MIKELANGELO

D4.5

OSv - Guest Operating System – intermediate version

Workpackage	4	Guest Operating System	
Author(s)	Nadav Har'El		ScyllaDB
	Gregor Berginc		XLAB
	Miha Pleško		XLAB
	Shiqing Fan		Huawei
	Fang Chen		Huawei
Reviewer	Joel Nider		IBM
Reviewer	Holm Rauchfuss		Huawei
Dissemination Level	PU		

Date	Author	Comments	Version	Status
2016-11-10	Nadav Har'El, Shiqing Fan, Gregor Berginc	Initial document structure	V0.0	Draft
2016-12-15	Nadav Har'El et al.	Document ready for review	V1.0	Review
2016-12-20	Nadav Har'El et al.	Document ready for submission	V2.0	Final

Executive Summary

This deliverable is part of Work Package 4, "Guest Operating System", of the MIKELANGELO project. MIKELANGELO builds a new cloud stack, making it easier and more efficient to run I/O-heavy and compute intensive High Performance Computing (HPC) applications in the cloud. The "guest operating system" is the operating system kernel, and system libraries, which run on each virtual machine. MIKELANGELO's guest operating system runs existing Linux applications - but strives for more convenient application deployment and better efficiency and lower resource usage compared to Linux. For an additional performance boost, the guest operating system also provides new APIs which can be used to write extremely efficient I/O-heavy asynchronous applications.

The focus of this deliverable is the second source code release of the four major components of the guest operating system: **OSv**, **Seastar**, **vRDMA** and **MIKELANGELO Package Manager**. OSv, a library operating system, implements the standard Linux ABI to allow running Linux applications; Seastar provides new APIs for highly efficient asynchronous applications; vRDMA allows efficient communication between guests using RDMA hardware in the host; and MIKELANGELO Package Manager is a mechanism for conveniently packaging an application and the MIKELANGELO guest OS together into a VM image which can then be easily deployed in the cloud. This document accompanies that source code; The document presents these components briefly, describes what is included in this source-code release, how these components improved since the first year release, and how to use this source code.

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

The MIKELANGELO project builds a new cloud stack with the intention of making it easier and more efficient to run I/O-heavy and compute intensive High-Performance-Computing (HPC) applications in the cloud.

The overall architecture of the MIKELANGELO cloud stack includes the following parts:

1. The cloud (OpenStack) and HPC (Torque) management software,
2. The hypervisor (sKVM), running multiple virtual machines (VMs, also known as "guests") on one physical machine ("host"),
3. The **guest operating system**, the operating system running on each virtual machine. This document focuses primarily on the guest operating system.
4. The actual application to run on the VMs.

The main focus of this deliverable is the second source code release of four major components of the guest operating system: OSv, Seastar, vRDMA and MPM. We will present these components only briefly, as a more detailed report on these components and their architecture was already presented in the M9 deliverable D2.16 [2]. In this document we focus more about describing what is included in this source-code release, what improvements it contains, and how to use it.

This is the *second* yearly release of all of these components - the previous release was described by three separate M12 reports - D4.4 [3] (about OSv and Seastar), D4.1 [4] (about vRDMA), and D4.7 [5] (about the application packaging framework, MPM). Additionally, D5.7 [6] also contained information on monitoring of OSv guests. Accordingly, in this document we will focus not only on explaining these components, but more importantly - on documenting the new work done since the previous release.

We will also refer to the M18 deliverable D2.20 [7] ("The Intermediate MIKELANGELO architecture") which already reported some of the improvements to the components of the guest operating system in the first half of this year, and laid out additional requirements, on which we will survey our progress.

Much of the work summarized in this deliverable was driven by requirements from the MIKELANGELO use cases in work packages 2 and 6. These requirements are a result of actual benchmarks and experiments with OSv and Seastar, building actual VM images using MPM, and actually implementing and testing vRDMA. During the course of that work, we had to develop these components in the directions needed by the actual use cases.

1.1 OSv

MIKELANGELO replaces the Linux kernel and system libraries by OSv, a new operating system designed especially for running efficiently on virtual machines, and capable of running existing Linux applications (with certain limitations). Compared to Linux, OSv has a significantly smaller disk footprint, smaller memory footprint, faster boot time (sub-second), fewer run-time overheads, faster networking, and simpler configuration management.

OSv is an open-source project [1] which started prior to the MIKELANGELO project by Cloudius Systems (now ScyllaDB), one of the MIKELANGELO consortium partners writing this document. Part of the WP4 effort of MIKELANGELO is to continue developing OSv to better suit the requirements of the MIKELANGELO use cases, which have been chosen to represent a broad spectrum of potential applications. This continued development considers performance, usability and application compatibility.

Most of the work done on OSv in the scope of this document has been contributed back to the OSv community, and has been merged into the main OSv source-code repository. We aim to keep contributing most patches to the main OSv repository, so that MIKELANGELO needs to maintain only a small set of MIKELANGELO-specific patches. This approach means less maintenance work for MIKELANGELO, more visibility and uptake for its patches, and more testing for MIKELANGELO source code by the general OSv community.

1.2 Seastar

While OSv allows running existing Linux applications, in D2.16 [2] we already noted that certain Linux APIs, including the socket API, and certain programming habits, make applications which use them inefficient on modern hardware. OSv improves the performance of such applications to some degree, but rewriting the application to use new non-Linux APIs can bring even better performance. So we described a new API, called "Seastar", for writing new highly-efficient asynchronous network applications, which are significantly faster than traditional applications.

Seastar is an open-source project [8] which was started, also by ScyllaDB, right before the MIKELANGELO development officially commenced. At that point, it was becoming clear that OSv improved the performance of certain applications (e.g., the slides [10] accompanying the OSv paper [9] reported a 34% throughput improvement to Cassandra), but we hoped that much bigger improvements could be achieved by designing new APIs and rewriting applications to use them.

Part of the WP4 effort of MIKELANGELO is to develop Seastar to better suit the requirements of the MIKELANGELO "Cloud Bursting" use case, which uses Cassandra. Seastar is developed

as a general asynchronous application framework which could be used by many different server applications on the cloud, but ScyllaDB is focusing on developing a Cassandra rewrite to showcase the potential of Seastar, and so far the results have been very encouraging ([11] reports throughput 10 times faster than Cassandra). This Cassandra rewrite is also released as open-source (see [11]), and a small part of that work is also funded by MIKELANGELO (as part of the "Cloud Bursting" use case work in Work Packages 2 and 6).

Another part of the WP4 effort on Seastar is to improve its usefulness to a wide range of potential applications for the MIKELANGELO cloud - beyond the specific use case we are evaluating.

As for OSv above, all of the work done on Seastar has been contributed back to the Seastar community, and has been merged into the main Seastar source-code repository. Again, we aim to keep contributing all patches to the main Seastar repository, so that MIKELANGELO does not need to maintain its own patches. This approach means less maintenance work for MIKELANGELO, more visibility and uptake for its patches, and more testing for MIKELANGELO source code by the general Seastar community.

1.3 vRDMA

The vRDMA part of the MIKELANGELO project aims to develop mechanisms for more efficient communication between different VMs, whether on the same host or different hosts, by virtualizing the RDMA hardware available in the underlying cloud hardware. In M8 deliverable D2.13 [21], we proposed three architecture designs of virtualized RDMA, addressing different hardware needs and application requirements. In the first year, we completed the development of prototype I [4], which implements the traditional socket APIs over virtualized RDMA.

The work described in this document (mainly section 4) covers the efforts in the developing virtualized RDMA solutions during the second year of the MIKELANGELO project. This year we focused on the development of vRDMA prototype II, which supports guest applications that directly use RDMA verbs. Prototype II differs from prototype I in the way that communication buffers and completion events are managed directly on the guest, allowing direct sharing of the pinned memory region between the guest and RDMA device. Since a guest application can directly pull the Completion Queues (CQ), this prototype largely mitigates the overhead of switching between host and guest.

1.4 MIKELANGELO Package Manager (MPM)

Besides the advances in compatibility and performance in the guest operating system, a flexible application package management is imperative. In previous reports (e.g. D2.16 [2]

and D4.7 [5]) we have already indicated the existence of tools for building OSv-based virtual machine images. However, certain limitations in them are preventing broader adoption.

MIKELANGELO Package Manager aims to remedy this by adding support for dynamic composition of virtual machine images based on pre-built application packages. These packages serve as self-contained building blocks. They consist of all the libraries, applications and other configurations necessary to deploy them into a runnable virtual machine. Dependencies between them simplify the organisation of packages into hierarchical units. Using these packages, end users are no longer obliged to install complete development environments just to have their applications built into runnable VMs.

In scope of application package management two tools are currently actively being worked on. On one side, extensions to Capstan [12] facilitate the composition of suitable virtual machine images, while on the other integration of the OpenStack provider into UniK project [13] simplifies the holistic management of the unikernels in the cloud environment.

1.5 Organization of this document

The rest of this document is organized as follows: The next four sections **Section 2, 3, 4,** and **5**, will be devoted to the four components of the guest operating system outlined above - OSv, Seastar, vRDMA and MPM, respectively. Each of these four sections will explain the contents of this release of the component and how to use it, and will survey the main contributions of MIKELANGELO to the development of this component in the past year. **Section 6** will summarize the key take-away points from this document.

2 OSv

2.1 Getting and running

As explained in the introduction, we contributed all patches that we had developed for MIKELANGELO to the main OSv repository, so that this component of MIKELANGELO can be downloaded directly from the public repository.

To retrieve OSv, run

```
git clone https://github.com/cloudius-systems/osv.git
git submodule update --init --recursive
```

The second command retrieves all git "submodules", which are external projects required by OSv, but are not included in its main repository. These external projects include, for example, ACPICA (for ACPI support), OpenJDK (for Java support), Musl (which implements parts of the standard C library), libNFS (for NFS client support), and more. These external projects are all open-source projects which OSv uses without any modifications.

The project includes a `README.md` file explaining how to build this project and run it. This README.md also lists the build tools which need to be installed on the build machine.

In section 5 below, we survey the MIKELANGELO Package Manager (MPM). Those are tools that we developed in the MIKELANGELO project for building a VM image, ready to run on the cloud, which contains the OSv kernel and the user's chosen application. MPM is more versatile and convenient than the tools which OSv previously had for building images, such as Capstan and "scripts/build". However, for testing purposes, one can also build simple images using the "scripts/build" tool bundled with OSv. For example, to build an OSv image with the "memcached" application (an in-memory caching application popular in the cloud (see [14])) - one should first make sure that the build environment is set up as described above, and then run in the OSv source directory:

```
scripts/build image=memcached
```

This command compiles the OSv kernel, the memcached application (which is fetched at build time directly from memcached's official source code repository), and finally creates an image containing both of them. The image is created in the file `build/last/usr.img` in qcow2 format, and can be converted to other formats needed by various hypervisors with the `scripts/convert` tool included with OSv.

