Cassandra, Hadoop HDFS and Apache Storm. Different versions as well as different Java profiles are used to pre-package Java runtime suitable for as many deployment scenarios as possible.

Node.js[31]

Node.js is another popular runtime environment used for backend development. It allows execution of JavaScript code making it ideal for lightweight services, microservices. It has also been used extensively in so called serverless architectures that are becoming widely popular with all public cloud providers. Several versions of Node are available, both for the 4.x and 6.x series which are provided as LTS (Long Term Support) versions. Users of these packages can deploy any of their existing Node.js applications and run them without any changes. A three part blog post[32] has been written to explain the use of this package in a simplistic microservice-like application.

Cloud-init[33]

This package allows contextualisation of running instances, for example setting the hostname of the VM and creating files with specific content. If this package is included in the application image, it will, immediately upon start, look for available data sources used in common cloud environments:

- Amazon EC2 [24]: used by Amazon AWS as well as default OpenStack. It provides a metadata service that cloud-init [25, 26] can access at a specific IP address.
- Google Compute Engine [27]: used in cloud, provided by Google
- No cloud-like data source [28]: in this case user data is attached to the target VM as a dedicated disk image from where the cloud-init will read and load necessary data.

Hadoop HDFS[34]

Hadoop HDFS is one of the core components of the big data application enabling practically unlimited distributed storage. It is used in many applications based on the Hadoop/Apache big data ecosystem. It is a Java-based application that inherently uses subprocesses for various internal tasks. This package was chosen to demonstrate the required steps for patching such applications and making them compatible with the OSv unikernel. Albeit HDFS works in OSv it has been shown that it lacks performance of Linux-based systems in report D6.3[35]. The performance issues have mainly been attributed to the virtual file system (VFS) and specifically the implementation of the ZFS file system in OSv.

Apache Storm[36]

Experiments with Apache Storm have started to complement the HDFS packages. Storm is used for processing Streams of data. Because of its architecture where workers get allocated by a master entity, it is an ideal example of the application that could benefit from the lightweight nature of OSv-based unikernels. Spawning additional OSv workers should be significantly faster than requesting new Linux virtual machines. One of the issues with Storm is that these workers are created by forking and launching a separate Java virtual machines (JVM) which makes them impractical for OSv. Since Storm also started losing in popularity we focused more on Apache Spark.

Apache Spark[37]
Apache Spark is one of the most popular tools for working with large amounts of data. Although it was initially designed for batch processing, Spark Streaming removes any necessity for Apache Storm. Similarly to Storm, Spark creates new workers by bootstrapping additional JVMs, however it does this by the support of the resource manager. This simplifies the integration and makes it significantly more transparent. To this end, we are working extensively in making the Spark package suitable for OSv.

5.4.1 Capabilities of the Application Packages

The aforementioned packages have already been used in various configurations. The Aerodynamics use case have extensively used the OpenFOAM simpleFoam package along with the Open MPI to support parallelism. All packages have been tested in local environment and in the cloud, while some have also been deployed in HPC environment (MPI, OpenFOAM, cloud-init).

The following are the functionalities of application packages that have been presented in the previous version of the package report. These descriptions provide updates as well as plans for those that haven't been addressed yet.

1. **Cloud-init module must support attaching of network shares.** Standard cloud-init supports the *mounts* option allowing users to provide specific volumes and network shares to be mounted into the target VM automatically upon start. For HPC applications it is mandatory that such support is integrated into OSv's cloud-init module allowing execution of experiments based on external data.
   Update: cloud-init module has been extended with the ability to attach one or more additional volumes. Only NFS backends are supported as no other requirements have been made. The structure of the mount command adheres the specification of the cloud-init.
2. **Customisable Open MPI application package.** The existing Open MPI application package must be extended to support additional configuration options suitable for wider ranges of MPI applications.
   Update: MPI package is exposing the functionalities of the Open MPI commands and has already been validated by the Aerodynamics and Bones use cases.
3. **Support for *mpirun* command.** HPC use cases in MIKELANGELO project (Aerodynamics and Cancellous Bones) rely heavily on the MPI for distribution of workload and synchronisation between individual processes. *mpirun* is the main entry point for these two applications. Because the OSv-based infrastructure is fundamentally different from the one in a typical HPC setting, the MPI application

package must ensure transparent use of virtualised workloads running in OSv. This requirement is also fundamental for external exploitation in HPC applications.

Update: *mpirun* command has been updated to work exactly the same as in standard environment. The only difference is the addition of specific communication channel that is required because MPI workers are configured through their REST APIs instead of being invoked through SSH.

4. **Additional use-case specific pre-built application packages.** This is a placeholder requirement for all the remaining applications that will be running OSv-enabled application images, in particular for the Big-data and Bones use cases. All of these packages need to be prepared in a way suitable for a broader audience. This will allow reuse of composable, pre-packaged applications, and seamless deployment in target environments.

   Update: no additional packages have been created specifically for the use cases. Hadoop HDFS has already been created and Apache Spark is being investigated.

5. **Multiple runtime environments.** Besides Java we are also interested in other runtime environments, such as Node.js, Go and Python. Node.js has already been tested, but not provided as an application package. Other environments are going to be evaluated and integrated accordingly. This requirement has not been expressed by any of the use cases. However, following discussions with potential external stakeholders it is vital that they are supported with little or no modifications required to user's applications. Very minor and mostly boilerplate changes should be required at most.

   Update: as described above, LTS versions of Node.js are now available as pre-built packages. Golang support for OSv has been tested, however not brought to a production-ready package so it was not made available as a package yet.

We will continue to update existing and create new packages as they become available or interesting enough for this line of work. Main focus will continue to be on the packages related to the use cases of the MIKELANGELO project. However, to improve chances for exploitation and dissemination, other packages may be used as well.

6 vRDMA - Virtual RDMA

6.1 Introduction

The goal of the virtual RDMA is to provide better communication performance between VMs and high flexibility for the virtualization environment, by using a paravirtualized RDMA device, virtio-rdma, based on the virtio standard. To fulfill the requirements from the MIKELANGELO project and its use cases, we introduced the new vRDMA design with three prototypes for different scenarios, which has been initially described in D2.13[38] (The first sKVM hypervisor architecture) and D2.16[39] (The First OSv Guest Operating System MIKELANGELO Architecture), including the background and challenges of RDMA virtualization. While D2.13 and D2.16 introduced the hypervisor and guest architecture respectively, we will cover them both in this deliverable.

The overall design aims to provide solutions to support different types of guest applications with RDMA virtualization, allowing these applications to exploit benefits of virtio-rdma without any changes in the application code itself. Within the scope of MIKELANGELO project, we support guest applications including linux and OSv. Both socket and RDMA verbs are targeted in the design to support cloud and HPC applications. A list of supported RDMA verbs has been presented in D2.20[40] (The intermediate MIKELANGELO architecture).

Three design prototypes providing RDMA features through a paravirtualization architecture were proposed based on the requirements of different applications. These three choices build an evolutionary development and integration path as the result of iterative design evaluations and reviews. Moving towards the end of the project, we ensure that these prototypes will meet the requirements and objectives that were proposed initially, supporting virtual RDMA in both InfiniBand and RoCE mode. Moreover, we describe the recommendations for future work. To accommodate future market demands, advanced features including the ability for live migration, QoS, VM isolation and security could be of research interests. In the next few sections, we will recapitulate the three basic prototypes, and show an upgraded overall architecture, as well as several consideration points for other new features based on this design.

6.2 Design Prototypes

6.2.1 Prototype I

Design prototype I is based on several open source packages, which can be directly used and integrated. It supports the sockets API for the guest applications. The data is communicated through virtio-net and vhost-user, and meanwhile network API calls are translated into RDMA verb calls by the DPDK rNIC PMD (Poll Mode Driver)[41]. With help of this PMD driver, the

communication data are sent in raw ethernet format without further RDMA interaction. For prototype I, the real communications are performed in the backend driver, and every single operation requires switching between the guest and the host many times, e.g. forwarding control commands and performing buffer operations. D4.1 has shown the performance of prototype I, and the peak number achieved 25% of the bare metal performance. While this number is very low when compared to the overall throughput of the RDMA device itself it is an improvement over standard virtual Ethernet networking.

6.2.2 Prototype II

Design prototype II supports the verbs API on the guest, and it is the most direct and efficient solution of RDMA virtualization among the three prototypes. On the guest side, a new virtio-rdma module has been developed, which manages the RDMA memory directly and send RDMA commands and guest memory addresses to the host through hypercalls. On the host side, the corresponding vhost-rdma driver receives the hypercall commands. It processes the hypercall parameters by first mapping the guest memory addresses to the host physical addresses, then perform the corresponding RDMA action by calling the kernel verbs. By the means of maintaining the mapping between the guest and host memory addresses, there is no need to communicate any RDMA buffer between guest and host, as both guest and host can access the RDMA buffer directly. Only the hypercalls, i.e. formatted RDMA commands, are transferred between the guest and host, which introduces additional overhead. However, due to the host kernel bypassing and the batch processing ability of RDMA devices, i.e. accumulated numbers of send or receive requests can be processed by a single RDMA action (single hypercall in the virtualization case), the overhead of running prototype II is very small. The shown peak performance in D3.2 has achieved 98% of the bare metal performance.

6.2.3 Prototype III

Derived and extended from prototype I and II, design prototype III supports not only socket API but also reuses prototype II as the fast communication stack to replace the traditional TCP/IP stack, thus minimize the overhead.

The socket calls from the guest application are firstly converted to RDMA verbs by *virtio-rdma* and passed to the backend driver. The frontend driver will map the communication resources into RDMA memory regions. This will allow the frontend driver to actively poll the completion event and avoid sending events between the guest and the host, which is expected to show a significant improvement in performance comparing to virtio-net.

The socket-to-RDMA translation is done by *rsocket[42]* from *librdmacm[43]*, which implements a communication manager for wrapping and matching the functions of corresponding socket calls into RDMA verb calls. When using rsocket, we do not introduce

extra overhead into existing applications, instead, we replace the kernel socket library by preloading the rsocket library before loading the application into OSv. Consequently further socket calls will be redirected to use the *rsocket* functions, which are compatible with the standard socket API.

Prototype III has been designed also as an update for prototype II with additional features, including shared memory for intra-host inter-VM communication, which is based on the integration of ivshmem frontend driver in OSv.

The ivshmem support allows memory sharing between VMs that are on the same hosts. The virtual rdma driver is able to manipulate the ivshmem functionality, in order to share required parts of the RDMA memory to other VMs on the same host.

When the VM starts, a shared memory region which is large enough for storing the communication buffers is preallocated, The virtio-rdma frontend driver is then able to store and delete data from that shared memory region. When the communication is between VMs on the same host, virtio-rdma will do a memcpy between the source and destination buffers based on the virtual memory address information, and then update the communication status in the Completion Queues of the VMs. This is done directly in the guest, and no hypercall to the host is necessary.

6.3 Overall Architecture

Based on the design of prototype II and III, we are able to merge these two prototypes into one overall architecture, as shown in Figure 7. Prototype I is not included into the final architecture due to its inefficiency and low performance.

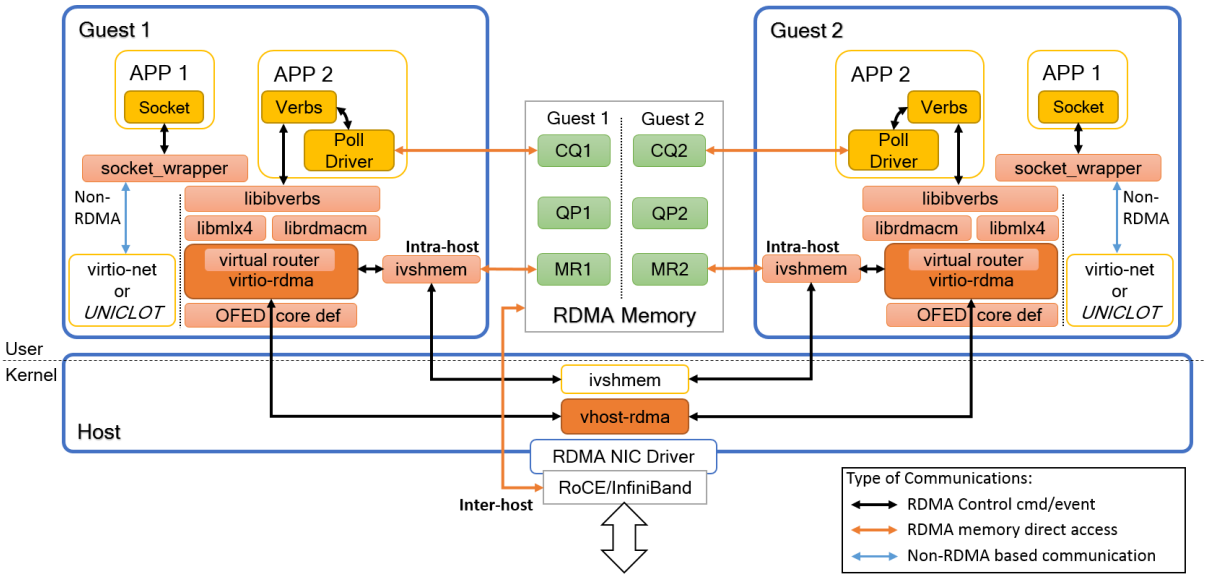

Figure 7. Overall Architecture of the Virtual RDMA design prototype II and III

The architecture shows a hybrid mode of intra-host and inter-host communications. The GUIDs, which are identical ids of each RDMA NIC, are compared between the source and destination peers, in order to identify whether they are on the same host. The virtio-rdma frontend driver maintains a list of the GUIDs of the local host, and checks if the communication peers' GUIDs, appended with the remote and source QPs, are matched in the list.

If the VMs are not on the same host, then the RDMA stack of prototype II will be directly used, i.e. virtio-rdma sends hypercalls to host to change the RDMA NIC state and start the communication.