To run this resulting image on your local machine (using qemu-kvm), run

```
scripts/run.py
```

2.2 Documentation

In addition to the publically-available source code repository mentioned above, we have made available, freely on the Web, extensive documentation about OSv:

- OSv homepage: http://osv.io/
- OSv research paper from Usenix ATC 2014, giving a detailed overview of OSv and its design: https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity
  This includes a 12-page research paper, slides, and a video of their presentation.
- OSv wiki: https://github.com/cloudius-systems/osv/wiki
  This includes over 80 documents explaining how to run various applications on OSv, how to debug on OSv, how to benchmark OSv, how to use OSv in various hypervisors, and more.
- OSv bug tracker: https://github.com/cloudius-systems/osv/issues
  We use this bug tracker extensively, to track the progress of all known OSv bugs and feature requests, including those found by the MIKELANGELO project. Bugs are extensively documented, and solved bugs are linked to the source code commits which fixed them, making the bug tracker very useful for understanding open bugs as well as how we solved past bugs.
- The OSv mailing list: https://groups.google.com/forum/#!forum/osv-dev
  This is a very responsive mailing list with some 390 members ranging from the OSv core developers and MIKELANGELO developers - to end users.

2.3 Main contributions

In this section we shall briefly survey the main improvements that were done to OSv in the second year of the MIKELANGELO project, since the previous source code release in D4.4 [3].

Much of our development and testing efforts on OSv this year focused on running two of the MIKELANGELO use cases - the *Aerodynamic Maps* and the *Cancellous Bones* use cases; A third use case, *Cloud Bursting*, benefits much more from Seastar and will be discussed in the next section. Both the Aerodynamic and Bones use cases required running the *Open MPI* HPC library. The Aerodynamic Maps use case uses MPI to run the *OpenFOAM* aerodynamic simulation application, and was particularly interesting to several MIKELANGELO partners (XLAB, Pipistrel and Scylla). Getting the Open MPI library to work, and work well, on OSv is important not just for these two uses cases, but also other use-cases based on Open MPI, and basically, most HPC workloads, since most of them are based on MPI.

A second goal which drove the development of OSv this year was the desire to improve OSv's support for applications, i.e., make OSv usable by a wider range of Linux applications.

In light of these goals, the following were the main improvements we contributed to OSv this year:

1. **Implemented missing or broken system calls and C library functions**:
   Though dozens of different Linux applications already ran successfully on OSv before this year, there still remained a tail of rarely used system calls and C library functions which OSv had not yet implemented, or had implemented incorrectly due to lack of testing. Large and complex software such as Open MPI and OpenFOAM (which span several millions lines of code), unsurprisingly used several of these unimplemented or mis-implemented functions, which we discovered and fixed as part of WP4. Some of the many examples include better support for `getrlimit()`, edge cases of `unlink()`, support for various system calls being called through `syscall()`, `__asprintf_chk()`, `memalign()`, `faccessat()`, `fstatat()`, `malloc_hooks`, `get_mempolicy()`, `pthread_key_delete()`, fixing bugs in `pthread_spin_lock()`, `sendfile()`, `getifaddrs()`, `if_nameindex()` and many more.
2. **Implemented support for additional runtime environments, such as Go, Node.js, Pascal, Erlang and different versions of Java:**
   OSv can, at least in theory, run any Linux executable with certain limitations (notably, that it does not use fork()). However, in practice running different applications for the first time may expose different existing bugs and still-missing features in OSv. So one of the most productive ways to support many additional applications at once on OSv is to support and test additional languages, runtime environments and application frameworks. For example, as soon as OSv supported Java's JVM well (this was the first use case of OSv, prior to the MIKELANGELO project), a large number of different Java and Scala applications could be run on OSv.
   This year we added support to OSv for the Go language, node.js, Free Pascal, Erlang and also for various new variants of Java (including Java 9 and Java with "compact profiles").
   Supporting the Go language runtime required the most effort because it behaves significantly different from most other Linux applications: While most Linux applications use the C library, which OSv implements (and links into the executable with its dynamic linker), Go actually uses the SYSCALL machine-language instruction to call system calls directly, and support for this was missing in OSv before we added it this year. The Go support also required fixing numerous additional bugs in OSv that Go exposed - just one example: A bug in our seeding of the pseudo-random number generator (/dev/urandom) caused Go programs to take a very long time to start, before we fixed it this year.

3. **Support newer build environments:**
   OSv was originally developed using specific versions of the C++ compiler (gcc) and libraries (libstdc++, Boost, etc.), In MIKELANGELO, different developers have different build environments, running different versions of the compiler, libraries, and other tools. Moreover, during this year, newer versions of these tools and libraries kept coming out, including a major release of Gcc, 6.0. Some of these upgrades resulted in breakage of OSv - sometimes because of new bugs in these tools, but sometimes because of old bugs in OSv which surfaced with the new compiler or new library. We had to fix or work around these bugs to get OSv to compile on a large variety of build environments. This is important for MIKELANGELO partners (who don't necessarily have identical build machines), but also for the general OSv users. OSv has now been built and tested on many different build environments, ranging all the way from 4 year old Linux distributions with gcc 4.8 (which was the first version of gcc to properly support C++11), to the newest Linux distributions at the time of this writing (Fedora 25, with gcc 6.2.1).
4. **Improved OSv's NFS client:**
   In the first year of the project we added an NFS client to OSv, as reported in D4.4 [3]. The NFS client allows mounting of shared directories over the NFS protocol, which is often used in HPC workloads, and was necessary for the MIKELANGELO cloud. This year, we continued to develop and de-bug this NFS client.
5. **Improved OSv's isolated thread APIs:**
   As a matter of design principle, OSv does not support classic Unix "processes", i.e., threads which are fully isolated from each other. However, while running our Open MPI-based use cases we realized that in some cases, we do need a certain level of isolation beyond that which threads can traditionally offer. In the first year, as reported in D4.4 [3], we already added new process-like APIs to support some isolation requirements that arose: be able to run the same executable in multiple threads, without sharing their copies of global variables, and give the different threads different environment variables.
   This year we had to improve on the new process-like API, to support more robust mechanisms for starting such isolated threads and waiting on them, and operating on the children of a particular thread.
6. **Partial NUMA support**:
   Large multi-core VMs are often multi-socket, and have non-uniform memory access (NUMA). In other words, subsets of the cpus are closer to parts of the memory. Most of our use cases, including Open MPI and Seastar, make use of various NUMA-related features supplied by the kernel. These kernel features allow the application to query the NUMA configuration, to allocate memory for a specific core, and to pin a thread to a specific core. Our recent benchmarking efforts of OpenFOAM indicated that not

properly supporting NUMA can slow down an Open MPI application by about 20%. This year, we added support to OSv for some of these features, notably the ability of one thread to pin() another thread to a specific core, to query about this pinning, and to have it inherited when spawning a new thread. Next year, we will need to continue adding the rest of the unimplemented NUMA features.

7. **Cloud-init:**
   Cloud-init is a standard mechanism of configuring VMs dynamically based on configuration files presented to the image by the cloud provider, over the network or over a secondary disk attached to the VM. OSv's cloud-init support was missing some standard cloud-init features which were important for the MIKELANGELO cloud, and this year we added some of this missing functionality. This includes the retrieval of a configuration file from a secondary disk, the ability to specify mountpoints, and to specify the VM's hostname.
8. **Improved DHCP support:**
   OSv already supported DHCP, but this year we needed to add two missing features that were required for correct operation of the MIKELANGELO cloud: Support for hostname setting (the VM gets its hostname via cloud-init, and then tells the DHCP its own hostname), and the ability to release the DHCP lease on shutdown of the VM.
9. **Improving file-system performance**
   In our experiments with the Big Data use case, we have realised that OSv's operations on the underlying file-system (ZFS) do not perform as well as in Linux. Thorough analysis has revealed limiting factors in the implementation of the Virtual File System (VFS) that essentially serialised all disk access. Preliminary patches have been provided, significantly improving the performance, however additional experimentation will be required next year to stabilize these patches, and to improve the performance even more.
10. **Virtual RDMA**
    This year, we implemented "prototype II" of the vRDMA code, which required extensive coding within OSv, in cooperation with changes in the host. We describe vRDMA in detail later in this document.
11. **Fixed miscellaneous bugs in OSv**
    Above, we already mentioned numerous bugs we had to fix in OSv before certain applications or frameworks like OpenMPI or Go could properly run. We found and fixed numerous other bugs in the last year, some of them could potentially cause a crash of the entire VM (e.g., a rare bug involving thread objects saved in the stack). Other bugs were less serious, but nevertheless needed to be fixed for certain applications to work correctly on OSv.

2.4 Key Performance Indicators

The MIKELANGELO Grant Agreement lists the following Key Performance Indicators which are relevant to the OSv guest operating system:

- KPI 3.1: The relative improvement of efficiency of MIKELANGELO OSv over the traditional guest OS.
- KPI 3.2: The relative improvement of efficiency [size, speed of execution] between the Baseline Guest OS vs. the MIKELANGELO OSv.
- KPI 3.3: The relative improvement of compatibility between baseline and MIKELANGELO versions of OSv.
- KPI 7.1: All use cases properly demonstrated.

This year, we've made significant headway on improving **KPI 7.1**, by improving OSv's support for running the MIKELANGELO's use cases. We've put a special focus on making the Open MPI HPC library work properly on OSv, as this library is used by two of the use cases (Aerodynamic Maps, and Cancellous Bones). In the beginning of the year, neither Open MPI nor OpenFOAM (the computational fluid dynamics software used by the Aerodynamic Maps use case) worked correctly on OSv, because of various missing features, small and large, as described earlier. Today, the Aerodynamic Maps use case is fully able to run on OSv, including parallel runs on multiple cores and nodes. Performance on a single node with multiple cores on a single socket (a case where there is no I/O, so we did not expect MIKELANGELO to provide a speedup) showed OSv's performance to be very close to Linux's, as expected. A second MIKELANGELO use case, the Cancellous Bones one, also fully works today on OSv, although this effort has not progressed enough to provide meaningful benchmark yet. For a third MIKELANGELO use case, "Big Data", we have already successfully run several applications (such as Hadoop), although performance was disappointing - mainly because of slowness in OSv's filesystem layer, an issue we already mentioned above and we will need to address next year. For the fourth MIKELANGELO use case, "Cloud Bursting", based on the Cassandra NoSQL database, we have seen Seastar - described in the next section - to provide much better improvement to performance than OSv does, so our main focus in that use case is Seastar. Still, we have verified that Cassandra does work correctly on OSv.

We also made significant progress in **KPI 3.3**, which is about OSv's ability to run a large number of different Linux applications. As mentioned in the previous subsection, this year we fixed a large number of C library functions which were either missing, or had implementation bugs, which prevented some applications from running correctly on OSv. We got several important application frameworks and languages, including Go and new variants of Java, to work on OSv, which allows running a large number of new applications on OSv.

Another way to demonstrate improvement of **KPI 3.3** is the number of Linux applications which have been tested to run correctly on OSv. osv-apps.git [15] is a public repository of applications and packages which are known to work on OSv. We also have in MIKELANGELO our own repository of applications for the MIKELANGELO use cases, such as Open MPI and OpenFOAM. Since the beginning of the MIKELANGELO project, the number of these tested packages grew from 55 to about 90.

As already mentioned above, most of the work this year was to get the different use cases running correctly on OSv, and do so with reasonable efficiency close to that of Linux. To achieve good results for **KPI 3.1,** we will need to demonstrate performance which is better than Linux. We plan to do this next year, and find specific scenarios where such an improvement is possible (e.g., since a single-node MPI computation does not do any I/O or system calls, on such a setup OSv is unlikely to be better than Linux). In the past, we were already able to measure various cases were running an applications on an OSv guest was faster than the same application running on a Linux guests - such as Cassandra was measured to have 34% higher throughput on OSv than on the Linux baseline, and memcached got 22% higher throughput.

For **KPI 3.2,** we did not yet focus on optimizing OSv beyond its initial performance improvement because most of the work spent on OSv has been to improve its compatibility with applications, as mentioned above. We did some work this year to make OSv images even smaller than they used to be, by dropping unnecessary libraries, and by allowing the creation of images without a ZFS filesystem (for applications which do not need permanent storage on the image, just a RAM disk and network storage). These improvements reduced as much as 3 MB from a typical OSv image size (which, for some tiny applications, may be a significant percentage of the image size).

3 Seastar

3.1 Getting and running

As for OSv, also for Seastar we have contributed all patches that we developed for MIKELANGELO to the main Seastar repository, so that this component of MIKELANGELO can be downloaded directly from the public repository.

To retrieve Seastar, run

```
git clone https://github.com/scylladb/seastar.git
git submodule update --init --recursive
```

The second command is needed for the same reason we needed it for OSv in the previous section: It retrieves external open-source projects that are required by Seastar, but are not included as part of it. These external projects include DPDK (for user-space networking, which can optionally be used in Seastar), and several more.

Seastar also has certain requirements on tools that are installed on the build machine. The script install-dependencies.sh installs them automatically using your Linux distribution's package manager (Ubuntu, Debian, RHEL and Fedora are supported).

The project includes a "README.md" file explaining in more detail how to build the project and run it.

3.2 Example

In this section we will show how to build and run one simple Seastar application. A more complete tutorial of how to write and build Seastar applications is included as an attachment to this document. Additionally, the quintessential (and most complex) Seastar application is the Cassandra clone, ScyllaDB, available (also as open-source software) at [11], which we use for the "Cloud Bursting" use case of MIKELANGELO.

To build the Seastar library (libseastar.a) and some test applications, first make sure the build environment is properly set up as described above, and then run in the Seastar source directory

```
./configure.py --mode=release
ninja
```

Among the test applications that this builds, can be found a Seastar-based memcached rewrite. It can be run with the command:

```
build/release/apps/memcached/memcached -c1 -m1G
```

This will run the Seastar-based memcached rewrite, with 1 CPU core and 1 GB of memory (note the application here will be run on the Linux host, not inside a guest). While [9] reported that OSv improved the throughput of the unmodified memcached by 22% (over running the same memcached in Linux), we measured in [16] a 78% improvement in throughput for the rewritten Seastar-based memcached.

We attach to this report a document titled "*Asynchronous Programming with Seastar*". This is a draft of a tutorial to Seastar, which includes many examples of Seastar programming and a much longer explanation how Seastar works, and why. The tutorial is still incomplete - both in depth and in breadth - but is already useful for introducing new application writers to Seastar, and we are working on expanding it.

We believe that excellent documentation - including a tutorial and the API documentation (see below), are absolutely necessary to drive more adoption of the Seastar library by more users beyond the MIKELANGELO participants.

3.3 More documentation

In addition to the publically-available source code repository mentioned above, we have made available, freely on the Web, extensive documentation about Seastar:

- Seastar homepage: http://www.seastar-project.org/
- Documentation of the Seastar API: http://docs.seastar-project.org/master/index.html (built from the Seastar source code using doxygen).
- Seastar tutorial: https://github.com/scylladb/seastar/blob/master/doc/tutorial.m This document introduces beginners to asynchronous programming in Seastar with practical examples. A snapshot of this work in progress is attached to this document.
- Seastar bug tracker: https://github.com/scylladb/seastar/issues
- Seastar mailing lists: https://groups.google.com/forum/#!forum/seastar-dev
- A description of ScyllaDB's architecture, explaining how the use of Seastar made it 10 times faster than Cassandra: http://www.scylladb.com/technology/architecture/
- Seastar wiki: https://github.com/scylladb/seastar/wiki

3.4 Main contributions

In this section we will briefly survey the main improvements that were done to Seastar in the second year of the MIKELANGELO project, since the previous source code release in D4.4:

1. **More Seastar documentation:**
   We believe that good documentation is essential for the adoption of Seastar both inside and outside MIKELANGELO. Seastar's APIs are significantly different from the more widely familiar Linux APIs (such as sockets, read(), write(), etc.), and also require

significant C++14 familiarity. So we believe that without good documentation, we might scare away potential users outside its current circle of developers. This is why we continued to spend significant effort to document Seastar in various forms: On its website and wiki, a tutorial in book form, and online API documentation (in doxygen form). All this documentation is freely available on the web (see links above), just like Seastar itself. This effort is ongoing, and we believe it is important to continue this effort.

2. **Improved Remote Procedure Call (RPC) in Seastar:**
   Seastar runs on a single machine, but many Seastar applications, including the ScyllaDB 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.
   This year we implemented several improvements to the RPC mechanism. In addition to numerous small improvement and bug fixes, two notable improvements were compression and negotiation. Negotiation allows different versions of the application to communicate with each other, which is important for distributed applications on the cloud because it is impossible to upgrade all the nodes at the same time without incurring significant downtime of the service.
3. **Seastar I/O scheduling:**
   One of the key requirements that arose in the "Cloud Bursting" use case was to ensure that performance did 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. For this, we implemented this year an I/O scheduler for Seastar: The application can 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.
4. **IOtune:**
   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. The requirement to tune the I/O parallelism to what the disk can actually handle led to the development this year of "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.

5. **CPU scheduling (partial implementation):**

   The I/O scheduling feature which we described above (and implemented this year) 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. This year we designed a full-featured CPU schedule for Seastar, but did not implement it yet (we plan to do it next year). Instead, we implemented a stop-gap measure: Seastar already supported the concept of "Seastar threads". Those are not actual O/S-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.

   This year, we added to Seastar threads the ability to control the amount of CPU time that each Seastar thread gets; A Seastar thread can yield if it receives over a threshold percentage of the CPU. This feature is useful to guarantee that low-priority background tasks in the server cannot monopolize the CPU.

6. **Miscellaneous performance improvements:**
   We discovered and fixed several places where the Cloud Bursting use case was showing reduced performance because of inefficient code. For example, we improved networking performance by implementing better batching, improved asynchronous disk I/O, and reimplemented output streams (used for both disk and network I/O).
7. **Improved monitoring capabilities:**
   Seastar already had some monitoring features via collectd. This year we added additionals statistics, new types of statistics (such as histograms) and additional protocols for collecting these statistics (we added support for a REST API and for Prometheus).
8. **Support the ext4 filesystem:**
   To support asynchronous disk I/O, Seastar requires the O/S to correctly implement AIO on files. Unfortunately, Linux's ext4 filesystem has several bugs in its support for AIO, which means several important operations may sometimes block (e.g., waiting on a lock), something which hurts performance in the single-thread-per-CPU Seastar. For this reason, we used to recommend that only the XFS filesystem be used. But unfortunately, this was an inconvenience for users because XFS is not a very popular filesystem.
   This year, we discovered that XFS actually has its own rare AIO bug - size-changing operations (append) are serialized by the kernel. We fixed this bug, and the same mechanism we needed to circumvent the XFS hangs (shadowing XFS's internal lock in Seastar) could also be used to circumvent the ext4 hangs. So now Seastar also works well on ext4.
9. **Log-Structured memory allocator:**
   Seastar's malloc()/free() 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 evicted), and the LSA APIs allow tracking where an object moved, or prevent movements temporarily.

The LSA code is still work in progress, so it hasn't been committed into the Seastar repository yet, but it is heavily used in ScyllaDB, and its implementation can already be found in ScyllaDB's repository.

10. **Improved DPDK support:**

    Seastar allows optionally using the DPDK user-space network card drivers for even better network performance in Seastar. This year we worked on supporting newer versions of DPDK, and newer network cards, in Seastar.

11. **Miscellaneous Seastar improvements:**

    During the year we also did a lot of other miscellaneous Seastar improvements - fixing bugs, adding more APIs for more convenient asynchronous programming, adding techniques and tools for making debugging easier, and adding missing features. For example, one of the many features we added this year is asynchronous DNS (domain-name service) lookups. Some of these features are already documented in the Seastar documentation (tutorial and doxygen). The git repository's log contains a full list of everything we have done in Seastar over the year.

3.5 Key Performance Indicators

The MIKELANGELO Grant Agreement lists the following Key Performance Indicators which are relevant to Seastar:

- KPI 3.1: The relative improvement of efficiency of MIKELANGELO OSv over the traditional guest OS.
- KPI 3.2: The relative improvement of efficiency [size, speed of execution] between the Baseline Guest OS vs. the MIKELANGELO OSv.
- KPI 7.1: All use cases properly demonstrated.