If the VMs are on the same host, then intra-host communication is enabled and ivshmem will take care of the actual data transmission by copying source buffer to the destination instead of sending the hypercall to the host and changing the Queue Pair status of the real RDMA hardware. The Completion Queue will be updated for the finished communication by virtio-rdma driver, after the communication buffer is copied. This will reduce the overhead of passing hypercalls between guest and host. But the memory copy is not avoidable or replaceable by remapping the source and destination virtual address due to the uncertainty of the RDMA memory usage by the application, i.e. we do not know when the application will reuse the same communication or when it will be released.

The support of socket API is managed by the socket_wrapper module, which detects if the virtual RDMA device is available and if the rsocket library is loaded into OSv. If the virtual RDMA is ready for use, all further socket calls will be redirected to rsocket library, and converted to corresponding RDMA verb calls. The stack of prototype II is then used for processing the converted RDMA verb calls. However, if the virtual RDMA is not available, which means the rsocket is also not loaded, then the traditional virtio-net will be still used for socket communication. In this case, UNCLOT (section 7) may be also used for intra-host communication as a fast path.

Another advantage of this design is that, for example in Figure 7, the APP1 is able to communicate to APP2 on the other host no matter if they are on the same host or not, as long as they all have virtual RDMA drivers available. Because the application is not sensible to the underlying RDMA stack, and both socket and RDMA verbs API will be processed in the same stack and data structures. More precisely, all the RDMA memory regions, e.g. CQ, MR and QP, are registered and recorded on the host via hypercalls, and they are compatible for both intra- and inter-host communications. The only difference is that, for intra-host communication, virtio-rdma driver will update the RDMA memory, and for inter-host communication, the RDMA NIC will do the same job.

6.4 Further Consideration of Advanced Features

The research and development on vRDMA that has been undertaken for this project are leading to a number of topics on which further research would be beneficial. In this section, we list a group of advanced features that may be valuable for further development beyond the scope of this project. These features include live migration, quality of service (QoS), VM isolation, hardware compatibility and security.

For live migration with vRDMA enabled VMs, there are a few prerequisites: the RDMA NIC has to be in an idle and stable state before starting the migration, i.e. the previous work request is finished and stopped to continue to process the next; all the mapped guest virtual addresses have to be remapped on the new host; the QP number in the guest has to be implemented as a virtual one, i.e. not as incremental as the host QP, but is rather mapped; then the entire VM, the RDMA memory, and the remapped virtual address table can be migrated to the new host.

For QoS, isolation and security, additional layers in virtio-rdma and vhost-rdma may be implemented to intensively process the work requests and apply user defined or preconfigured processing rules before sending them to the RDMA NIC.

In order to support new hardware, e.g. iWARP card, the solution is basically to add additional plugins (driver modules) to libibverbs library and virtio-rdma, similar to libmlx4.so and mlx4_core.ko modules. Finding or developing a common virtual RDMA interface that supports all RDMA NICs is not straightforward.

Due to the fact that shared memory is used as a fast channel for the intra-host inter-VM communication, data protection and isolation still need to be considered, Additional firewall mechanism may be designed and implemented with virtio-rdma to increase the security level of the vRDMA communication. Access permissions can be applied based on different rules, for example, the processes from the same protect domain may have access permission of the shared memory regions belonging to this group. Or even dynamically controlled access permission, for example, only the memory that contains communication data will be accessed by the communication peers during the period of the single RDMA poll function. The actual implementation will highly depend on how ivshmem device is implemented and extended to isolate the shared memory among the VMs.

7 UNCLOT - UNikernel Cross Level cOmmunication opTimisation

7.1 Introduction

Over the course of the MIKELANGELO project, different attempts have been made to improve the performance of applications running inside OSv operating system. In the context of this work, shorter latency and higher throughput are the main metrics of the network performance. MIKELANGELO project has already made significant improvements in supporting functional requirements of the workloads running in OSv. This has been demonstrated by the Cancellous bones and Aerodynamics use cases that have been successfully executed in OSv early in the project. On the other hand, OSv is still lacking performance improvements that would allow it to be used in the high performance computing domain. One of its most limiting factors is its inability to interpret and use NUMA (Non-uniform Memory Access) of modern processors. Because OSv equally treats memory from different NUMA domains, it is unable to optimise CPU and memory pinning the way general purpose operating systems do.

One way of addressing this problem is the adoption of Seastar framework that handles CPU and memory pinning internally and properly handles this limitation. However, this requires a complete rewrite of the application logic which is typically impractical or even impossible for large HPC simulation code bases. To this end, this section introduces a novel communication path for colocated virtual machines, i.e. VMs running on the same physical host. With the support of the hypervisor (KVM) the operating system (OSv) is able to choose the optimal communication path between a pair of VMs. The default mode of operation is to use the standard TCP/IP network stack. However, when VMs residing on the same host start communicating, standard TCP/IP stack is replaced with direct access to the shared memory pool initialised by the hypervisor. This will allow the cloud and HPC management layer to distribute VMs optimally between NUMA nodes. This section explains in detail the design and the findings based on the prototype implementation.

Another driver for investigating addition approach to optimising intra-host communication is the raise of so called serverless operations. Serverless architectures are becoming widely adopted by all major public cloud providers (Amazon, Google and Microsoft) in a form of so called cloud functions (Amazon Lambda, Google Cloud Functions and Azure Cloud Functions). This has been made possible with the tremendous technical advances in the container technology and container management layer planes. Cloud functions are stateless services communicating with each other and other application components over established channels using standard APIs and contracts. The very nature of these services allows for

seamless optimisation of workloads, service placement management and reuse, as well as fine-grained control over the billable metrics in the infrastructure.

OSv-based unikernels are ideal for the implementation of such serverless architectures because they are extremely lightweight, boot extremely fast and support running most of the existing workloads, i.e. they mimic the advantages of containers. All the benefits of the hypervisor are furthermore preserved (for example, support for live migration, high security, etc.). In light of the aforementioned facts, optimising the networking stack for collocated unikernels will allow for an even better resource utilisation because the interrelated services will be able to bypass the complete network stack when sharing data between each other.

7.1.1 Background

The serverless operations described in the introduction are an extreme example of separating large monolithic applications into smaller services specialising for specific business logic. Even when not targeting serverless deployments, modern applications are typically split into multiple weakly coupled services, which communicate via IP network. Figure 8 presents typical communication paths used by applications running inside virtual machines. In this case, the server application, e.g., database, is running in VM-1, while client applications, e.g., data processing services, run in VM-2 and VM-3. Virtual machines VM-1 and VM-2 are collocated on the same, while VM-3 is placed on a different host.

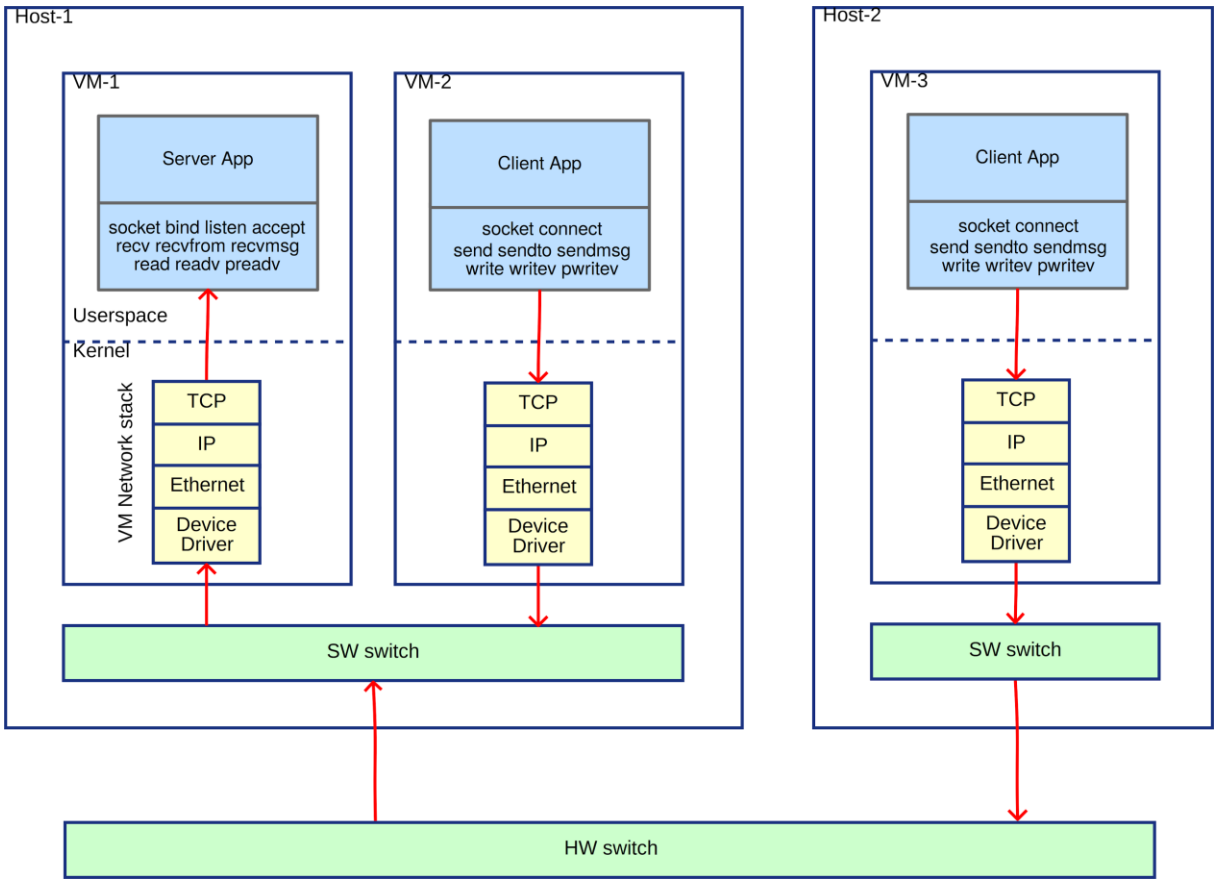

Figure 8. Traditional communication paths used by applications running inside virtual machines

When client application (VM-3) runs on a different host than server application (VM-1) data traverses many layers. First, the data goes through the entire network stack of the guest operating system and sent to the software switch in the client virtualization host. From there, the packet is sent to the physical network. On the other side, the virtualization host of the server receives the packet from the physical network and forwards it to the software switch. The packet finally goes through Server's network stack and is delivered to the server application.

When server and client machines are collocated on the same virtualization host (VM-2 and VM-1), the whole process is greatly simplified. In this case, the physical network in not used because the virtual switch in the host automatically detects collocated virtual machine and delivers the packet directly into the network stack of the target VM. This allows higher throughput at lower resource usage.

The work proposed in this section is expected to further improve the latter case, i.e. the intra-host TCP/IP-based communication by eliminating most of the IP processing. This is particularly important because the TCP protocol was designed on top of unreliable, lossy, packet-based network. As such it implements retransmission on errors, acknowledge and

automatic window scaling in order to guarantee reliable and lossless data stream. However, it is safe to assume that communication between collocated VMs is highly reliable.

To eliminate most of the aforementioned processing overhead of the network stack we try to implement TCP/IP like communication based on shared memory. Instead of sending actual IP packets via software switch, the source VM puts data into memory shared between VMs on the same host. Target VM can then directly access the data. Multiple approaches to bypassing network stack have been analysed by Ren et. al.[44]. The approaches are classified according to the layer of the network stack at which the bypass is built into. In general, the higher in the application/network stack the bypass is implemented higher throughput and lower latency can be achieved because more layers of the networking stack are avoided. On the other hand, the higher the layer at which the bypass is implemented, the less transparency is preserved. For example, replacing all `send()` system calls with a simple `memcpy()` will result in the fast communication between two VMs. However, this performance benefit will result in significant additional constraints for the said application. It will require that every part of the application that needs to communicate with other VMs makes a change in the source code. Furthermore, it will prevent distributed deployment of such application because of memory-only communication. This thus requires additional development and debugging effort for each application which should benefit from modifications.

7.2 Design Considerations

Our main interest lies in massively parallel workloads on top of Open MPI and microservices. MPI-based workloads typically use shared memory interprocess communication between parallel workers running on the same host and Infiniband or similar interconnects when deployed in HPC environment. Both of these techniques reduce latency and improve the overall throughput of the communication. However, when workloads are deployed in virtual machines these mechanisms are no longer supported out of the box. While vRDMA brings Infiniband into virtual machines by providing paravirtual device drivers (Linux and OSv) inter-VM communication between virtual machines running on the same host has not been addressed in OSv. Standard TCP/IP networking is used when VMs from the same host need to share data. This adds a significant overhead due to the fact that communication must adhere the protocols that have been designed for unreliable media. When it comes to microservice and serverless architectures, this problem escalates even more because services that need to collaborate to fulfill the tasks typically reside in different container or virtual machine.

Both of the above types of workloads typically use TCP (Transmission Control Protocol) transmission protocol. This allows us to focus only on bypassing the TCP protocol using

shared memory approach, while preserving UDP (User Datagram Protocol) or any other protocol intact.

For Open MPI, we intend to use bypassed TCP to overcome current OSv limitation regarding NUMA memory pinning support. When used on a server with multiple NUMA nodes (e.g. with multiple physical CPU sockets) Open MPI uses memory pinning to ensure that CPU cores use memory with lowest possible latency. MPI processes within one server communicate via shared memory based BTL (Byte Transport Layer), and MPI processes on different servers communicate via dedicated interconnect (infiniband) BTL or via TCP/IP BTL.

We have already mentioned in the introduction that OSv is currently not NUMA memory aware. OSv VM with multiple NUMA nodes correctly pins MPI threads to CPU cores, however the memory for a specific thread is allocated from the memory pool of the entire virtual machine. Memory of a thread pinned to one NUMA node can thus be allocated on a different (foreign) NUMA node resulting in deteriorated performance (according to our measurements, this is typically about 30%). To overcome this, we intend to exploit the network stack bypass to deploy one OSv VM per each NUMA node. The libvirt domain specification used to configure virtual machines will ensure that each VM will get memory from a single NUMA node. Just as before, MPI processes running in the same VM will communicate between each other via shared memory BTL[45]. Processes running on different physical hosts will also use the same channel (either via dedicated interconnect or via TCP/IP). The main difference will be for the processes running in VMs on the same physical hosts. These will still use the TCP/IP BTL provided by Open MPI, however, OSv with the support of the hypervisor will ensure that the channel is properly bypassed exploiting the memory shared between VMs. The approach is expected to improve the performance of MPI computing in OSv by improving the communication performance itself.