As we already observed in D2.16 [2], beyond the performance improvements that can be achieved by changing the kernel from Linux to OSv, a very promising direction is to introduce new APIs, namely Seastar, which a modified application could use and achieve far better performance than could be achieved by just modifying the kernel and keeping the Linux APIs. As mentioned above, we've done some preliminary measurements of the contribution of Seastar to **KPI 3.2** and **KPI 3.1:** We found that a Seastar-based memcached re-implementation achieved 78% higher throughput than the standard one (compare this to 22% improvement by baseline OSv), and that a Seastar-based re-implementation of Cassandra achieved 900% higher throughput than the standard one (compared to just 34% improvement by baseline OSv).

We are using ScyllaDB, an open-source reimplementation of Cassandra using Seastar, for improving the performance of the "Cloud Bursting" use case. Therefore, Seastar contributes to **KPI 7.1** (all use cases properly demonstrated).

4 vRDMA

In this section, we describe the design and technical details of vRDMA Prototype II. First we introduce the overall design architecture in section 4.1. Next we provide a detailed description of corresponding drivers implementation for the backend in section 4.2 and frontend in section 4.3. In section 4.4, we demonstrate an example of installing vRDMA patch using OSv as the guest OS. We measure the I/O performance and analyse the key performance indicators in section 4.5.

4.1 Architecture Overview of Prototype II

This section gives a brief introduction to the prototype II architecture. As introduced in Deliverable 2.13 [21], the goal of vRDMA is to develop a paravirtualized device driver that supports RDMA. As shown in Figure 1, the paravirtualized I/O driver consists of a frontend driver running on a guest OS and a backend driver running on the host side with direct hardware access. The frontend driver of prototype II functions as an I/O request manager in OSv. It provides a hypercall functionality similar to the one used in HyV [22], however with a different implementation which suits the needs of virtio API defined in OSv. The backend driver of prototype II manages the I/O requests on the host side which directly accesses hardware devices. The proposed backend driver in Prototype II modifies and extends the existing HyV vhost implementation to the target Linux kernel version 3.18. Our architecture differs from HyV implementation in the way that we eliminate unnecessary context switches between user and kernel space, port only required OFED (The OpenFabrics Enterprise Distribution) kernel definitions in order to support the kernel verb structure in the user space library. This will largely simplify the overall complexity of the architecture and make it easy to be implemented and ported to other Unikernel-like Library OSes.

In our case, a library OS like OSv normally contains a minimum set of libraries that are required to support the targeted application. Thus porting HyV frontend to such OSes becomes an infeasible solution. In order to avoid the large amount of tedious work involved in porting OFED modules and additional FreeBSD kernel support to OSv, our newly designed and implemented virtio-rdma provides a feasible solution and reuse the existing HyV backend driver on the host.

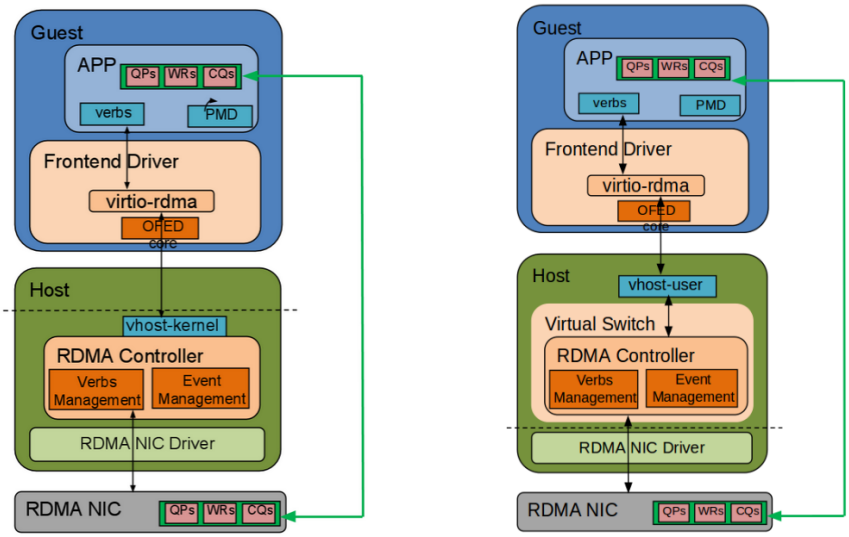

Figure 1. Architecture of vRDMA prototype II. Left: updated architecture with vhost driver moved to kernel space. Right: Original design proposed in D2.13.Control path (black arrows) is managed by the hypercalls. Data path (green arrows) is directly mapped and shared between guest and host.

4.2 Backend Driver

On the host side, we take the advantage of using HyV vhost driver. The communication contains the same data structure as HyV uses. Hypercalls are made between guests and host side, where the hypercall parameters are converted by a context switcher in virtio-rdma. We will describe the context switcher in the frontend driver (section 4.3 Frontend Driver) in this deliverable. More details in backend driver installation can be found in D3.2 [23].

4.3 Frontend Driver

There are a few existing design schemes for the RDMA para-virtualization frontend driver. A virtualized RMDA can be implemented in different layers of the guest OS, including drivers in guest user space, providers in guest kernel space, or both. For vRDMA prototype II, we designed and implemented a new virtio-rdma frontend driver, which supports single-address-space library operating systems. In the following section, we first introduce the design of our virtio-rdma front driver, then we compare the proposed design with the existing HyV approach, analyse the differences and present the benefit of using virtio-rdma driver.

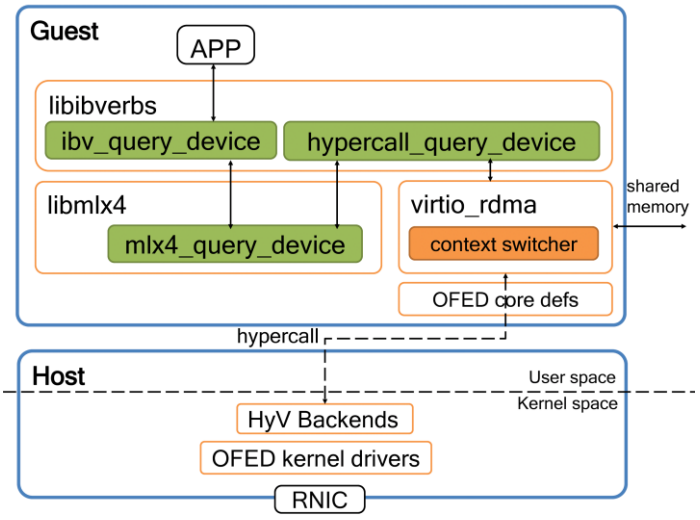

Figure 2. A detailed description of prototype II front driver.

Figure 2 presents an example of using the Prototype II frontend driver. The frontend driver is designed to work with minimum support of OFED kernel data structures and without any extra kernel providers or drivers. Only two user space libraries are reused from OFED: the *libmlx4* library and the *libibverbs* library. The *libmlx4* library provides the basic hardware information such as vendor ID, device ID and hardware specific parameter formats. The *libibverbs* library exposes the verbs API to the user application, composes and sends the hypercall messages to the the host driver. In addition, we simplified both libraries to keep only the fundamental support that is necessary.

In addition to the two libraries above, the *virtio-rdma* serves as an important component on the guest side of prototype II design. Virtio-rdma is a user-to-kernel context switcher. It translates the user space data structures to the corresponding kernel ones, for instance *device handle* and *PQs*. As previously mentioned, OSv uses a single address space implementation without separation of user space and kernel space on the guest side. With this feature, we are able to avoid a number of context switches which are carried out in guest OSes which do have two address spaces. While using the *virtio-rdma* switcher, the frontend driver no longer sends *uverbs* commands to the OFED kernel drivers, for example, the traditional OFED kernel provider ib_uverbs opens up the uverbs0 device to exchange user and kernel space commands. Instead, *hypercalls* are directly made from virtio-rdma with a simple user to kernel space structure translation. As a result, unnecessary context switches have been eliminated, saving up to 5 library calls per verb command. The backend driver will receive the hypercall command and execute the job as is.

Compared to our frontend driver design, the existing HyV implementation are mostly used for guest applications which separate user/kernel spaces and with full OFED support. The HyV guest drivers abstract the kernel verb calls which are capable of working with OFED kernel

modules. Thus the HyV implementation is highly dependent on the corresponding OFED modules. Moreover, OFED currently does not provide any support for OSv or other library OSes.

In summary, the proposed frontend driver provides virtualized RDMA device drivers on the guest side. It is a lightweight solution which has minimal dependency on OFED modules. This driver is suitable for library OSes with single address spaces,

4.4 Example

In this section, we present how to install and run OSv with vRDMA patches. We also present how to check the device information and run benchmark tests. The following commands are used to set up the guest side. A detailed description on setting up the host is given in Deliverable 3.2 [23].

Download OSv source containing virito-rdma implementation (the public URL [24] will be available soon, as the source code of virtio-rdma is under internal review process).

Install and setup OSV with vRDMA support with the following commands:

```
$ cd /path/to/mike-osv
$ sudo ./scripts/setup.py
$ git submodule update --init --recursive
$ sudo ./scripts/build image=vrdma-ulibs,cli -j4
```

To check if RDMA devices are available, run with the OSv run.py script:

```
$ sudo ./scripts/run.py  -d -e "/tools/ibv_devinfo.so" --verbose
```

After run the run.py command, it is possible to run the same command with QEMU commandline:

```
$ sudo qemu-system-x86_64 -s -vnc :1 -m 5G -smp 1 -name "vrdma" -chardev
stdio,mux=on,id=stdio -mon chardev=stdio,mode=readline,default -device isa-
serial,chardev=stdio -enable-kvm -drive file=/path/OSv/usr.img,if=virtio,cache=unsafe -
device virtio-rng-pci -device virtio-hyv-pci,config_path="/path/hyv_config" -netdev
bridge,id=eth0 -device virtio-net-pci,netdev=eth0,id=br0,mac=52:54:00:12:34:56 &
```

The output of the above commands looks like:

```
OSv
4 CPUs detected
Firmware vendor: SeaBIOS
```

```
... OSv initialization outputs ...
hca_id: mlx4_0
        transport:                      InfiniBand (0)
        fw_ver:                         2.36.5000
        node_guid:                      f452:1403:006f:5dd0
        sys_image_guid:                 f452:1403:006f:5dd3
        vendor_id:                      0x02c9
        vendor_part_id:                         4099
        hw_ver:                         0x0
        phys_port_cnt:                  2
                port:   1
                        state:                  PORT_ACTIVE (4)
                        max_mtu:                4096 (5)
                        active_mtu:             4096 (5)
                        sm_lid:                 3
                        port_lid:               3
                        port_lmc:               0x00
                        link_layer:             InfiniBand
                port:   2
                        state:                  PORT_ACTIVE (4)
                        max_mtu:                4096 (5)
                        active_mtu:             4096 (5)
                        sm_lid:                 1
                        port_lid:               1
                        port_lmc:               0x00
                        link_layer:             InfiniBand

Requested termination of /libhttpserver.so, waiting...
VFS: unmounting /dev
VFS: unmounting /proc
VFS: unmounting /
Powering off.
```

4.5 Key Performance Indicators

The MIKELANGELO Grant Agreement lists the following Key Performance Indicators which are relevant to vRDMA:

- KPI2.1: relative efficiency of virtualized I/O between KVM and sKVM (developed in the project).
- KPI3.1: The relative improvement of efficiency of MIKELANGELO OSv over the traditional guest OS.
- KPI3.2: The relative improvement of efficiency [size, speed of execution] between the Baseline Guest OS vs. the MIKELANGELO OSv.

In D4.1 [4], we presented the benchmark performance of vRDMA prototype I, which showed a big improvement of the bandwidth and latency comparing to the traditional virtio based network interface. Prototype II is targeting at achieving more improvements than prototype I. For KPI 2.1, the hypercall implementation in virtio-rdma device has been developed to improve the virtualized I/O communication between the host and guest. For KPI 3.1 and KPI

3.2, the virtio-rdma component on OSv has been implemented, which will improve the network communication performance comparing to the traditional guest OS.

The performance testing framework *perfTest* has been used to benchmark the two key factors of network I/O performance, i.e. bandwidths and latencies. For OSv, *perfTest* has been adopted to be compiled within the guest image. The perfTest, adapted for OSv will be published on MIKELANGELO's GitHub [24] after the internal review process is finished.

To build *perfTest*, run commands

```
$ sudo scripts/build image=perftest,vrdma-ulibs,cli  -j4
```

We test write- and read- bandwidth with *ib_write_bw* and *ib_read_bw*. For example, following commands can be used to run the benchmark application in server mode:

```
$ ./scripts/run.py  -d -e "/tools/ib_write_bw.so" --verbose
$ ./scripts/run.py  -d -e "/tools/ib_read_bw.so" --verbose
```

And following commands can be used to run the benchmark application in client mode with the assigned IP address to the OSv guest:

```
$ ./scripts/run.py  -d -e "/tools/ib_write_bw.so 192.168.2.220" --verbose
$ ./scripts/run.py  -d -e "/tools/ib_read_bw.so 192.168.2.220" --verbose
```

Initial performance results on Ubuntu guests have been presented in D3.2 [23]. Similar performance results on OSv guests will be available soon.

5 Application Package Management

The importance of application packages and management thereof has been established early in the project. Deliverable D2.16 (The First OSv Guest Operating System MIKELANGELO Architecture) [2] has described this as one of the fundamental topics that must be considered whenever new platform is being introduced. In order to gain initial traction the early adopters must be presented with a compelling story and user-friendly first experience. The creators of OSv have always paid special attention to this and offered a curated list of commonly used applications, such as database, web and application servers, different runtime environments (e.g. Java, Ruby, Erlang) as well as several simple examples supporting users getting started.

This section is a continuation of the deliverable D4.7 (First version of the application packages) [5]. The previous deliverable presented the initial version of the tools being developed in the MIKELANGELO project simplifying the application package management. The deliverable has also listed the preliminary version of the packages that have already been provided as simple, standalone and reusable components. These packages were mainly focusing on the use cases that have already been properly ported onto the platform that the MIKELANGELO project is proposing.

Deliverable D4.7 defines a packages as a "*bundle of one or more components that work together to achieve one common goal. The package may contain an application binary and one or more libraries used by the former. It may also contain one or more configuration files used to alter the behaviour of the application or just some input data used to operate on when started. The application package may further be standalone containing all the necessary bits and pieces for proper operation or comprised of several other application packages.*" This notion of a self-contained package is at the very core of the package management system proposed and implemented within the project. Instead of requiring users to deploy complete development toolchains in their environments, application users are allowed to compose runnable images out of prepackaged components. This significantly simplifies the entire process and reduces time needed to deploy an application. As has been shown by the Aerodynamics use case (OpenFOAM Cloud demonstrator [38]) this furthermore simplifies the integration workflow because the provided tools support all the required functionalities.

This section is structured as follows. We start with the review of the task T4.3, responsible for the implementation of the application packaging mechanism. This will provide additional context to the work described in this section. Next, we continue with the review of all requirements that have been presented in past reports along with a short evaluation. We are also introducing new ones as they have been identified. We proceed with the description of the major advances since the last official report. These include the features that have been

released as part of milestone MS3 (M18) public release as well as those that correspond to this report, i.e. the milestone MS4 (M24) release.

5.1 Task T4.3 Overview

The main aim of this task is to package the applications that are required by the use cases of the project, namely the big data, bones and aerodynamics simulations and the cloud bursting use case. These use cases have significantly different requirements for the application packages, ranging from small- and large-scale native applications to Java based parallel workers. The review of existing mechanisms for packaging OSv apps, presented in report D2.16, have shown that they have certain limitations that should be addressed within the project.

This task has therefore started with a thorough evaluation of available tools and collection of initial requirements simplifying management of application packages. The initial study has shown that the existing Capstan [12] tool provides a solid baseline for extending the capabilities which resulted in a series of patches oriented towards the exploitation of packages, instead of only virtual machine images. Further work on the integration, primarily into the cloud environment (OpenStack), have later shown that the decision to extend Capstan was indeed correct: Capstan source code was easy to extend and had no major bottlenecks. On the other hand, it became evident that the integration with infrastructure provisioning services should not be tightly coupled with the application package management. To address the latter, we have designed an extensions for UniK [13]. UniK is an open source project focusing on the overarching unikernel management. It supports several popular unikernels and infrastructure providers (details are presented in Section 5.6.2, Holistic Unikernel Management Using UniK). Our improvements to the UniK project involve integration of the new capabilities of Capstan and support for OpenStack cloud provider.

The following subsections describe the work, results and outline related to the application package management in the project.

5.2 Requirements Overview

Report D2.20 (The intermediate MIKELANGELO architecture) [7] presented the state of the high level requirements for the application package management. The same requirements, along with the state at milestone MS4 is presented in this section.

1) **Elimination of external dependencies (initial requirement)**
    a) MS3 status: 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.

b) MS4 status: Along standalone Capstan that still requires QEMU/KVM, a Docker container is provided. This significantly simplifies execution of Capstan commands in environments that don't allow for additional customisation. This approach has also been integrated into UniK project presented in Section 5.6.

2) **Change of the underlying architecture (initial requirement)**
   a) MS3 status: 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.
   b) MS4 status: As we explain in Section 5.6 advanced integration with cloud environment is now built into UniK, whose architecture is in line with this requirement (RESTful API and transparent use of the services provided).
3) **Support for other guest operating systems (initial requirement)**
   a) MS3 status: 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.
   b) MS4 status: Further attempts have been made to test some of the use cases of the MIKELANGELO project to run on different unikernels (rump and partly IncludeOS). Because of different limitations, we have decided to omit this requirement to be part of MIKELANGELO packaging support.
4) **Run-time options (initial requirement)**
   a) MS3 status: 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).
   b) MS4 status: This has been worked on intensively since MS3. Even this preliminary support allows application packages to provide their default boot

commands as part of their manifest. This resembles how packages in existing systems provide shell scripts that are used to simplify execution of applications. Details are presented in Section 5.4.

5) **Finalise cloud integration**
   a) MS3 status: 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.
   b) MS4 status: Image and instance management has been built into Capstan and UniK tools. Further integration with more advanced networking and external volume management is planned for the next iteration. Section 5.6 presents the status in more detail.
6) **Abstraction of cloud provider**
   a) MS3 status: 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.
   b) MS4 status: This has not been addressed in this iteration. By relying on UniK for infrastructure provider, this requirement has lost its priority.
7) **Package hub**
   a) MS3 status: 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.
   b) MS4 status: Following the release of the initial version of MIKELANGELO Capstan and in particular after publishing of the user's manual [25] we have received few requests for this feature. Besides these requests, we have also discovered that integration is cumbersome when the entire package repository has to be installed manually. Work on package hub is presented in Section 5.3.

5.3 Package Repository

The package repository has been one of the fundamental missing components since the initial release of the modified Capstan tool (MPM) [26]. In order to use existing packages, the user was required to download them and manually import them into their local package repository. Even though some users have downloaded the packages, they described the whole process as cumbersome.

The idea for a centralised and remotely accessible package repository is similar to those of APT/RPM repositories or the Docker Hub [27]. The repository stores all available packages and the corresponding metadata information. The structure of the repository provided by the MIKELANGELO project separates two distinct:

- Base images: contrary to the existing Capstan tool, the MPM tool uses small base images that are to be extended into target images. They typically contain only the OSv loader that is used to instantiate the image. Multiple versions of the loader image are provided to support multiple versions of the OSv kernel.
- Packages: packages are stored as archives of all files necessary for the proper operation of the package. Every package is accompanied with a separate metadata descriptor providing details of the package to the users.

The client tool has been extended to support the use of the remote package repositories. Following functionalities have been added

- Query allows searching for packages in the remote repository by the package name.
- Pull (`capstan package pull`) downloads the package from the remote repository and imports it into a local one.
- Push allows repository maintainers to add packages to the repository. Prior to uploading the package into the repository, package is validated. User needs administrative privileges in order for push to succeed.

Existing commands of the tool (e.g. collecting package content and composing VM images) have also been extended allowing users to request automatic pulling of the packages that are missing in the local repository.

5.4 Runtime Configuration Support

Our internal evaluation of the initial version of MPM has shown that packages are suitable for building complex applications out of manageable building blocks. However, users were required to understand the package in entirety in order to use it properly. Although the contents of the package can be reviewed at any time there is no easy way to know how to execute the unikernel application as it is not set by the packages. Due to the most complex boot command, support for configuring a Java Virtual Machine has been introduced initially.

Unfortunately, the command is not inherited when a package providing such configuration is used by another application. For example, users who are using the HDFS package that sets the command line to execute HDFS datanode [28] need to make a new package descriptor with a dependency set to HDFS package and optionally include the settings files that need to be updated (e.g., setting the location of the namenode and number of replicas to keep). Since the java command to launch a new data node is not inherited, the user is also required to construct the appropriate command. This can either be done manually or by inspecting the manifest of the HDFS package and copying it into her own package.