As discussed in the introduction to this section, the higher in the communication stack the shared memory is introduced the higher the performance gain is expected. In case of Open MPI this means that optimal performance would be achieved by replacing communication channel either in the application (for example OpenFOAM) or by introducing an additional BTL component. However, this would result in loss of transparency and would only work for a single workload. While this would be sufficient for the use cases of the MIKELANGELO project, we decided to make the bypass one layer below, i.e. built directly into network stack of the OSv kernel. The existing networking API (POSIX socket API) is fully preserved offering existing applications ability to use the new communication channel when available and in transparent way.

The implementation of this bypass requires changes in the OSv implementation ensuring that standard functions like `socket()`, `bind()`, `listen()`, `connect()`, `send()`, `recv()` are reimplemented in order to support shared memory as a communication backend.

The following diagram shows a high-level design. Neither server nor client application needs to know about the change in the communication channel. The aforementioned API functions transparently choose either standard network stack or shared memory depending on the IP address of packet's destination.

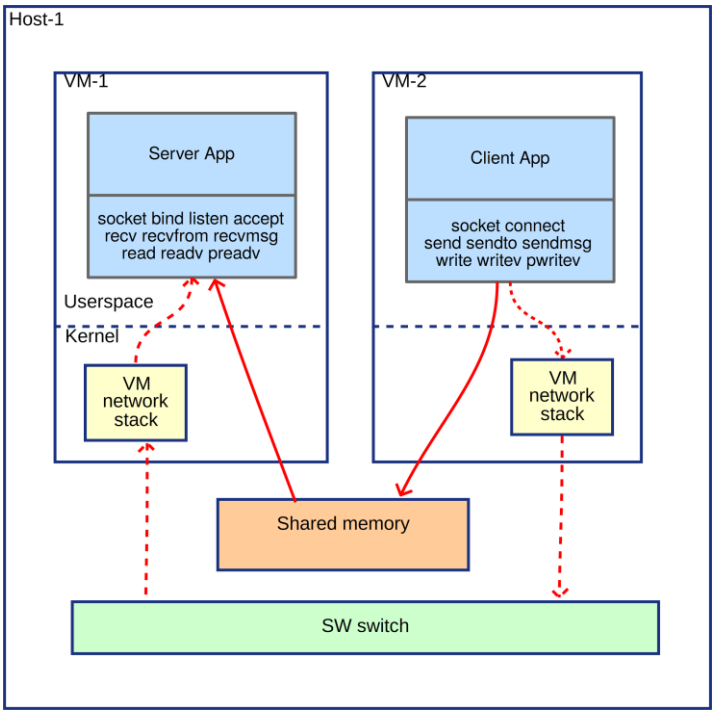

Figure 9. High-level architecture of UNCLOT.

The current IP bypass will be implemented for KVM virtualization only. The KVM/QEMU ivshmem[46] will be used to established shared memory between VMs on the same host. Further to basic read/write access to shared memory, notification mechanism will have to be implemented. This will be used by the sending VMs to notify the receiving ones of the availability of new data. Different types of notification system are going to be examined (e.g. IPI interrupts, ivshmem interrupts or simple polling).

The detection of colocated VMs should be done automatically. Colocation detection assumes that the IP is used as unique identifier of a given virtual machine. A central registry of all VMs running on the same host and their IP addresses) has to be integrated allowing the host (or VMs on that host) to check whether the IP belongs to a VM on the same host. This registry needs to be updated by Libvirt start/stop hooks as well as some other changes in the VM networking. The mechanisms are thus built into the network stack of the underlying

operating system which, with the support of the hypervisor, automatically identifies and configures networking channels between VMs depending on their location.

An important consequence of this is that the live migration of the all or just a subset of virtual machines running on a single node is not prevented by the introduction of these mechanisms. Similarly to vRDMA, certain prerequisites are required in order to ensure smooth migration:

- Disable writing to the socket
- Wait for old data in socket to get consumed, or, alternatively, copy data to normal TCP/IP socket buffer
- Disable bypass for that socket and restart normal TCP/IP; reuse the same file descriptor
- Continue with the live migration

After a successful migration, the following steps are required to resume with the proper functioning of the networking stack:

- Identify sockets suitable for bypass
- Temporarily stop writing to that socket
- Wait for old data in socket to be consumed by the receiver, or, alternatively, copy data to the bypassed socket buffer
- Enable bypass for the corresponding sockets

For IP bypassed sockets, the IOCTLs (Input/Output Control) for selecting blocking/non-blocking mode, enabling/disabling Nagle's algorithm, etc., should also work. Some of the socket options specified by the POSIX API do not have much impact when used in shared memory. For example, the Nagle's algorithm[47] was developed in 1984 to improve performance by reducing the number of packets sent over the network - a feature that is not mandatory when communicating over shared memory. However, other options have to be implemented in a transparent way. This will allow existing applications to rely on the same contract offered by the API as they are using right now. For example, reading from a blocking or non-blocking socket behaves differently when there are no data to read, and existing applications were written with that in mind.

The initial testing and benchmarking will be done using the common synthetic tools like iperf[48], netperf[49] and Apache benchmark[50]. Additional synthetic tests will be implemented for functional testing, ensuring that specific mechanisms of the network bypass are properly supported and compliant with the POSIX socket API. This should cover basic socket API functions (`socket`, `bind`, `listen`, `accept`, `connect`, `send`, `write`, `recv`, `read` etc) and `ioctl()` behaviour. As many new applications use select/poll/epoll and event based processing, behaviour of those calls should be checked too. Final evaluation will be performed using real world workloads such as OpenFOAM.

8 Snap Telemetry

8.1 Introduction

As documented in D5.2[51], Intermediate report on the Integration of sKVM and OSv with Cloud and HPC, Snap Telemetry [http://snap-telemetry.io/] is an open-source telemetry framework specifically designed to help simplify the gathering and processing of rich metrics within a data center. With deeper instrumentation and analysis of such infrastructure and hosted applications, more subtle measurements of performance can be gathered, and more efficiencies can be realised. The architecture of snap has not altered significantly since the initial open-source release during Year 1 of MIKELANGELO.

8.2 Architecture

As an extensible and open telemetry system, snap aims to provide a convenient and highly scalable framework from which arbitrary metric-collecting systems, analytics frameworks, and data stores can be leveraged. Thus, full stack monitoring is achievable, allowing data from hardware, out-of-band sources such as Node Manager, DCIM and IPMI to be analysed together with data from the host operating systems, hypervisors, guest operating systems, middleware and hosted applications and services. These metrics can be transformed, filtered or processed using any external tool locally, before being published to any destinations, be they local or remote.

Snap Telemetry is organised into several core components as illustrated in the following figure.

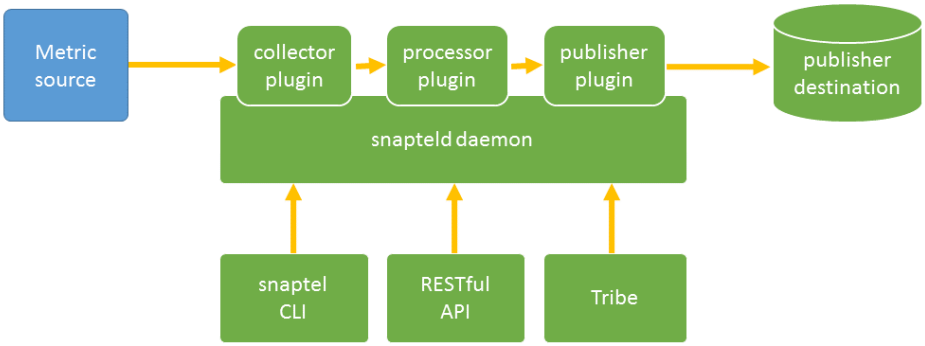

Figure 10. Core components of the snap architecture.

snapteld is a daemon that runs on nodes that collect, process and publish telemetry information. snapteld may collect the information from the local node (localhost), or the information may be captured from remote nodes, over the network. The latter allows out-of-band metrics to be captured from systems that are in the powered down state. The collection, processing and publishing of telemetry information is done through a flexible and

dynamic plugin architecture that is described later in this section. This daemon also schedules the tasks that define what data is collected, how it is processed and published.

snaptel CLI is a command line interface that allows snap to be managed. It allows snap metrics, plugins and specific monitoring instructions known as tasks to be queried and manipulated as required.

The snapteld daemon can also be manipulated via a **RESTful API**. This gives infrastructure administrators and devops teams additional flexibility for integration with any third-party or bespoke management tools and scripts.

snap also boasts a **Tribe** component which allows nodes to be organised into arbitrary (and multiple) tribes. Management commands can be addressed to a particular tribe of machines. The snap nodes pass these instructions to other members of the tribe in a peer-to-peer fashion. This highly scalable approach allows the snap infrastructure for a very large deployment to be managed via simple individual commands.

8.3 Plugins

snap has a flexible plugin architecture. To facilitate administration plugins can be versioned and signed. Digitally-signed plugin verification is switched on by default, API invocation is through SSL, and all payloads transferred between components can be encrypted. Three types of plugins are supported:

Collector plugins allow data to be fed into snap from a particular source. The metrics exposed by a collector plugin are added to a dynamically generated catalogue of all available metrics. A collector plugin may be engineered to capture data from any source including hardware, operating system, hypervisor, or application source. The data may come from out-of-band systems such as baseboard management controllers - essentially small dedicated always-on CPUs that allow computers to be managed remotely via a dedicated network channel - even if the main system is powered off. The data may come from data-center utility systems – or indeed sources outside the data-center.

Processor plugins allow snap telemetry data to be queried and manipulated before being transferred. A processor plugin can be used to encrypt the data, or perhaps convert the data from one format to another. Data can be cached or filtered or transformed – e.g. into rolling averages.

Publisher plugins are used to direct telemetry data to a back-end system. Data could be published to a database, to a bus, or directly to an analytics platform. Destinations may be open source or proprietary.

Available plugins are loaded into the snap framework dynamically, and exposed functionality is available without needing to restart any service or node. Once plugins are loaded into the local snap daemon, specific workflows called Tasks can be defined to detail what data is gathered where, and how it is processed and shared. The currently available plugins are listed in the plugin catalogue[52]. At the time of writing (June 2017) there are 96 open source snap plugins available from the main snap repository on GitHub. These include 63 snap collector plugins, 9 snap processor plugins, and 24 snap publisher plugins. Twelve of these 96 plugins have been contributed and open-sourced from the MIKELANGELO project. Other members of the open-source community have already started to build on these contributions and open-source their incremental enhancements. Plugins that are not yet relevant to the broad snap user base are typically open-sourced elsewhere. MIKELANGELO has open-sourced two collector plugins which fall into this category via the MIKELANGELO GitHub repository. These collect data from IOcm and vRDMA - systems which for now are only relevant to snap users that have installed the MIKELANGELO IOcm and vRDMA solutions.

The following table lists the plugins that have been developed and open-sourced by the MIKELANGELO project to date.

Plugin Type	Plugin Name	Plugin Description	2015	2016	2017	GitHub
Collector	Libvirt	Captures data from libvirt	x			snap
	OSv	Collects from OSv	x			snap
	MongoDB	Captures data from MongoDB		x		snap
	vRDMA	Captures data from vRDMA process		x		MIKELANGELO
	OpenFOAM	Captures data from OpenFOAM		x		snap
	yarn	Captures data from yarn scheduler		x		snap
	schedstat	Captures data from Linux scheduler		x		snap
	SCSI	Captures data from SCSI devices		x		snap

Plugin Type	Plugin Name	Plugin Description	2015	2016	2017	GitHub
	USE	Captures Utilisation, Saturation and Errors information from various system components		x		snap
	IOcm	Captures IOcm data		x		MIKELANGELO
	KVM	Captures KVM data			x	snap
Processor	Tag	Allows metrics to be tagged		x		snap
	Anomaly Detection	Dynamically scales resolution of telemetry to reduce system overhead		x		snap
Publisher	PostgreSQL	Writes to a PostgreSQL [21] database	x			snap

Of particular note, the Anomaly Detection plugin implements the Tukey Method to automatically reduce the resolution of published data for metrics while they are stable, significantly reducing overall telemetry system overhead without reducing statistical usefulness of the data. The USE plugin captures high-level metrics: Utilisation and Saturation data for system components including CPU, network and storage. These metrics are designed to allow very fast identification of resource bottlenecks across a cloud-scale data centre deployment.

Based on feedback from the use cases, additional plugins are currently being developed by INTEL as part of the MIKELANGELO project. These will deliver the ability to collect data from OpenVSwitch, OpenDaylight as well as profile the execution of functions.

9 SCAM - Side Channel Attack Monitoring and Mitigation

Cache side-channel attacks allow a malicious VM to extract secret information such as long term secret keys from co-tenants. Cache side-channel attacks are one of the only attack vectors that is specific to co-tenancy in virtualized environments. SCAM (Side Channel Attack Monitoring and Mitigation) is a module within sKVM, which is used for monitoring, profiling, and mitigating such attacks. As part of our preparatory work for SCAM, we have implemented a full L3 cache side-channel attack that serves as a benchmark for testing and evaluating the SCAM module.

SCAM runs in user space and has two layers. The upper layer includes a control function, configuration data and a user dashboard. The lower layer includes the three main functions of monitoring, profiling and mitigation.