This process becomes even more cumbersome when using packages running native applications or applications based on alternative runtimes (Node, Erlang, ruby, …). The initial version of the Capstan tool provided no support for application packages to export the default commands, leaving users on their own to investigate how to use the package properly.

The **runtime configuration support** intends to resolve this issue. It allows package managers to integrate one or more preferred command lines. These commands are inherited automatically by the Capstan tool allowing package users to invoke them automatically without knowing the exact details of the command. As a side effect, these commands offer self-documenting system allowing users to review the possible commands. The following example is taken from OpenFOAM's simpleFoam application package. It depends on the Open MPI package providing the mpirun command.

```
$ --env=WM_PROJECT_DIR=/openfoam /usr/bin/mpirun -np 2 -x MPI_BUFFER_SIZE=20000000 --allow-
run-as-root /usr/bin/simpleFoam.so -case /case -parallel
```

The command first sets the environment variable mandatory for the execution of the OpenFOAM application. Then it uses the mpirun command to launch the simpleFoam solver using two parallel workers. This command, provided in the run configuration, can be used directly in case it suits user's needs. Alternatively, the user can review this command and set it manually according to run the required workload.

Run configurations are currently all stored in a single package manifest file, *meta/run.yml*. The basic structure is independent of the underlying runtime type and includes three top-level elements:

- **runtime:** defines the type of the runtime (currently supported are native, java and node)
- **config_set_default**: the name of the default run configuration (optional)
- **config_set**: one or more named run configuration adhering to the requirements of the chosen runtime type. Each runtime supports setting the environment variables.

The following subsections briefly present each of the available runtimes.

5.4.1 Native Application Runtime

Native runtime is the simplest type and is used for specifying commands for compiled applications, such as OpenFOAM and Open MPI. The only mandatory configuration option (*bootcmd*) allows for specification of the run command to execute when user opts for one of the available options.

```
runtime: native
config_set:
  help:
    bootcmd: --env=WM_PROJECT_DIR=/openfoam /usr/bin/simpleFoam.so -help

  simpleFoam:
    bootcmd: --env=WM_PROJECT_DIR=/openfoam /usr/bin/simpleFoam.so -case /case
```

Using the additional option for specifying environment variables (*env*) the simpleFoam configuration could be rewritten as

```
  simpleFoam:
    bootcmd: /usr/bin/simpleFoam.so -case /case
    env:
        WM_PROJECT_DIR: /openfoam
```

5.4.2 Java Application Runtime

Java runtime separates application specification into four configuration options typically used when running applications inside Java Virtual Machine: the fully qualified name of the main class, the classpath containing all necessary code, application command line arguments and JVM arguments. Below is an example using a single run configuration:

```
runtime: java
config_set:
  mainapp:
      main: eu.mikelangelo-project.java.MainApp
      classpath:
         - /src
         - /src/main
         - /src/main/java
      args:
         - --host 10.0.6.66
         - --port 8008
      vmargs:
         - Xmx256m

config_set_default: mainapp
```

In case a single configuration is given, Capstan will automatically treat it as a default. Consequently, the *config_set_default* option is not mandatory.

The specific run configuration will result in the following command line for running Java application within OSv:

```
java -classpath /src:/src/main:/src/main/java -Xmx256m \
     eu.mikelangelo-project.java.MainApp --host 10.0.1.11 --port 8008
```

5.4.3 Node.JS Application Runtime

Node.JS currently supports a single mandatory configuration option (*main*). The runtime will ensure appropriate Node interpreter is going to be launched.

```
runtime: node
config-set:
   app:
      main: /app.js

   # This is second named configuration
   alternative:
      main: /alternative.js

config-set-default: alternative
```

During virtual machine composition, runtimes can require additional package dependencies. These are implicitly added to the application without user having to change application package manifest. Java runtime will be added as a dependency to the Java Virtual Machine, while Node.JS will require Node engine.

Each run configuration is stored as a separate file in the composed OSv virtual machine. For example, the above Node.JS configuration will result in three files

- */run/app* whose content is *node /app.js*
- */run/alternative* whose content is *node /alternative.js*
- */run/default* that is a link to the file */run/alternative*

To run any of the existing run configurations using Capstan tool, one simply requests the named configuration, e.g.

```
$ capstan run app
```

to run *node /app.js* from the Node.JS example or

```
$ capstan run simpleFoam
```

To run *--env=WM_PROJECT_DIR=/openfoam /usr/bin/simpleFoam.so -case /case* from the Native example above.

5.5 Updating Virtual Machines

Our evaluation using different kinds of applications has shown that changing only a small part of the application being composed into a VM image consumes significant overhead. This became evident when user's application is just a small portion of the overall application content. Two cases where this has become evident are:

- Deployment of HDFS cluster: in order to deploy such a cluster the end user is required to modify several properties in the HDFS configuration (for example, the name of the given data node, the location of the master node, the minimal required replication, etc). These changes are done in configuration files that are used during application composition. Every change in the application configuration required complete recomposition of the target VM image which took approximately 30 seconds due to the size of the Hadoop distribution.
- Node.JS application: Node.JS applications typically result in hundreds or even thousands of files that are provided by the required libraries. To demonstrate this, we have used a simple Node.JS application looking up words from a fixed dictionary [29]. The source code consumes approximately 4 kB of disk space. Besides 10 MB consumed by the dictionary, the installed application will consume additional 90 MB of libraries and additional tools, all stored in approximately 46000 files.

In both of these two cases most of the time only a small subset of the files are being changed. Therefore, we have revised the package composition and designed a simple caching mechanism. While deploying package content onto target VM image, hash value of the file, directory or link is stored in a VM-specific cache. When doing consecutive updates of the VM image, the cache is consulted and only modified content is updated (for example, modified JAR file or a new configuration). This allows for composition without any significant overhead compared to previous VM composition mechanisms.

The above Node.JS application took approximately 90 seconds to complete VM image composition. On the other hand, updating an already composed VM image with changes done only to our source code finished in less than 6 seconds overall time (boot OSv instance, use the cache to check for changes and upload modified files).

5.6 Integration with the Cloud

Ease of use has been the main motivation for designing extensions for the Capstan tool. So far we have focused mainly on the topic of virtual machine image composition. The composition workflow allows users, either software developers or just application end-users, to create full-blown application images that they can deploy on their own or rented infrastructures. Similar to how Linux containers (Docker [30] in particular) simplifies the

execution of containerised applications, we are closing the gap between ordinary processes and processes running in lightweight virtual machine images, i.e. unikernels.

Although deployment of these images can be manual using existing tools, the application packaging went further by working intensively on integrating backend provider support. The purpose of this support is to allow transparent execution of unikernels regardless of the deployment target: local, private OpenStack deployment or a public cloud. To this end we have initially extended the Capstan tool to support automatic deployment and provisioning of OSv-based virtual machine images. OpenStack has been chosen for the initial proof-of-concept backend provider. Two integrations were provided so far:

1. Integration into Capstan: simplified version of the cloud integration allowing deployment of VM images into OpenStack image service (Glance) and running OSv-based VM instances using the compute service (Nova).
2. Integration into UniK [13]: holistic integration of the application package management with the unikernel and cloud management. Part of this integration is also implementation of the OpenStack provider.

The following subsections describe both of these in more details.

5.6.1 OpenStack Provider for Capstan

As we have described in the previous section, the provider for Capstan tool is straightforward and supports only two basic operations: publishing OSv images into OpenStack and provisioning them. The tool has been extended to resemble local execution as far as possible, using the same command line switches and completely reusing the approach for image composition. The major difference is related to the user authentication and authorisation. Whereas a local environment assumes the user running Capstan tool is the owner of the local package and image repository, the OpenStack provider needs to authenticate the user against the provided backend API (Keystone - Identity service, [31]).

Publishing images. Publishing images consists of composing the image in a local repository (temporary image), authenticating the user, creating a new image and uploading the image file. The composition itself uses the same information as in the case of local images. In order to optimise deployment, the size parameter is used only as the guideline or minimum value. Actual size is determined by inspecting available flavors whose disk capacity is equal or larger than the requested size. This is important because contrary to Linux-based instances where disk size is determined at the time of VM provisioning OSv-based instances are provisioned with a disk partition of fixed size, specified at the time the image is created.

Consequently, to support more complex scenarios, the image publishing also supports additional option to request the specific flavor by its name or ID instead of just providing the required image disk size.

Launching instances. Once the image has been pushed to the OpenStack image service, it can be launched easily via the Capstan tool exactly the same as if it were launched in a local environment. Required resources can be given either via specific options (memory and virtual CPU cores) or via flavor name or ID. The former option will look for the flavor that best matches the requested resources, if there is no exact match. For example, if the user requests 2GB of RAM and 2 CPU cores, but there are only two flavors exceeding these requirements (first with 4GB of RAM and 2 CPU cores and the other with 4GB of RAM and 4 cores), the first option will be chosen. In case the size of the root partition is larger than the one used for pushing the image to OpenStack, it will not be resized automatically. However, if the target image is smaller, the provisioning of instances is going to fail.

The Capstan tool furthermore allows provisioning of multiple instances from the same image simultaneously. All instances are created using the same flavor type. This feature has been used while running the Hadoop HDFS [32] experiments.

While these two capabilities are essential for the integration with the cloud provider, difficulties experimenting with various workflows have been noticed. Once the instances are provisioned, there is no way to query their status, restart or shut them down. In order to accomplish these tasks, users are required to access the cloud either through the dashboard or by using the corresponding cloud API manually.

About the same time these integrations were being implemented, a new open source project, UniK [12], was released by EMC [33]. To understand the general direction of the project, we have monitored this project closely checking the code releases and their public Slack channel. Once it became clear that extending our support for cloud integration into Capstan would result in duplication of work on two open source projects, we decided to discuss the potential for collaboration. Details on this line of work is presented in the following section.

5.6.2 Holistic Unikernel Management Using UniK

Contrary to our efforts on simplifying the management of application packages and composition of runnable OSv-based virtual machines, UniK focuses on the holistic management of arbitrary unikernels in different infrastructure providers. Currently supported unikernels are, besides OSv, rump [34], IncludeOS [35] and MirageOS [36]. Multiple provider types are already available: local (VirtualBox, QEMU and Xen), private infrastructure (vSphere) and public clouds (Amazon Web Services and Google Compute Engine). At the time we decided to contribute to the project, there was no OpenStack provider support and the

authors were willing to review our contributions in order to improve the general purpose of the UniK project.

Main contributions of the MIKELANGELO project, already accepted in the upstream repository, are the following:

- **Capstan Integration.** UniK project has already integrated the original Capstan tool [12], initially developed by Cloudius Systems. This supports building virtual machine images only for Java applications. In order to extend the support for a wider range of applications, the MIKELANGELO project contributes the integration of the improved Capstan, including access to the MIKELANGELO application package hub and the simplified and more customisable VM image composition.
- **OpenStack Provider** is an abstract representation of the specialised management components (image, instance and volume management). It is the layer that authenticates and authorises users for the other components of the provider.
- **Image Management** consists of image composition, image publication in OpenStack image service, image querying and removal from the image service. This contribution reuses the approach for working with OpenStack image service that was used while adding support into Capstan.
- **Instance Management** enhances image management by allowing provisioning of new instances out of existing images, querying the status of the provisioned instances and performing various power operations (restart, termination, etc). Similar to image management, reuse of existing functionality from the Capstan has be employed.
- **Volume Management** is part of the UniK backend provider, however it has not been integrated with OpenStack yet. Initial research on this topic is presented in the future work section below.

5.7 Other Improvements

Improvements presented in this section are considered minor, however still important for the successful release of a complete application package management suite. All of these have been a result of internal evaluation of the tool. Two different perspectives have been used while choosing the relevant extensions:

- **End user perspective.** End users are interested in using MPM tools for composing, running and managing application packages and runnable virtual machine images. They are mainly looking for ease of use of repetitive tasks.
- **Integration perspective.** While working on the integration of MPM tools into cloud/HPC environment and the specific use case applications, various missing features have been observed. Because MIKELANGELO is targeting a fully integrated stack, these missing features have already been addressed to some extent.

The following list is a subset of advances since the first version of the tools.

Package initialisation. Previously, the user was required to initialise the package from within the package directory. The patch was provided allowing them to specify arbitrary location in the filesystem. Capstan uses this new information to initialise the package at the given location. The parameter is optional, still allowing users to use the current path as the package.

Package dependencies. Further to the previous improvement, support for specifying the list of required packages during initialisation has been added. Primary request for this came from the integration point of view because it was frequently necessary to compose VM images directly and without manually writing package manifest files. Using the "--require" command line switch, Capstan can request the given package while composing.

Better support for package content. Using increasingly more complex packages (for example native OpenFOAM simulations, large Node.JS packages and MySQL) has revealed several issues with the management of package content. Albeit the content was stored, it was not properly deployed in the VM images resulting in incomplete applications failing to launch. Changes were thus introduced supporting better management of symbolic links, empty directories and the order of creation thereof. A patch resolving an issue deploying large packages consisting of thousands of files allowed users to use the Erlang runtime.

QEMU lookup improvements. Because Capstan relies on the QEMU for some under-the-hood operations, it is a vital component of the installed system. We have added several improvements allowing automatic lookup of QEMU's components. To facilitate execution in an environment not supporting advanced QEMU features, an option is offered to the user allowing to opt-in for a completely emulated environment. This results in a significantly slower execution, however all capabilities are now supported in such environments.

Overriding package content. While working on the Hadoop HDFS application packages we realised that the specialised application requiring other packages has no way of overriding parts of the dependent application (for example a configuration file). A patch was thus introduced changing the order of package composition ensuring that every package can override the content of any of the required packages. Further to changing the configuration, this enables users to modify parts of the application by simply changing one or more application libraries.

Capstan configuration. This minor extension to the way Capstan loads its settings allows users to replace environment settings with repository-specific configuration. Within the local Capstan repository, configuration file allows for setting the root of the remote application

package hub and some additional options for the virtualisation (for example whether to use KVM or not).

5.8 Application Packages

Pre-built application packages are important contribution of WP4 when it comes to an overarching technology stack of the MIKELANGELO project. Initial set of packages that have been provided by the consortium and is described in D2.20 [7] has been expanded with the following packages.

NodeJS [18]. The package contains Node.js runtime environment and supports running arbitrary application on top of OSv. It has been successfully tested with a large application that contains thousands of files.

Erlang [20]. The package contains shell for Erlang programming language. User can execute arbitrary Erlang script using the shell or type-in the command directly. At the moment, only basic testing was performed.

MySQL [19]. The package contains MySQL server that serves a database. Default administrator username and password is set so that user can quickly boot OSv and configure database using TCP connection. Alternatively, user can provide SQL script to automatically configure database on boot. Package has been tested with a web application that managed to use MySQL database (inside OSv) as a backend storage successfully.

Additional OpenFOAM Solvers. Multiple OpenFOAM solvers have been added to the MPM repository. Each solver is packaged as a separate MPM package and only contains a single binary (application) as all the remaining libraries are already available in the core OpenFOAM package. Below is a list of the newly packaged solvers. One can consult OpenFOAM documentation [17] to learn what each solver is used for.

- pimpleFOAM
- pisoFOAM
- porousSimpleFOAM
- potentialFOAM
- rhoPorousSimpleFOAM
- rhoSimpleFOAM

These OpenFOAM solvers have been successfully tested on the Aerodynamics use-case.

5.9 Key Performance Indicators

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

Objective	Key Performance Indicator	Status
O1: To improve responsiveness and agility of the virtual infrastructure. By responsiveness and agility we denote the efficiency of setting up a new set of virtual machines and their pulling back. The objective is to improve the time to "burst" cloud or to set-up and tear- down a set of virtual machines, either with the same or with a different application.	KPI1.2: Relative improvement of time to change the installed application on a number of virtual machines	Indirectly addressed
O4: To provide application packaging, improving portability and deployment of virtualized applications.	KPI4.1: A system for appropriately packaging of applications in MIKELANGELO OSv is provided.	Already addressed with ongoing improvements
	KPI4.2: A relative improvement of efficiency [time] between deploying a packaged and non-packaged application.	Already addressed with ongoing improvements
O7: To appropriately demonstrate the project's outcomes on ground-breaking use-cases.	KPI7.1: All use cases appropriately demonstrated	Partially addressed due to the varying status of use cases

A more detailed assessment of these KPIs based on the current version of MPM is presented in the following subsections.

KPI 1.2 Relative improvement of time to change the installed application on a number of virtual machines

The status of this KPI has been marked as *indirectly addressed* because this task is not focusing on it explicitly. However, as the description of KPI 4.2 status below shows, we have managed to extend the application packaging tools to support application updates to the existing images that are significantly faster than any of the previous techniques by storing the

information about the VM image content. This information is used during application updates to deploy only the modified content onto target VM images.

KPI 4.1 A system for appropriately packaging of applications in MIKELANGELO OSv is provided.

The current report has presented the advances in the application package management tools. Most of these advances are related to this particular KPI with a focus on delivering a tool, suitable for management of lightweight unikernel images using OSv.

KPI 4.2 A relative improvement of efficiency [time] between deploying a packaged and non- packaged application.

Report D4.7 [5] has introduced the improvements of efficiency between deploying a packaged and unpackaged application, comparing existing tools (build scripts and Capstan) against the MIKELANGELO package manager (MPM). That comparison has shown that MPM is slightly slower than Capstan, however it is still significantly faster when using pre-compiled packages when compared to the build scripts.

Since the last report, MPM has been enhanced with a support for caching. The cache stores the information about previously uploaded files and directories and improves the performance of consecutive VM builds as it uploads only the modified content. In case the modifications are minor, these builds are practically instantaneous. For example, the HDFS application is a large Java application with lots of different libraries and configuration files. In case the user is only changing the configuration of the deployed application, this mechanism will deploy only the modified configuration file to the existing VM image. Alternatively, if only a subset of Java libraries (JAR file) needs to be modified, the caching mechanism will request only the changed files.

KPI 7.1 All use cases appropriately demonstrated

The application package management supports this KPI twofold. First, it is providing a tool suitable for the preparation of packages suitable for execution in virtual environments using OSv. Second, it supports the use cases in preparation of the corresponding application packages. In doing so, it provides the mandatory base packages (for example Open MPI for HPC use cases and OpenFOAM solvers for the Aerodynamics use case).

As of the time of writing this report, the packages required by the Aerodynamics use case have been prepared using the latest versions of OSv source tree and a stable version of OpenFOAM platform (2.4.0) used by the use case owner. These packages have been put in use in HPC and cloud environments. Packages for several of the most popular OpenFOAM application solvers have been provided in a central package repository allowing external users to reuse them in their own simulations. Packages are provided in such a way that

solvers are completely interchangeably, meaning that one input data can be processed using any of the supported solvers.

The status in regards to the following use cases is as follows:

- Bones simulation: base packages have been used to compose target OSv images, however no packages have been provided for the application due to certain limitations in the application license itself.
- Big data: easily extendable packages for the HDFS have been provided. These allow users to deploy a complete distributed file system using OSv-based master- and data-nodes storing the data.
- Cloud bursting: no additional applications have been required at this point

The current evaluation of KPIs show that significant improvement to the application package management has been achieved. In the future we are going to focus primarily of KPI 4.1, further improving the usability of the provided tools. KPI 7.1 is also going to be addressed in collaboration with WP6 resulting is as many publicly available application packages as possible.

5.10 Exploitable Result

Application packaging is mainly contributing to the exploitable result ER5, the MIKELANGELO Package Management toolchain. The goal of this exploitable result is to open-source all changes and also work towards upstreaming the changes back to the originating repositories (Capstan, operated by Scylla and UniK by EMC). We've been successful merging several of the key changes related to the integration of the modified Capstan and OpenStack provider into UniK project. However, we are currently lacking merges with the original Capstan repository. There are two reasons for this:

1. While simplifying the application management workflow we have introduced some changes that might break backwards compatibility. We plan to review the changes made to both repositories in order to analyse the possibility of merging them into a single product repository.
2. Although initial Capstan tool was designed to simplify creation of OSv-based virtual machine images, it has recently lost its momentum by the original authors. Partners participating in WP4 are already discussing whether merging changes of the MIKELANGELO project into an upstream repository is sensible due to the fact that the core maintainer is no longer actively involved in the project.

Deliverable D7.3 provides more information about exploitable result ER5.

5.11 Outline and Future Work

5.11.1 Future Work on Capstan Tool

Capstan tool converges towards user friendliness, yet there are many things left to be improved. The overarching goal of our work on the application package management is to make execution of unikernels as similar to running ordinary processes as possible. Below is a list of the areas that we are planning to work on in the next iteration.