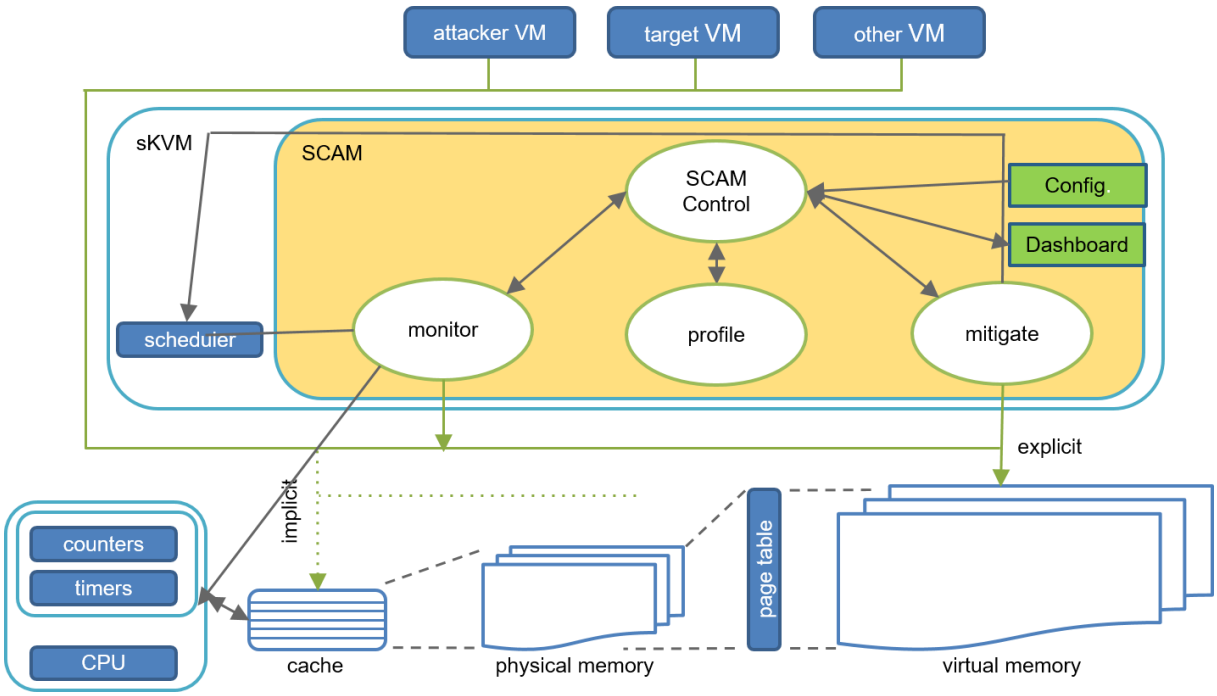

Figure 11. SCAM architecture.

The configuration includes instructions on which sub-modules to run (e.g. whether to execute the performance intensive mitigation module) and the hardware properties of the L3 cache, which are a function of the CPU type.

The control function uses the cache information for its initialization which includes cache mapping and target reconnaissance. Cache mapping refers to allocating a block of memory that covers the whole cache and mapping all its addresses to cache sets. Cache mapping also determines the timing values for cache hits and misses. Target reconnaissance finds possible

cache sets that may interest an attacker for a given target. All this data is necessary for the mitigation sub-module of SCAM.

Following initialization the control function runs the lightweight monitoring module in a loop and uses profiling to determine whether an attack is taking place. If that is the case then the control function activates the mitigation module and continues the monitoring activity. If the attack stops then mitigation is also stopped.

SCAM provides two types of data to a user dashboard, raw measurements by the monitoring module and decisions by the profiling module on whether these measurements constitute an attack.

9.1 Components Overview

The monitoring function of SCAM collects data on the pattern of cache access of virtual machines running over (s)KVM and uses this data to profile VMs and evaluate the likelihood that a monitored VM is executing a cache side-channel attack. This monitoring activity is based on retrieving information from a number of system counters via PAPI (Performance Application Programming Interface) in user-space. The complete list of counters is described in D3.4, but the most important for SCAM are the L3 hit and access counters which collect information separately on each VM.

A cache side-channel attack learns information on the target by repeatedly accessing the same cache sets that the target uses and analyzing the resulting hits and misses. That activity implies both more L3 hits and misses than most applications have and in addition a noticeable correlation between the activity of the attacker and the activity of the target. SCAM profiling detects these properties of the attack and reports on attacks with high accuracy. The current monitoring and profiling modules improve significantly over the module made available last year by its ability to identify an attacker *in the process of extracting the key* instead of only identifying the intensive procedure of cache-set mapping, which can be done over a long period of time or even mostly offline.

The mitigation module of SCAM disrupts the measurements the attacker makes by adding noise to the cache. The attacker can only count the number of cache-lines that it introduced to a specific set. Therefore, the attacker cannot distinguish between the data added by the target to the cache and the noise. An important tradeoff in this “Noisification” method is between the number of “vulnerable” cache sets and the amount of noise that SCAM can add to a set before the attacker completes one measurement. One way to improve the tradeoff is to let SCAM mitigation run with only the target VM active to determine which of the cache sets are activated by the target and may interest an attacker. This mode of operation requires

some a-priori cooperation with the target and is mostly useful for customers who wish to pay for the added protection of SCAM.

10 MCM - MIKELANGELO Cloud Manager

The MIKELANGELO Cloud Manager (MCM) is a cloud service for live resource management. The service integrates seamlessly with the cloud middleware, in our case OpenStack. The service offers an environment to execute resource management strategies. Resource management entails control of the physical and virtual infrastructure. By integrating tightly with monitoring data, MCM strategies form a closed-loop between resource control and monitoring.

10.1 The need for closed-loop resource management in the cloud

The special importance of MCM for the MIKELANGELO cloud stack lies in its potential for cross-layer optimization. By combining a multitude of monitoring collectors from snap and the control features of the MIKELANGELO components MCM can identify and exploit inefficiencies. MCM itself provides the execution environment for resource management algorithms. The algorithms themselves can be researched more conveniently than ever before by combining MCM with Scotty (Section 11). MCM has been created out of the need to research live-resource management across all deployment layers in the cloud. In MIKELANGELO, MCM is integrated with OSmod, so that IOcm and SCAM can be controlled comfortably within MCM strategies. In addition, MCM directly integrates with Snap to retrieve its monitoring data.

10.2 Design considerations for MCM

The design and implementation of MCM revolve around general applicability, re-usability, and quick implementation-validation loops for research. MCM is written in Python and is self-contained regarding most of its functionality. All communication with external components happens via RESTful interfaces, which allows for easy extensibility. The two main components for interaction are a source for monitoring data and a sink for control commands in the cloud middleware.

Figure 12 shows the architecture of MCM. The lower part shows the expected model for the cloud architecture. The model is quite generic and has only few assumptions. Compute nodes are expected to run virtual machines, which are in turn controlled by a control node. The control node needs to provide a central endpoint for control commands. Furthermore, all nodes should collect monitoring data, which should be made available via a central endpoint. In the concrete case of our cloud testbed the deployment builds on OpenStack, with OpenStack Nova as control endpoint, snap for data collection, and InfluxDB for data storage.

MCM itself consists of three architectural layers. The lower layer contains wrappers classes for control and monitoring commands. So far, we have developed a control wrapper for

OpenStack Nova and a monitoring wrapper for data collected by InfluxDB and Snap. The InfluxSnap-wrapper in turn is based on a common base class for InfluxDB. While the abstract InfluxDB-wrapper manages queries to InfluxDB, the concrete wrappers for Snap and collectd know how to deal with the data structures collected by the respective frameworks. Additionally the lower layer contains configuration management, utilities, and tests with a coverage above 95% of the core code base.

The middle layer contains the main application loop, the abstract base scheduler, and customized base schedulers. The customized base scheduler comprises a control component and a monitoring component. In our example, these components are the NovaWrapper and the InfluxSnapWrapper. Concrete schedulers inherit from this customized base scheduler. As an effect the concrete schedulers can use all of the custom features provided by the customized base scheduler. As an example, the InfluxNova base scheduler can make use of custom extensions of Nova and of assume the availability of certain snap metrics. As a downside, such a scheduler will be harder to migrate to a different system. Nevertheless migration from one customized scheduler to another one would only mean that the base class needs to be changed and that custom interfaces need to be redirected or wrapped.

The top layer in MCM contains all concrete schedulers. These contain exclusively code needed for resource management, because all the boilerplate to interact with the cloud middleware is hidden in the customized base scheduler. We have implemented two test strategies and three representative strategies to validate our design and implementation. The first test scheduler communicates with both Nova and InfluxDB, but issues no control operations to the cloud middleware. The random scheduler randomly initiates live-migration of VMs to verify functionality and to obtain a baseline for other scheduling algorithms. The hotspot scheduler, vector scheduler, and CPI scheduler are based on published work by other researchers. The implementation of those scheduler serves for validation of our approach.

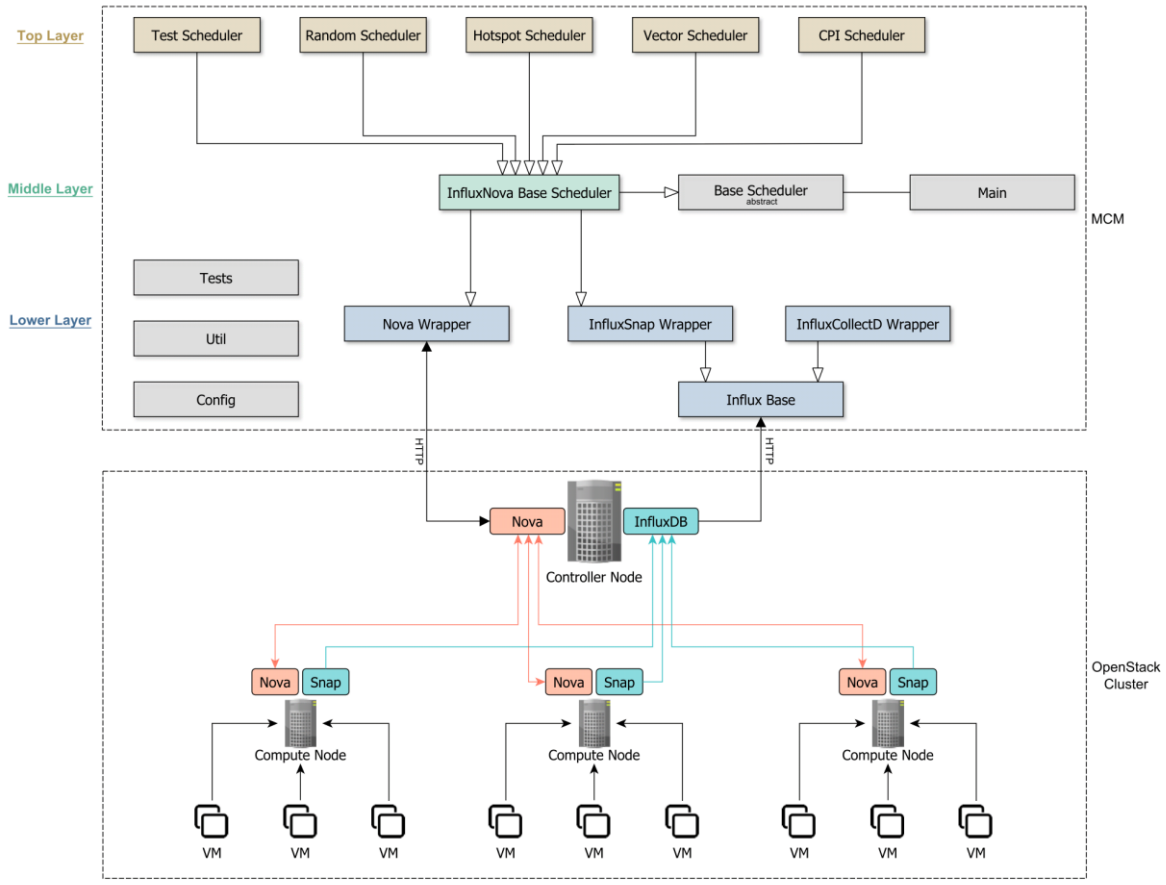

Figure 12. MCM architecture.

10.3 The integration of MCM with the MIKELANGELO cloud stack

MCM integrates with IOcm, SCAM, snap, OSmod, and Scotty within MIKELANGELO. IOcm and SCAM can be turned on and off in MCM strategies. These features will allow to analyze the performance boost for IOcm and the performance hit of SCAM with real workloads in Scotty. Furthermore, MCM's power comes from drawing data from all layers of the cloud and combining that data with vast control mechanisms across the cloud. As an example, MCM will allow to isolate or even stop a VM marked as suspicious through SCAM. MCM can stop and then migrate such a VM to a quarantine section to analyze the VMs behaviour. Such a scenario would be hard to implement without MCM. Further integration of MCM is available with Snap. The current implementation of MCM has wrappers for a snap-fed InfluxDB and for a collectd-fed InfluxDB. All concrete schedulers rely on snap's wealth of data metrics. The integrations with IOcm and SCAM are provided by OSmod's driver plugins. Thus, MCM does not use drivers for IOcm and SCAM directly, but instead it leverages OSmod's capabilities.

Further integration of Scotty is available via OSmod. OSmod allows control of the host hardware and host OS. For example, OSmod can control TurboMode (e.g. TurboBoost) activation, cache partitioning, CPU pinning, the active IO and CPU scheduler, and more

system parameters. Since OSmod features a pluggable architecture, new integrations can be implemented easily.

In the cloud layer, MCM knows how to control the cloud middleware via Nova. The currently implemented commands are migration, live-migrations, control of the VM state, and resizing of VMs.

Finally, MCM integrates with Scotty. Scotty can load user-implemented resource management strategies into experiments. The experiments then execute the corresponding scheduler and collect resulting data. This approach allows for quick development cycles and an agile approach to algorithms design with an included empirical validation step.

11 Scotty

Scotty's overall architecture is shown in Figure 14. The shown architecture is the result of multiple iterations with continual simplification to get to the essence of Scotty's use case. Scotty's high-level architecture consists of an installation of GitLab, the scotty-gateway, a resource management system, a monitoring system, and OSmod. GitLab is used as code repository for workloads and experiments. Workloads are implemented in python with various configuration files. Experiments consist of YAML-files[53] that describe the system parameters and workloads to be used for an experiment. The Scotty gateway contains a gitlab-agent that is notified about git-push commands to experiments in GitLab. The agent then calls the python-scotty package, which implements Scotty's core logic. Python-scotty receives experiments, sets the system configuration, allocates workload resources and finally runs the workloads. The resource management system allocates infrastructure resources, in which workloads can be installed.

OSmod is a component, which allows the configuration of the cloud infrastructure. Configuration means changing system parameters before experiments and even during runtime. By cloud infrastructure we mean the cloud hosts including hardware settings, the host OS, the virtual infrastructure, and even application-layer mechanisms. OSmod's architecture is shown in Figure 13. To facilitate the control of such a broad set of components, OSmod uses a AMQP-based message bus and clustering methods. The available control features come from plugins, which are managed by OSmod. Thus, OSmod provides a plugin-based, distributed architecture for infrastructure control. These features are a key enabler for Scotty and MCM. In Scotty, OSmod is used to prepare the execution environment for experiments. In MCM OSmod facilitates the control commands issued by MCM's resource management algorithms.

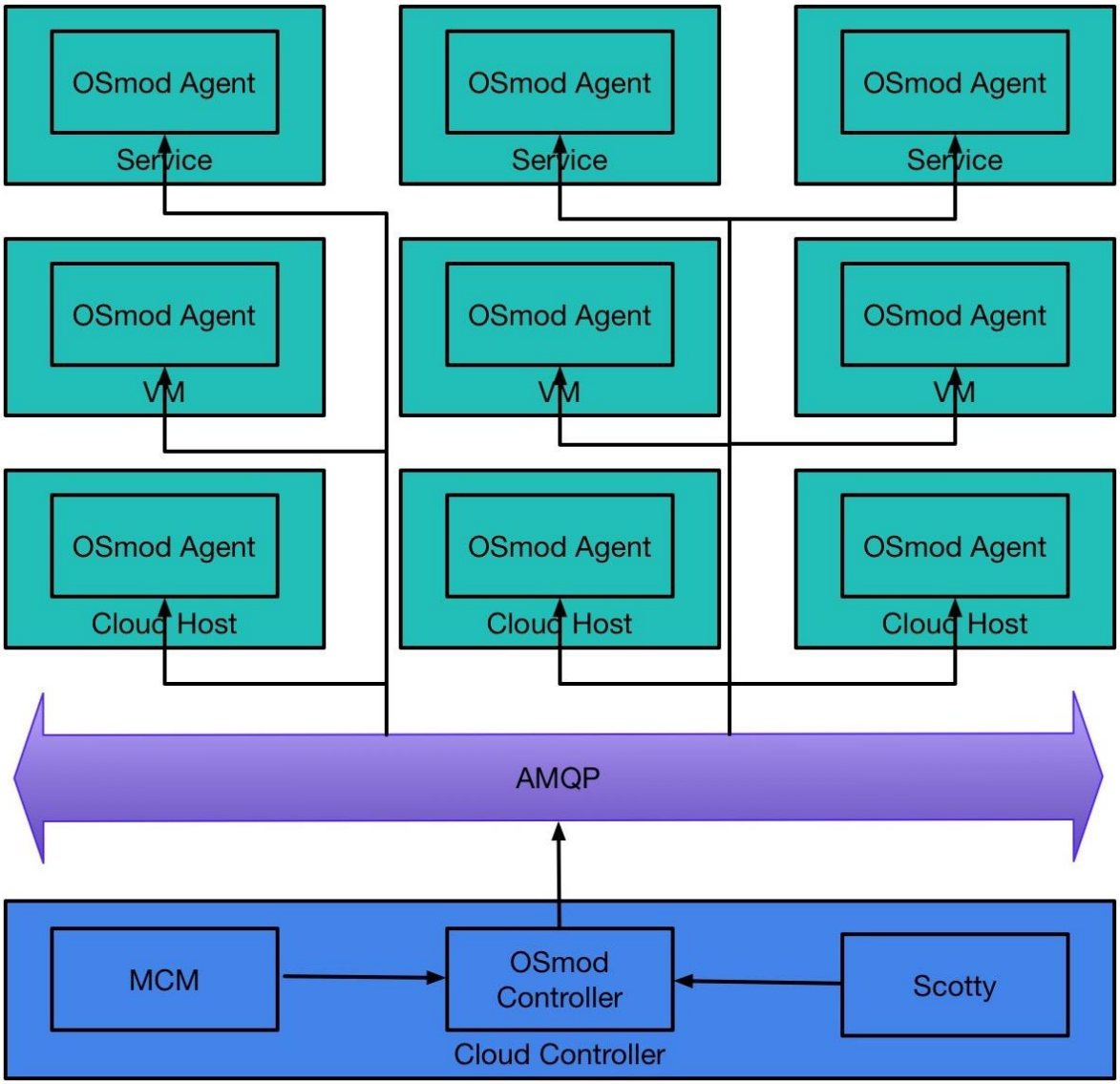

Figure 13. OSmod architecture.

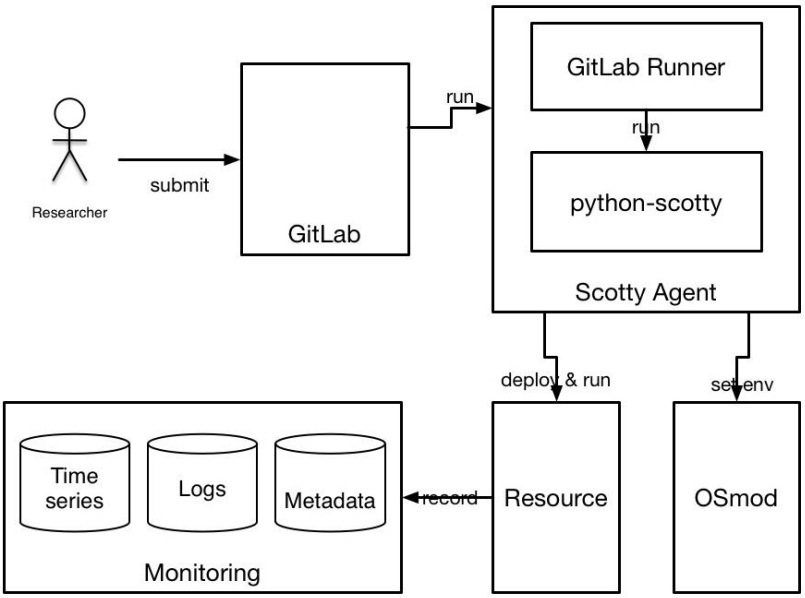

Figure 14. Scotty architecture.

As a concrete example, we use OpenStack with Heat as the resource management system. Via Heat, OpenStack provides us with VMs and VLANs. Furthermore, Heat then orchestrates the installation of the benchmarks we use as workloads. A comprehensive monitoring system collects data about experiments. The monitoring system consist of three subsystems: time series metering, metadata collection, and log collection. Time series metering builds on snap for data collection, InfluxDB [InfluxDB] for data storage, and Grafana [Grafana] for data visualization. Metadata collection builds on custom scripts that snapshot static data from hosts, and on Heat templates. In our use case metadata refers to static data about the execution environment that helps to reproduce experiments in the future. Log data is collected from virtual machines that run workloads, is processed by logstash [Logstash], redis [redis], and visualized with Kibana [Kibana].

12 Integrated Cloud infrastructure

The integrated cloud infrastructure combines most components of the MIKELANGELO stack to verify and validate the system integration.

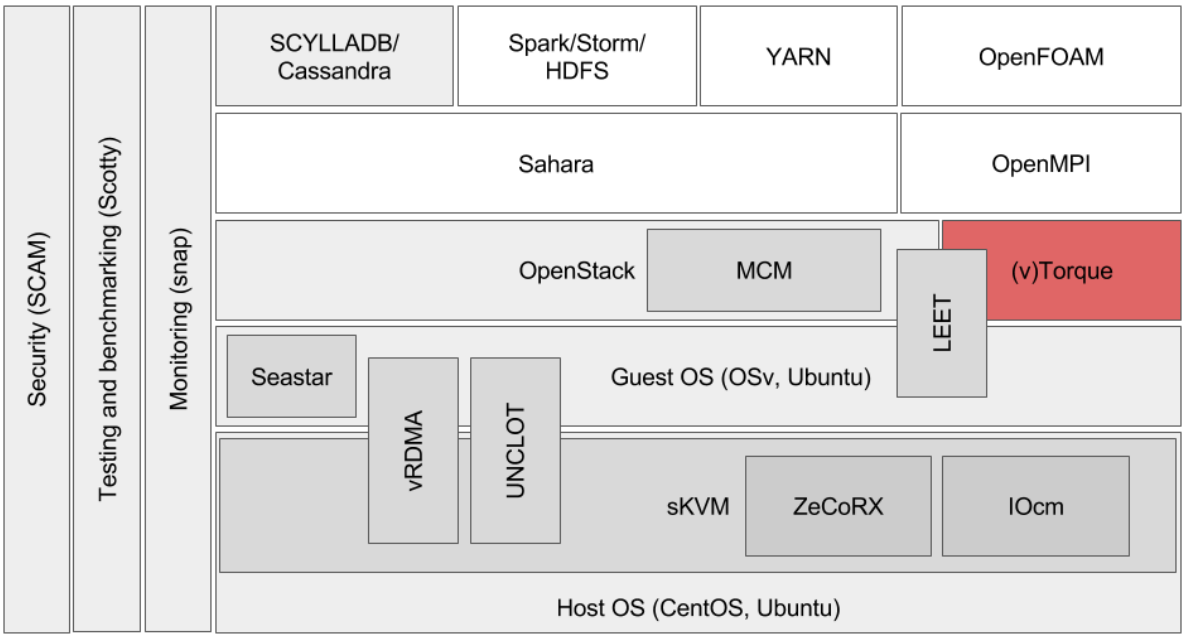

Figure 15. MIKELANGELO Components for the Cloud.

All, but two, components mentioned in this document are or will be integrated with the cloud testbed. The two missing components are vTorque and vRDMA. An integration with vTorque does not make sense, because vTorque is a virtualized batch system for HPC systems. Thus, vTorque is part of the HPC architecture. An integration with vRDMA in the cloud testbed, while technically available cannot be evaluated. Our implementation of vRDMA relies on RoCE-capable NICs, which are not used in the cloud testbed, and more generally are seldom used in cloud deployments.

The remainder of this section discusses how individual components are integrated with the cloud testbed and which component features are used.

12.1 MCM

MCM is integrated as a fixed service in the cloud testbed. The component is running as a service on the cloud's control node inside a container, like the other management services. MCM has access to OpenStack's APIs to control VM configuration. The integration with OpenStack allows for live-migration of resizing of VMs. Additionally, MCM integrates with OSmod. OSmod allows to MCM to control all components of the MIKELANGELO stack.

Finally, MCM relies on metering data collected via snap and stored in an InfluxDB-instance on the control node.

In the MIKELANGELO architecture we use MCM to research algorithms for cross-layer optimization. Here, we especially rely on the control mechanisms in MCM via OSmod. Furthermore, we use MCM's plugin mechanism for resource management strategies.

12.2 IOcm and ZeCoRx

IOcm and ZeCoRx integrate with the cloud via an installation on all compute hosts. IOcm can be turned on and off via a python script on the host. ZeCoRx can be enabled by installing a kernel with ZeCoRx. ZeCoRx does not need any additional control mechanisms since it is always on. Currently IOcm and ZeCoRx cannot run at the same time, because they build on different versions of the Linux kernel. Additionally to the direct integration, there is an integration of IOcm with OSmod. OSmod is capable to turn the IO core manager on and off. By extension IOcm's control feature is available for use in MCM resource management strategies and in Scotty as experimental parameter.

IOcm and ZeCoRx are fully used by the cloud testbed architecture. The only control parameter available is an activation switch of IOcm, which is integrated via OSmod with the cloud testbed.

12.3 UNCLOT

UNCLOT is an attempt to use cross-level optimisation techniques to optimise the overall performance of the applications running in a managed environment. This involves the guest operating system (OSv), the hypervisor (sKVM) as well as the management layer, which in case of OpenStack, relies heavily on Libvirt. Libvirt is used to configure the virtual hardware of virtual machines that are then launched with the support of the hypervisor. Before UNCLOT can be used, the virtual machines must be configured with a common shared memory pool which they are allowed to use to optimise network routing.

Contrary to HPC environment where Libvirt domains are driven by template fragments that can easily be merged into a complete machine specification, OpenStack constructs the domain files on the fly. This will therefore require minimal changes to the OpenStack code responsible for the preparation of the domain files. The necessary changes will be kept to a bare minimum to simplify the integration.

One of the remaining open questions is how to enable UNCLOT for deployed virtual machines. One possibility is to use the approach similar to IOcm, i.e. with an extension to OSmod that configures the use of UNCLOT. More flexible approach would be to allow

administrators to configure UNCLOT for virtual subnets via custom tags. Both of these approaches will be tested.

UNCLOT is being used primarily in the aerodynamics use case. Additionally, an integration with the virtualized data analytics use case is planned, but hinges on the integration of OSv with Spark.

12.4 SCAM

SCAM is part of the cloud testbed via installation on the compute hosts. SCAM is integrated with its detection, mitigation, and attack features. The detection mechanism is available as Snap collector. The snap collectors gathers SCAM monitoring data with the cloud testbeds snap installation. Mitigation happens via noisification directly on the compute hosts.

Beyond the basic functional integration of all SCAM features, the MCMintegration offers another feature, which can be used as part of infrastructure experiments. With MCM it becomes possible to isolate suspicious and malicious VMs on quarantine hosts.

12.5 Snap-Telemetry

Snap permeates the cloud testbed and thus is an integral part of the cloud testbed. Snap collectors and corresponding tasks run on all compute hosts directly. These collectors send their data to a central publisher plugin on the cloud testbed's control host. The data is collected in an InfluxDB instance. The stored data can conveniently be queried and visualized with the testbed's Grafana instance. A second, tight, integration with snap is available for Scotty. All of Scotty's experiments collect metering data via Snap. Scotty then offers the metered data directly for analysis to researchers. Furthermore, all Scotty experiments use Snap's tagging feature via OSmod to tag metering data for individual experiments.

Most of Snap's features are used in the cloud deployment. The installation and distributed management of snap are used by default. Furthermore, a long list of plugins is integrated in the cloud testbed. The most important plugins are psutil, libvirt, and OpenVSwitch plugins. Additionally, the collector plugins for the MIKELANGELO components are integrated as well. Finally, use-case specific plugins are being used by all use cases that run on the cloud testbed.

12.6 LEET

LEET is integrated with the cloud testbed via its OpenStack APIs. LEET-based applications are deployed in OSv via OpenStack. The aerodynamics use case builds on LEET to deploy OpenFOAM cases in a cluster on demand.

13 Integrated HPC infrastructure - vTorque

The integrated HPC infrastructure, called vTorque, is maturing towards a stable and feature rich solution for the execution of virtualized workloads in traditional HPC environments. During the last reporting period, while the integration of evolved components developed in the MIKELANGELO project took place, a few more differences between standard linux guests and OSv guests have been identified and are either in process of being or will be addressed in the last upcoming implementation phase. The intermediate architecture from M18 has proven to be flexible enough to cover the outstanding requirements, thus there no major changes introduced.

In the following subsections, the final HPC architecture of the vTorque[54] middleware addressing the last set of chosen requirements, is described. At first, newly introduced features independent of the actual guest operating system are presented, followed by changes for standard linux guests and those for the lightweight guest operating system OSv.

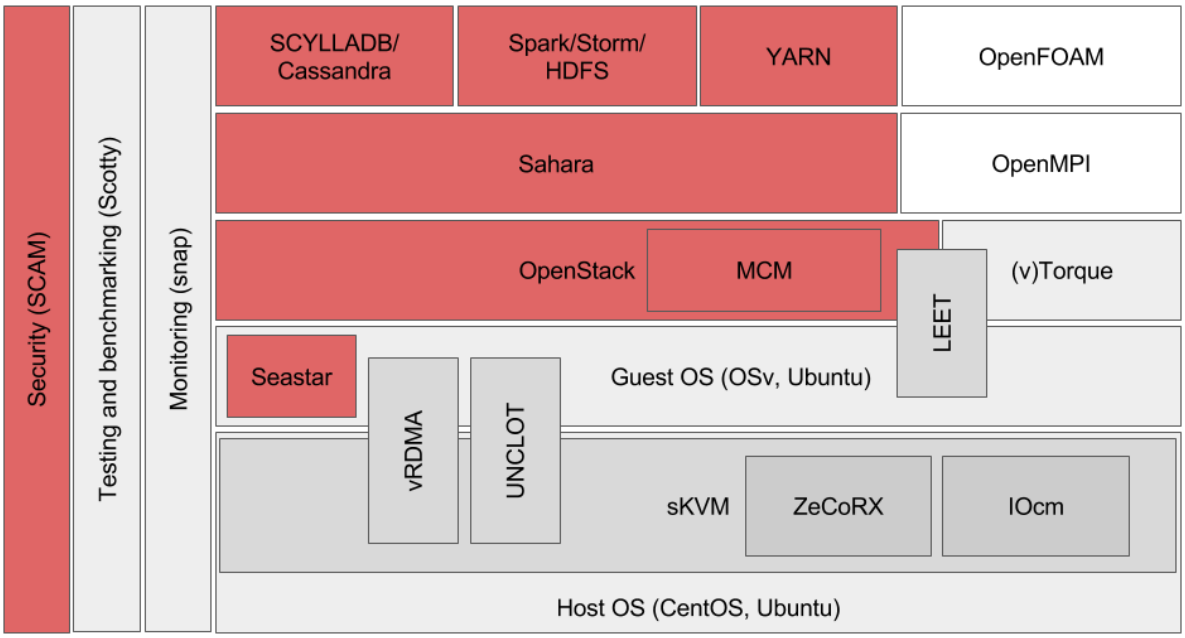

Figure 16. MIKELANGELO Components integrated in HPC.

The various components eligible for HPC environments presented in the figure above are described in the last set of subsections, with a clear focus on their integration into vTorque.

13.1 Extensions for VMs in general

The intermediate vTorque architecture from M18 presented in D2.20 [40] has been extended to cover the latest set of requirements. These focus in first place on additional features targeting more stability and flexibility. The changes introduced comprise the infrastructure

service layer as well as the command line interaction. In the following subsections the delta to the intermediate architecture are elaborated.

13.1.1 Command Line Interface

The command line interface design has been updated, new functionality added and existing revised. In the following text all changes are detailed.

qsub wrapper renamed to vsub

The job submission command line wrapper, formerly known as `qsub`, has been renamed to `vsub` and its interface also further extended as described in the succeeding parts.

mixed resource requests

The new design is going to provide users with the opportunity to define mixed resource requests in terms of different VM configuration within one job like Torque provides it for bare metal nodes. For example

```
vsub -l nodes=1:ppn=4,nodes=2:ppn=16 \
-vm img=<file>:1,image=<file>[:2] -vm vcpus=1,vcpus=6[:2] ..
```

This example above requests from Troque with '-l' one node with 4 cores and two nodes with 16 cores, similar to this approach the first image file and one vcpu is used for first guest on the first allocated node and for the other two allocated nodes the second image will be used to instantiate a guest with 6 vcpus each.

13.1.2 Image Manager

The final vTorque architecture also introduces a new tool called vmgr, that provides users and admin a convenient way to display information about available images for job submission and allows besides listing existing snapshots of suspended jobs also the deletion of snapshots, Users can manage their own images and list globally available ones for job submission, while administrators can utilize vmgr to delete images from the pool, add new ones including a description and list as well as delete all user snapshots.

There are no configuration options newly introduced by vmgr, however it depends on some of vTorque, thus uses the same config file. The command line offers the following commands:

Show vTorque's config. Admins will be provided with the full list, while user with all default parameters associated to the command line and the behavior of their job

```
vmgr show config
```

List available images.

```
vmgr show images
```

Show details for an image

```
vmgr show image <image_id>
```

Add new image, admins only

```
vmgr add --description=”image description” <image_file>
```

Delete an image, admins only

```
vmgr delete image <image_id>
```

Show all suspended VMs

```
vmgr show suspended [<jobID>|--user=<username>]
```

Delete suspended VMs

```
vmgr delete suspended [<jobID>|--user=<username>]
```

Suspended VMs can be resumed by the help of the submission tool vsub, as described in section 13.1.6.

13.1.3 CPU/NUMA pinning

Another addition that enriches the final architecture is the possibility to use CPU pinning either automatically by the help of numad and KVM or manually by defining the CPU pinning via a file passed on the command line

Users can control the pinning feature at submission time on the command line

```
vsub --vcpu_pinning=<yes|true|no|false|auto|path_to_pinning_file> ..
```

Corresponding global configuration parameters to be set in file `src/common/config.sh` are

```
VCPU_PINNING_DEFAULT=<true|false>
```

13.1.4 User provided metadata

User can pass on whitelisted targets used by cloud-init during the instantiation of guests. Administrators can on the one hand disable the feature completely or on the other hand define a set of whitelisted targets. It should always be considered that each whitelisted target may impact security. For example, if users can install a specific outdated package for which exploits are known or execute commands during guest instantiation with higher rights.

Users can provide a metadata file on the command line at submission time

```
vsub -vm metadata=<path to file> ..
```

Also as mixed requests, like the following one are possible, where 3 different files are provided with numbers starting at 1 indicating the corresponding guest. The last one without a number indicates that it should be used for all other guests.

```
vsub -vm metadata=<file>:1,metadata=<file>:2-3,metadata=<file>
```

Corresponding global configuration parameters to be set in file `src/common/config.sh` are

```
CLOUD_INIT_USERINPUT_ENABLED=<true|false>
CLOUD_INIT_USER_TARGETS=<comma separated list of cloud-init targets>
```

13.1.5 Live migration to spare nodes

Live migration of running virtual guests to a spare nodes utilizes PBS/Torque's node health monitoring functionality. When a node reports degrading health and the job has allocated one or more spare node(s), the migration from the degrading host to the idle spare node is triggered by Torque via the built-in node health monitoring[55] mechanism that executes the needed logic provided by vTorque.

Users can control the live migration feature at submission time on the command line

```
vsub --migrate=<yes|true|no|false> \
[--spare_nodes=<number>] ..
```

Corresponding global configuration parameter to be set in file `src/common/config.sh` is

```
LIVE_MIGRATION_ENABLED=<true|false>
DEFAULT_SPARE_NODES=<number>
```

When using this feature, vTorque will increase the amount of requested nodes from Torque, by the number of requested spare nodes. Spare nodes are not available for workloads, but since nodes are usually allocated exclusively, the overall resource consumption is increased. This feature shouldn't be used in conjunction with NUMA aware Torque installations, as in such cases the allocated spare node may be allocated on a different NUMA node, but still on on the same physical degrading host a guest from which we want to migrate.

13.1.6 Suspend and Resume

A suspend and resume feature for virtualized workloads requires from the integration point of view no major challenges. However, there is support to handle for fast forward jumps in the guest's clock in conjunction with established network connections, applications will obviously fail. All vTorque can provide at this point is the infrastructural support for this technology.

The feature must be enabled by administrators in order to be available to users. Users then can request the feature by appending `--checkpoint` to their `vsub` command line. Automatic resubmission can also be requested by appending in addition `--resubmit` to `vsub`'s command line call.

```
vsub --checkpoint=<true|yes|false|no> [--resubmit=<true|yes|false|no>]
```

Corresponding global configuration parameters to be set in file `src/common/config.sh` are

```
SR_ENABLED=<true|false>
DEFAULT_PARAM_SR_ENABLED=<true|false>
```

```
SR_AUTO_RESUBMIT=<true|false>
DIR_SUSPENDED_VMS=<path to global nfs snapshot dir>
```

When a job's walltime is hit and it is going to be stopped forcefully by the batch system, in the root epilogue instead of killing the guest, a snapshot is taken. Existing snapshots can be managed via commands

```
vmgr show suspended [<jobID>]
vmgr delete suspended <jobID>
```

And can be restarted by the help of command

```
vsub --resume <jobID>
```

The snapshot can be resubmitted in case auto-resubmission is not enabled, only. Note, resubmission requires job submissions to be allowed from compute nodes as the re-submit happens in the user epilogue wrapper from the nodes as last step .

It also requires the job wrapper to make use of a screen session inside standard linux guests since the initial SSH session will be killed by Torque during the epilogue. For OSv, due to its RESTful interface, no additional considerations need to be taken.

There is also a dependency on the DHCP service as resuming requires VMs to get the same IPs assigned as before, otherwise job execution will fail.

13.1.7 Generic PCI-passthrough

In order to support generic PCI devices, like accelerator and GPUs, vTorque provides capabilities to provide these devices via KVM passthrough mechanism to guests. In order to support generic devices on an infrastructural level, administrators need to provide logic that determines the actual device to be passed to a guest as a script. This script is run by vTorque during the metadata generation once for each guest on each physical node and expects either an empty string or a comma separated list of PCI device addressed to be passed on to the specific guest.

Corresponding global configuration parameter to be set in file `src/common/config.sh` is

```
PCI_DEV_SELECT_SCRIPT=<file>
```

The script is called with two arguments, the first one is a single number representing the guest, starting with zero. So, if there are two guests requested per node the script gets called with argument 0 as first followed by a second call where the argument is 1. The second argument is a string that the user provided on the command line at submission time with vsub. Keywords need to be provided by HPC system administrators as they need to parse it and give it a meaning. Such strings may also have an extended syntax similar to mixed resources requests.

```
vsub --pcidevs=”gpu,knightscorner,nvram,fpga” ..
```

This flexible approach allows administrators, i.e. to assign to each guest a specific device. The actual identifiers are defined by cluster administrators for their specific environment, as they also need to provide the corresponding mapping logic in form of a script.

13.1.8 Secured VM Management

Previously, the VMs were instantiated in the user prologue wrapper script vmPrologue.sh, however this requires users to be able to instantiate these. To ensure only authorized images can be instantiated users must not be able to instantiate or stop guests. Usually this is the default since standard users are not per default added to the KVM and libvirt user groups. It would allow them to upload an image file to the HPC system and boot it inside a batch job submitted through Torque, circumventing vTorque completely and gaining root in their VM. Then they could change to another user’s id and access its data stored on NFS.

The improved prologue and epilogue sequences resolves this issue, all guests are instantiated in the root-prologue sequence by the help of an asynchronous process as detailed in the figure below. And stopped in the root prologue.

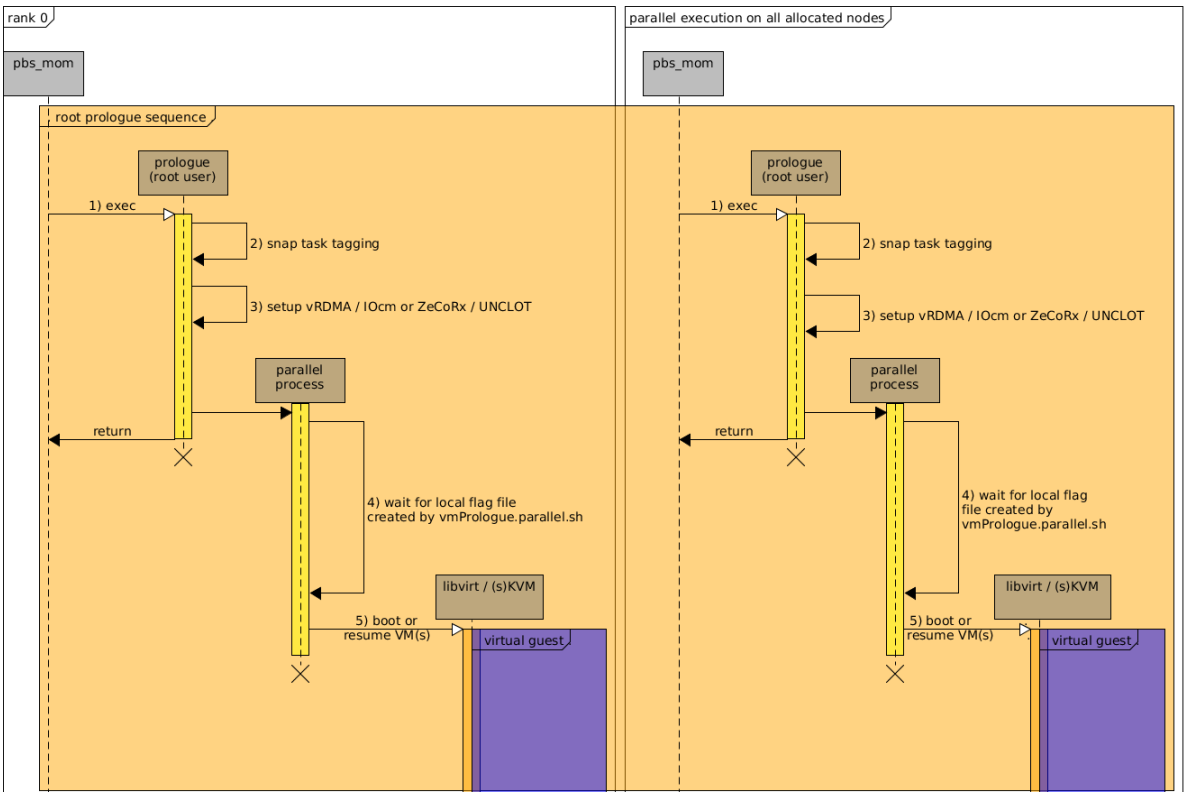

Figure 17. Final root prologue sequence.

Just before the process exists, the root prologue forks it. This process checks if a certain flag file is created by the user prologue, indicating all VM related files, that cannot be generated before, are ready and guests can be instantiated.

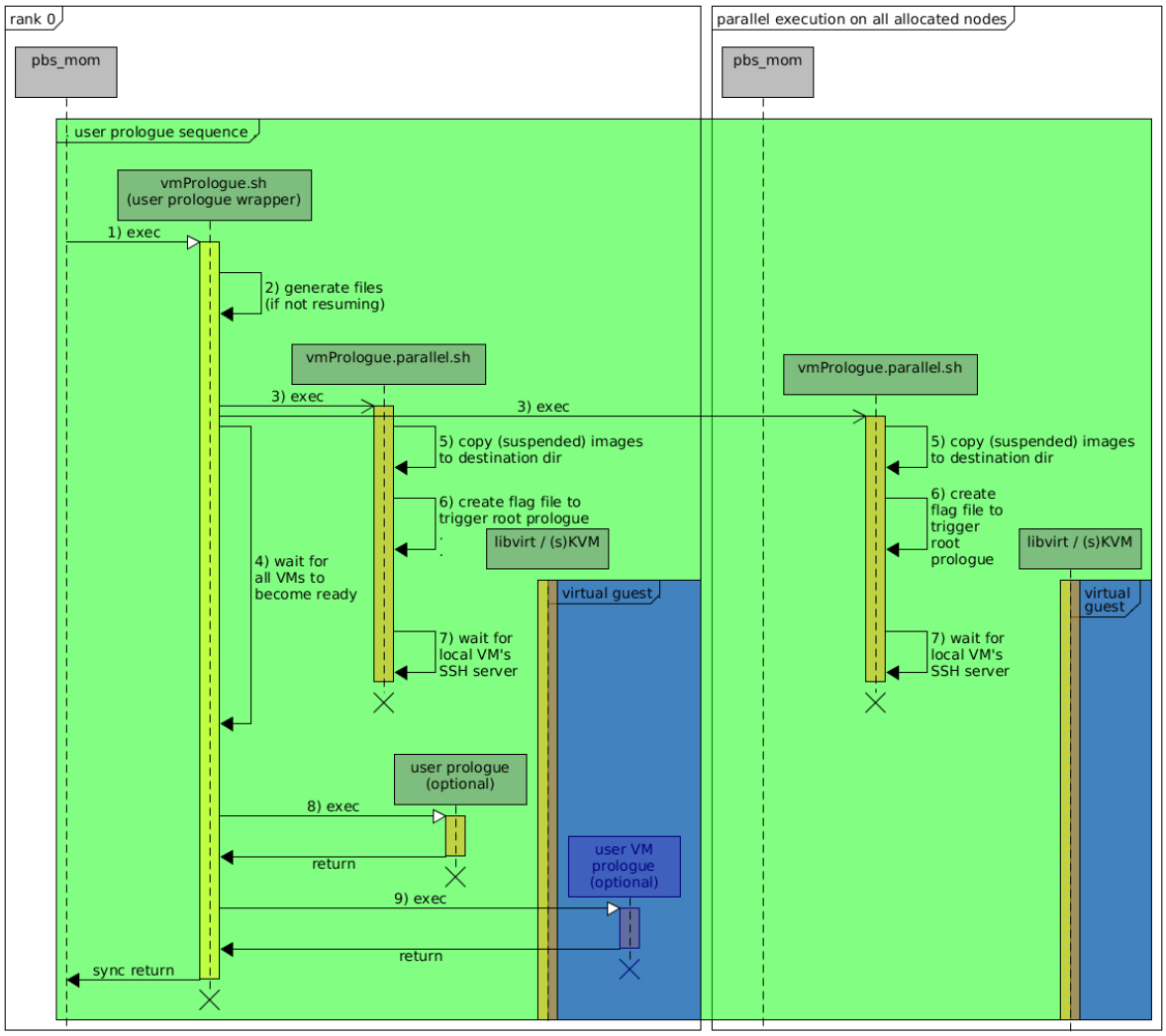

Figure 18. Final user level prologue.

The user prologue, instead of instantiating the guests directly, now creates a flag file that triggers the forked root prologue process to boot all VMs, as shown in step 11). The user prologue keeps running until all VMs to become available for job execution.

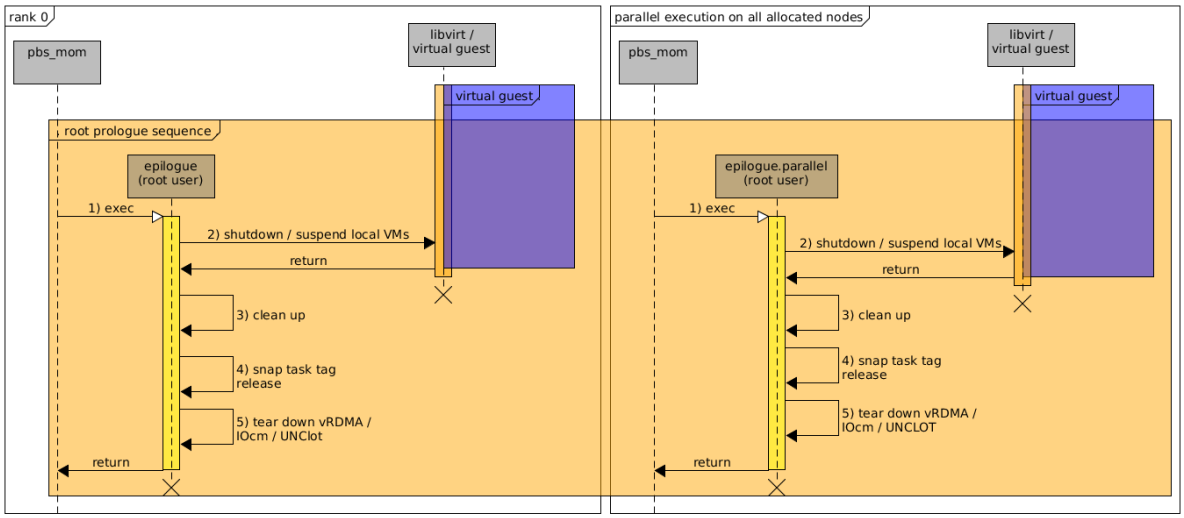

Figure 19. Final root user epilogue.

Root user epilogue, executed after the user epilogue, now takes care of VM shutdown / suspension. Since the VM user epilogue is triggered via systemd on shutdown, there is no more user epilogue logic for bare metal nodes needed.

13.1.9 Inline resource requests

Similar to the '`#PBS`' directive in header comments of job scripts '`#VPBS`' is introduced for all vTorque related parameters.

13.1.10 Submission filter

Also similar to standard PBS/Torque's configuration option SUBMITFILTER[56] there is a vTorque counterpart configuration option defined in file `src/common/config.sh`

```
VSUBMIT_FILTER=<file>.
```

It allows administrators to intercept job submissions before further processing. Using Torque's submission filter is not an option since all VM related arguments cannot be passed on to PBS/Torque as those are considered invalid.

13.2 Extensions for standard linux guests

Standard linux guests offer in comparison to the lightweight single user operating system OSv, additional possibilities. On the bare metal level Torque provides two sequences that are intended for preparatory work right before the actual job starts.

13.2.1 Root level VM prologue and epilogue

The root user prologue and epilogue can be used by the cluster administrators, to execute logic managed by cloud-init inside the virtual environment. It provides the same functionality as PBS/Torque provides for bare metal nodes, but for guests. The execution of optional root prologue and epilogue scripts happens via service scripts during the boot and shutdown process of guests, before and after the user job script has been run. This also means the root epilogue is not run when a bare-metal job ends, in case the suspend and resume feature is enabled, but when the user's job script has ended and the guest shuts down.

13.2.2 User level VM prologue and epilogue

User prologue and epilogue can be used inside the virtual environment like the user prologue and epilogue is available for the bare-metal execution as a standard PBS/Torque command line option.

13.3 Extensions for OSv

The start of user job scripts with OSv differs from standard linux guests in the way that it is achieved via a RESTful interface instead of ssh. However there is no impact to the overall workflow or any component interaction for this part.

However due to OSv's lightweight design approach there is no support for services like systemd provided in standard linux. Services are used by standard linux guests during the guest instantiation to steer the execution of (optional) root and user prologues in the virtualized environment as soon as dependencies, e.g. NFS mounted, are met. Instead there is only the possibility to execute a user level prologue, shortly before the actual job script is started.

13.4 Integrated MIKELANGELO components

The following subsections highlight the parts of the final architecture specific to each component's integration into vTorque's architecture.

13.4.1 IOcm and ZeCoRx

The integration of IOcm remains the same as presented in the previous architecture deliverable. IOcm, in contrast to ZeCoRx, requires root rights for the setup and tear down before and after batch job execution.. ZeCoRx is enabled with the corresponding kernel, without further configuration.

13.4.2 Virtual RDMA p2/p3

The vRDMA host layer functionality requires specific kernel modules to be removed and others to be inserted, thus it is handled at root user level during the prologue sequence. In the root epilogue the changes are reverted and the node is put back in the previous state.

For the guest it is also required to have certain kernel modules inserted providing vRDMA. This can be either achieved by appropriate packaging or via manual modification of the cloud-init metadata template file that is part of vTorque.

Corresponding global configuration parameters to be set in file `src/common/config.sh` are

```
VRDMA_ENDABLED=<true|yes|false|no>
VRDMA_ENABLED_DEFAULT=<true|yes|false|no>
VRDMA_NODES=<regular expression for node names>
```

13.4.3 UNCLOT

As we have seen in section 7, UNCLOT bypasses a standard networking (TCP/IP) stack with an intention to remove unnecessary copying of data when TCP communication occurs between VMs collocated on same virtualization host. It is currently implemented only for OSv unikernel. Inter VM communication relies on a shared memory provided by KVM/QEMU as an additional IVSHMEM virtual device.

The shared memory provided by IVSHMEM device is accessed via named pseudo files in `/dev/shmem/` (similar as regular memory mapped files). VMs which should exploit UNCLOT should be given access to the same shared memory region, i.e. pseudo file. If 2 VMs are started via QEMU command line, then example command looks like:

```
qemu --device ivshmem,shm=unique-name,size=128M ...(the usual parameters)
```

This command registers 128 MB of shared memory and exposes it to all virtual machines that are using the same unique-name. All of them will access this memory block in

```
/dev/shm/unique-name
```

QEMU will auto-create `/dev/shm/unique-name` pseudo file only when the first VM tries to access this file. However, QEMU does not take care of removing this shared memory region when all VMs stop using the file. Therefore, the infrastructure management layer needs to ensure these files are properly removed once they are no longer needed.

In order to integrate UNCLOT with vTorque, we need to be able to configure the virtualised job to use the shared memory. This will be done by introducing additional VM parameter that is provided to the command line tool. Because vTorque relies on Libvirt for VM configuration, enabling shared memory will result in the following additional Libvirt domain configuration

```
<domain type='qemu' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
  ...
  <qemu:commandline>
    <qemu:arg value='--device'/>
    <qemu:arg value='ivshmem,shm=__SHM_NAME__,size=__SHM_SIZE__'/>
  </qemu:commandline>
</domain>
```

The `__SHM_NAME__` is the unique name that is picked by the vTorque and depends on the job, i.e. every job should allocate its own shared memory region. `__SHM_SIZE__` is the size of the region.

Corresponding global configuration parameters to be set in file `src/common/config.sh` are

```
UNCLOT_SHMEM_ENABLED=<true|yes|false|no>
UNCLOT_SHMEM_SIZE=<size in MB>
```

13.4.4 Snap-Telemetry

Snap-telemetry is configurable by admins and job dedicated monitoring is started during the root prologue and stopped in the root epilogue. Those run as initial and final sequence in a batch job's life cycle.

Corresponding global configuration parameters to be set in file `src/common/config.sh` are

```
SNAP_MONITORING_ENABLED=<true|yes|false|no>
```

Further configuration can be found in file `src/components/snap/snapTask.template.json`. Ensure that file is not user-readable as it contains DB credentials

13.4.5 LEET

The Lightweight Execution Environment Toolbox (LEET) has been presented in section 5 where we described the details of three main components: package manager, cloud manager and orchestration manager. All these managers are simplifying the use of the OSv library operating system in various scenarios. However, only the package manager is relevant for the HPC integration. Package manager (Capstan) enhances the process of composing self-sufficient runnable unikernels based on OSv.

The package manager is a tool meant to be used by cluster admins and application developers. It therefore does not require any explicit integration with the target environment, in this case vTorque. The composed virtual machine images need to be placed in the global image folder by admins, the recommended way is with `vmgr` introduced in section 13.1.1. End users then can select the provided images from the global pool and use them with their own job scripts.

13.5 Guest Image Requirements for HPC

For both, standard linux guests and OSv guests, there some considerations to be taken when packaging an application as virtual guest image. This concerns mount points for shared filesystems, e.g. user's home, and the fast intermediate workspace that is usually available in HPC environments. Since the actual mount points differ from site to site, there is abstraction needed in order to make packaged applications flexible and independent of the targeted environment.

Cluster administrators are required to define the mapping from the bare metal environment into the guest layer by the help of mountpoint definitions in `cloud-init`'s metadata template.

- User's home
  User homes are expected to reside on a shared file-system, as common in HPC environments. Inside the guest it must be available under path '`/home/<username>`'. The `$HOME` directory from the bare metal environment has to be mounted inside the guest in order to provide direct access to the job script, the application binaries and input data residing on the cluster's filesystem.

- Shared intermediate fast workspace
  Another common aspect in HPC environments is a fast and highly scalable workspace directory used during computation to read and write intermediate data. It must be available under path '`/workspace`' inside guests. In comparison to the home, it is not intended for data persistency and will usually be cleaned, either directly after a job has run or after some defined timespan. User's are demanded in general to move their results to another persistent storage location.

- Manual installations
  It concerns packaging for standard linux guests, only. On linux manually built applications are usually placed in '`/opt`' must be placed in '`/opt-vm`' instead. Since '`/opt`' may be directly mounted from the cluster's bare metal environment in order to make commercial libraries accessible for virtualized workloads, however it requires these to be binary compatible.

- cloud-init support
  Guests have to come along with cloud-init support and datasource 'NoCloud' enabled. It is recommended to disable all other metadata-sources in case the packages application is intended for HPC, only.

- Required software packages

Depending on the target environment, slightly different packages may be required, e.g. specific infiniband driver for PCI passthrough into standard linux guests. These are recommended to be managed through cloud-init metadata templates, as provided with the vTorque source code and can be adapted for a specific environment, so guest images are independent of it. This concerns packages for i.e. nfs and ssh. However if administrators are going to build images dedicated to their environment and not intended for outside use, they may also preinstall all software packages that otherwise would be installed during boot by the help of cloud-init.

13.6 Security Considerations

Virtualization in HPC environments is required to consider basic security aspects that are given on the bare metal level. Users must not be able to gain root, as this allows them to change their uid and access other user's confidential data. This is achieved by the following two approaches:

- vTorque is shipped with an installer for convenience, however one is free to install all files manually. In the latter case it is crucial to ensure that proper access rights to all files are applied. Otherwise either Torque will not work, refusing to execute scripts as root that are user writeable, or user may exploit metadata and wrapper script templates. vTorque has several pre-condition checks implemented also checking access rights, but when files are user writeable any logic can obviously be circumvented by e.g. removing it.
- Authorized images
  User cannot be granted in production mode to utilize generic images they provide, as these may offer them root user access in their virtualized environment. Cluster administrators must remain in total control of user IDs, user access levels and thus the images. Only they can provide them to users via a global configuration parameter `'IMAGE_POOL_DIR'` that defines the location of whitelisted images available to their users. This directory must be owned by root and not must not be user writeable.

- The snapshot directory used for the checkpoint+restart feature must not be accessible by user at all, neither read nor write access must be granted to non-root users.
- User's `$HOME` in the guest
  To further increase security mounting is restricted to owner's home directory, achieved at the hypervisor level by mapping only his bare-metal environment's home into the VM.

- Automatic security updates for standard linux guests

Via cloud-init automatic security updates for standard linux guests can be applied during boot. While this is recommended, administrators may decide to turn it off, as it delays the boot process and requires connectivity from all guests to package repositories. Especially in this case it is strongly recommended to keep images up-to-date and rebuild them frequently and as soon as there are any security related updates available.

- Hypervisor hardening
  The last aspect is the hypervisor, zero-day exploits may allow users to break out of the virtual guest and access the underlying host operating system through the hypervisor with escalated privileges. This can obviously not be handled by vTorque, but it is recommended to look into SELINUX for Red Hat based host systems or for Debian based ones into Apparmor to limit possible impacts of such happening to a minimum.

13.7 Limitations

Even though vTorque provides a lot of new functionality to PBS/Torque, there are also two limitations introduced by it. These are mostly due to vTorque is implemented as non-intrusive wrapper layer making it independent of the actual Torque version in place. These limitations cannot be resolved without considerable efforts and a deeper investigation of PBS/Torque source code. Additionally, it would break vTorque's non-intrusive nature, but on the other hand performance benefits can be gained.

Reduced Job submissions per minute
One of the drawbacks is that vTorque is slowing down the maximum possible job submission in a certain timeframe. Argument parsing, string and file operations are more efficient in C than in bash and also some steps take place redundantly like parsing the command-line arguments.

No interactive jobs
Torque doesn't allow interactive jobs when there is a wrapper in front of its qsub submission command line tool. Also, there is no way to execute the required job wrapper script automatically before the start of the interactive user session.

13.8 Outlook

Even though vTorque provides production ready virtualization capabilities for PBS/Torque based HPC batch systems, there are additional features and nice-to-haves on the list that can't be addressed in a project's runtime. Also, minor drawbacks are introduced by vTorque in its current shape.

Increased efficiency and security

Moving from a non-intrusive approach over to an upstream C patch for the Torque source, would on the one hand provide faster processing of VM related steps like file generation, distributed logging, and also reduces chances for users further to tamper with previously generated files or misconfigured access rights for vTorque related directories.

Global spare node management

vTorque already provides the possibility to request spare-nodes on a per job basis, however this means doesn't increase the overall usage of HPC systems, but the opposite. Thus, it is desireable to have global spare nodes instead, not possible without patching PBS/Torque.

Interactive virtual jobs

With a trick user can have interactive jobs with vTorque and standard linux guests already. OSv on the other side is by design not intended for command line interaction, thus not eligible. All users need to do is to submit a job with e.g. a sleep command that keeps the job and so the VM alive. Then they connect manually via ssh to their virtual node(s). To provide this feature in a transparent way mirroring the interactive bare-metal job workflow from the user's perspective would be a nice feature, but requires major efforts.

14 Concluding Remarks

This deliverable presents the final architecture of the MIKELANGELO project, along with implementation and integration status information, where applicable. Individual components have been iteratively improved addressing the last set of requirements derived from our four use cases and surrounding usage scenarios originating from two distinct target environment, Cloud and HPC.

The first important conclusion of this deliverable is that contrary to our results in M12 and M18, where the main focus was on underlying components and their integration into the target environment and first validations by the four use cases, several cross layer optimizations and management tools came into place. Even more importantly, initial and intermediate architecture designs have proven to be valid for our integration and further matured requiring no or little change. The intermediate integrated prototypes released as part of the milestone MS5 supports a more stable and feature rich execution of complex scientific applications, such as Open MPI and OpenFOAM, and data analytics applications, like Spark and Hadoop HDFS, on top of the modified hypervisor with a lightweight unikernel OSv. Users are flexible to run their workloads and applications without modifications with either traditional middlewares (Torque and OpenStack) or in fully integrated environments cooperating all parts of the MIKELANGELO framework applicable to their target environment.

WP2 is, besides other tasks like the definition of Use Cases and collecting requirements, concerned with the design of architectures is accompanied by WP6 with continuous validations done by the four distinct use cases, leading to new requirements and refined usage scenarios. Few requirements were also introduced from external open source communities in regard to long-term exploitation and sustainability.

Some of the improvements and components outlined in the final architectures are still under development and can be expected with the final release at the end of the project delivering a production ready software-stack with several performance improvements and cross-layer optimizations beneficial for the Cloud and introducing virtualization support to widely spread PBS/Torque based HPC batch systems.

15 References and Applicable Documents

[1] Ronciak, J. B. (2004). *Page-Flip Technology for use within the Linux*. Retrieved from Ottawa Linux Symposium: https://www.kernel.org/doc/ols/2004/ols2004v2-pages-175-180.pdf
[2] MacVTap. (2010). *MacVTap*. Retrieved from http://virt.kernelnewbies.org/MacVTap
[3] Linux Foundation. (2016). *NAPI*. Retrieved from https://wiki.linuxfoundation.org/networking/napi
[4] Kivity, A. et al. OSv—Optimizing the Operating System for Virtual Machines. In *Proceedings of USENIX ATC'14: 2014 USENIX Annual Technical Conference* (p. 61).
[5] Memcached homepage, http://docs.libmemcached.org/bin/memaslap.html
[6] Seastar homepage http://seastar-project.org/
[7] ScyllaDB benchmarks, http://www.scylladb.com/product/benchmarks/
[8] Capstan homepage http://osv.io/capstan/
[9] UniK Github repository, https://github.com/cf-unik/unik
[10] D4.5 OSv – Guest Operating System – intermediate version, https://www.mikelangelo-project.eu/wp-content/uploads/2017/01/MIKELANGELO-WP4.5-SCYLLA-v2.0.pdf
[11] D2.16 The First OSv Guest Operating System MIKELANGELO Architecture, http://www.mikelangelo-project.eu/deliverable-d2-16/
[12] Capstan Package Automation, https://github.com/mikelangelo-project/capstan-packages
[13] MIKELANGELO Capstan Package Repository, https://mikelangelo-capstan.s3.amazonaws.com
[14] Report D4.5, OSv - Guest Operating System – intermediate version, https://www.mikelangelo-project.eu/wp-content/uploads/2017/01/MIKELANGELO-WP4.5-SCYLLA-v2.0.pdf
[15] Note: similar approach will be described in the next section where we discuss the application orchestration.
[16] OpenStack Heat, https://wiki.openstack.org/wiki/Heat
[17] DICE project home page, http://www.dice-h2020.eu
[18] Cloudify home page, http://cloudify.co
[19] OASIS Tosca standard, http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html
[20] Kubernetes home page, https://kubernetes.io
[21] Mirantis Virtlet plugin source, https://github.com/mirantis/virtlet
[22] Kubernetes Container Runtime Interface, http://blog.kubernetes.io/2016/12/container-runtime-interface-cri-in-kubernetes.html
[23] MIKELANGELO OSv Microservice Showcase Application, https://github.com/mikelangelo-project/osv-microservice-demo
[24] Kubernetes sig-node community, https://github.com/kubernetes/community/tree/master/sig-node
[25] NoCloud cloud-init documentation, http://cloudinit.readthedocs.io/en/latest/topics/datasources/nocloud.html
[26] MIKELANGELO Package repository, https://mikelangelo-capstan.s3.amazonaws.com/
[27] Open MPI Home page, https://www.open-mpi.org.
[28] OpenFOAM Home page, http://www.openfoam.com.
[29] Standard OpenFOAM Solvers, http://www.openfoam.com/documentation/user-guide/standard-solvers.php
[30] Java Home page, https://java.com/en/.

[31] Node.js Home page, https://nodejs.org/en/.
[32] Microservice demo application blog post, https://www.mikelangelo-project.eu/2017/05/the-microservice-demo-application-introduction/
[33] Cloud-init documentation, https://cloudinit.readthedocs.io/en/latest/.
[34] Hadoop HDFS, https://hortonworks.com/apache/hdfs/.
[35] https://www.mikelangelo-project.eu/wp-content/uploads/2017/01/MIKELANGELO-WP6.3-GWDG-v2.0.pdf
[36] Apache Storm, http://storm.apache.org.
[37] Apache Spark, http://spark.apache.org.
[38] Report D2.13 - The first sKVM hypervisor architecture, http://www.mikelangelo-project.eu/deliverables/deliverable-d2-13/.
[39] Report D2.16 - The First OSv Guest Operating System MIKELANGELO Architecture, http://www.mikelangelo-project.eu/deliverable-d2-16/.
[40] Report D2.20 - The intermediate MIKELANGELO architecture, https://www.mikelangelo-project.eu/mikelangelo-wp2-20-ustutt-v2-0/.
[41] DPDK Poll Mode Driver. http://dpdk.org/doc/guides/nics/mlx4.html.
[42] rsocket: Implementing TCP Sockets over RDMA. https://www.openfabrics.org/images/eventpresos/workshops2014/IBUG/presos/Thursday/PDF/09_Sockets-over-rdma.pdf.
[43] F. Figure, "Supplement to InfiniBand Architecture Specification Volume 1 Release 1.2.1," vol. 1, no. December 2009, pp. 1–7, 2010.
[44] Yi Ren, Ling Liu, Qi Zhang, Qingbo Wu, Jianbo Guan, Jinzhu Kong, Huadong Dai, and Lisong Shao. 2016. Shared-memory optimizations for inter virtual machine communication. ACM Comput. Surv. 48, 4, Article 49 (February 2016), 42 pages. http://www.cc.gatech.edu/grads/q/qzhang90/docs/YiRen-CSUR.pdf
[45] Open MPI Shared Memory BTL, https://www.open-mpi.org/faq/?category=sm.
[46] https://github.com/qemu/qemu/blob/master/docs/specs/ivshmem-spec.txt
[47] Nagle's algorithm, https://en.wikipedia.org/wiki/Nagle%27s_algorithm
[48] Iperf tool, https://iperf.fr
[49] Netperf tool, https://linux.die.net/man/1/netperf
[50] Apache bench tool, http://httpd.apache.org/docs/current/programs/ab.html
[51] D5.2 Intermediate report on the Integration of sKVM and OSv with Cloud and HPC, https://www.mikelangelo-project.eu/wp-content/uploads/2017/01/MIKELANGELO-WP5.2-INTEL-v2.0.pdf
[52] Snap plugin repository, https://github.com/intelsdi-x/snap/blob/master/docs/PLUGIN_CATALOG.md
[53] YAML. http://www.yaml.org/start.html
[54] https://github.com/mikelangelo-project/vTorque
[55] http://docs.adaptivecomputing.com/torque/6-1-1/adminGuide/help.htm#topics/torque/12-troubleshooting/computeNodeHealthCheck.htm
[56] Torque documentation for SUBMITFILTER, http://docs.adaptivecomputing.com/torque/6-1-1/adminGuide/help.htm#topics/torque/13-appendices/torque.cfgConfigFile.htm#SUBMITFILTER