Boot command with parameters that are evaluated on boot. As mentioned earlier each run configuration is being stored in a separate file (e.g. `/run/alternative`) in the composed OSv virtual machine during unikernel creation. User can then easily boot the unikernel by picking one of the stored configuration names that fully defines boot command. But in some cases users might need to pass parameters on boot time that should be integrated into the boot command to reflect their environment. One example is the value of environment parameters where user e.g provides database URL. It is cumbersome that user needs to modify runtime manifest (for example to set the new database URL) and then rebuild the unikernel only to reflect the change in the environment. In the future we should therefore consider a way to use parametrized boot commands.

Runtime manifest hierarchy. The package author is currently able to prepare a fully-defined runtime manifest before publishing the package in a public repository. An end user can then use it as a template to create runtime manifest for her own private package. But the process of manually editing third-party runtime manifest is cumbersome and requires deep knowledge of the third-party package content. If the author of the public package was able to prepare the runtime manifest in a way that would parametrize some settings and if end user was able to somehow just pick desired configuration set and provide parameter values for it, then the user experience would increase. In the future we should therefore consider a way to support runtime manifest hierarchy.

Package versioning. At the time being, package versioning is not supported yet. The package author needs to assign a different name to the package for every new version if they are to coexist in the repository. This has not been a problem so far since Capstan is being used in a controlled environment where we needn't care for backward compatibility. It is, however, important to address this topic prior to going live. Package versioning needs to deal with two version numbers for each package:

- Packaged software version (e.g. NodeJS 4.4.5)
- Software packaging version (e.g. NodeJS 4.4.5 - packaging 0.0.1)

Package author may not be the author of the software she packages, but only the one that has patched and compiled it for OSv. We must therefore distinguish between what version of

the software was compiled (Packaged software version) and what iteration of the compiling it was (Software packaging version). Semantic versioning [37] has to be considered when dealing with various package versions in manifest files in order to allow for generic version requirement specifications.

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.

5.11.2 Future Work on UniK Platform

Basic integration of Capstan tool into UniK platform has been carried out in present iteration. In the next iteration we plan to further focus on the following capabilities.

Use UniK Hub as a central repository for Capstan packages. UniK platform has recently 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.

Further integration of OpenStack provider. Basic OpenStack provider was implemented in the present iteration. UniK platform is able to upload unikernel to OpenStack Glance and run it with OpenStack Nova. We plan to extend support for external volumes. In terms of unikernels it is perhaps even more important to separate the application and the data the application works on than in a general purpose operating systems. The lightweight nature of the unikernel allows us to bootstrap a multitude of highly efficient workloads that are accessing external storage while processing the data. Similar to how integration with image and compute services allowed us to simplify use case workflows, ability to attach volumes is enhance the usability.

Support cloud orchestration templates. UniK platform is currently focused on a single unikernel, but in real life it is quite common that user needs multiple unikernels to deploy the application. Web server, for example, is usually deployed with two unikernels: one for business logic and one for database. In the future we should therefore consider extending

UniK platform with support for orchestration templates where user could describe the target application and UniK would build and deploy unikernels to support it. Research needs to be conducted, however, to verify that this feature is feasible for UniK platform. This has been partly addressed in cooperation with the Aerodynamics use case. This use case has already provided several OpenStack orchestration templates deploying and interconnecting OSv-based unikernels running OpenFOAM simulations (consult report D6.3 First report on the Use Case Implementations for details).

6 Key Takeaways

This document accompanies the second source code release of the four major components of the MIKELANGELO guest operating system: **OSv**, **Seastar**, **vRDMA** and **MPM**. In this document we described:

- We described these four components:
  - OSv is our unikernel for running an existing Linux-compatible application on the cloud with better efficiency;
  - Seastar is a set of new C++ APIs for newly written asynchronous applications to achieve even better performance;
  - vRDMA allows applications in the guest to make use of RDMA hardware in the host for faster guest-to-guest communication;
  - MPM, the Mikelangelo package manager, is a set of tools for conveniently composing the OSv kernel and the user's application and libraries into a ready-to-run image.
- We described what is included in this source code release, how to get it, and how to use it.
- We mentioned that all components are being released as open-source projects. Much of this work went directly into the project's main repository. That makes MIKELANGELO's improvements easier to maintain, more tested by the general public, and increases these changes' impact and dissemination.
- We described the improvements to these components done under Work Package 4 over the last year, since the last iteration of this deliverable.
- The primary goal of this development has been to run MIKELANGELO's use cases, and to do so conveniently and with better performance. An additional goal was to make the components we developed useful to additional users, besides the four chosen Mikelangelo use cases; Part of this goal included documenting the MIKELANGELO components, so users outside MIKELANGELO can more easily use them.

7 References and Applicable Documents

[1] OSv source code repository, bug tracker, and wiki, https://github.com/cloudius-systems/osv

[2] MIKELANGELO deliverable D2.16, "The First OSv Guest Operating System MIKELANGELO architecture", September 2015, http://www.mikelangelo-project.eu/deliverables/deliverable-d2-16/.

[3] MIKELANGELO deliverable D4.4 "OSv - Guest Operating System - First Version", December 2015, http://www.mikelangelo-project.eu/deliverables/deliverable-d4-4/.

[4] MIKELANGELO deliverable D4.1, "The First Report on I/O Aspects", December 2015, http://www.mikelangelo-project.eu/deliverables/deliverable-d4-1/.

[5] MIKELANGELO deliverable D4.7, "First Version of the Application Packaging", December 2015, http://www.mikelangelo-project.eu/deliverables/deliverable-d4-7/.

[6] MIKELANGELO deliverable D5.7, "First report on the Instrumentation and Monitoring of the complete MIKELANGELO software stack", December 2015, http://www.mikelangelo-project.eu/deliverables/deliverable-d5-7/.

[7] MIKELANGELO deliverable D2.20, "The intermediate MIKELANGELO architecture", June 2016, http://www.mikelangelo-project.eu/deliverables/deliverable-d2-20/.

[8] Seastar source code repository, bug tracker, and wiki, https://github.com/scylladb/seastar

[9] Kivity, Avi, et al., "OSv - optimizing the operating system for virtual machines." 2014 Usenix Annual Technical Conference (Usenix ATC '14). Vol. 1. USENIX Association, 2014.

[10] Kivity, Avi, et al., slides of "OSv - optimizing the operating system for virtual machines." 2014 Usenix Annual Technical Conference (Usenix ATC '14). https://www.usenix.org/sites/default/files/conference/protected-files/atc14_slides_kivity.pdf

[11] ScyllaDB homepage, http://www.scylladb.com/

[12] Capstan, https://github.com/cloudius-systems/capstan

[13] Unik, https://github.com/emc-advanced-dev/unik

[14] Memcached, https://memcached.org/

[15] OSv-apps repository of applications running on OSv, https://github.com/cloudius-systems/osv-apps

[16] Seastar homepage, http://www.seastar-project.org/

[17] OpenFOAM solvers, http://www.openfoam.com/documentation/user-guide/standard-solvers.php

[18] NodeJS, https://nodejs.org

[19] MySQL, https://www.mysql.com

[20] Erlang, https://www.erlang.org

[21] MIKELANGELO deliverable D2.13, "The first sKVM hypervisor architecture", August 2015, http://www.mikelangelo-project.eu/deliverables/deliverable-d2-13/.

[22] J. Pfefferle, P. Stuedi, A. Trivedi, B. Metzler, I. Koltsidas, and T. R. Gross, "A Hybrid I/O Virtualization Framework for RDMA-capable Network Interfaces," in Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, 2015, pp. 17–30.

[23] MIKELANGELO deliverable, D3.2 Intermediate Super KVM - Fast virtual I/O hypervisor, December 2016, http://www.mikelangelo-project.eu/deliverables/deliverable-d3-2/.

[24] MIKELANGELO GitHub Repositories: https://github.com/mikelangelo-project

[25] Unikernel Application Management – User's Manual, https://www.mikelangelo-project.eu/technology/unikernel-application-management-users-manual/

[26] Unikernel Application Management, https://www.mikelangelo-project.eu/technology/unikernel-application-management/

[27] Docker Hub home page, https://hub.docker.com/

[28] Hadoop NameNode and DataNode configuration, http://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html#NameNode_and_DataNodes

[29] Example Node.JS application, https://github.com/amirrajan/word-finder

[30] Docker homepage, https://www.docker.com/

[31] OpenStack Identity service, Keystone, http://docs.openstack.org/developer/keystone/

[32] Hadoop HDFS user guide, https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html

[33] DELL EMC, https://www.emc.com/

[34] Rump kernel, http://rumpkernel.org/

[35] IncludeOS, http://www.includeos.org/

[36] MirageOS, https://mirage.io/

[37] Semantic versioning, http://semver.org/

[38] OpenStack managed unikernel-based OpenFOAM Cloud, https://www.youtube.com/watch?v=UTcc_QRyzPk

8 Appendix A: Asynchronous Programming with Seastar

Section 3 provided a detailed description of the current state of Seastar, the novel API for developing asynchronous server-side application being implemented partially with the support from the MIKELANGELO project. One of the important contributions of the MIKELANGELO project is an improved documentation allowing third party developers get used to the new concepts. The current version (M24 of the MIKELANGELO project) is provided in the following pages.

Asynchronous Programming with Seastar

Nadav Har'El - nyh@ScyllaDB.com Avi Kivity - avi@ScyllaDB.com

Contents

1 Introduction

Seastar, which we introduce in this document, is a C++ library for writing highly efficient and complex server applications on modern multi-core machines.

Traditionally, the programming languages libraries and frameworks used for writing server applications have been divided into two distinct camps: those focusing on efficiency, and those focusing on complexity. Some frameworks are extremely efficient and yet allow building only simple applications (e.g., DPDK allows applications which process packets individually), while other frameworks allow building extremely complex applications, at the cost of run-time efficiency. Seastar is our attempt to get the best of both worlds: To create a library which allows building highly complex server applications, and yet achieve optimal performance.

The inspiration and first use case of Seastar was ScyllaDB, a rewrite of Apache Cassandra. Cassandra is a very complex application, and yet, with Seastar we were able to re-implement it with as much as 10-fold throughput increase, as well as significantly lower and more consistent latencies.

Seastar offers a complete asynchronous programming framework, which uses two concepts - **futures** and **continuations** - to uniformly represent, and handle, every type of asynchronous event, including network I/O, disk I/O, and complex combinations of other events.

Since modern multi-core and multi-socket machines have steep penalties for sharing data between cores (atomic instructions, cache line bouncing and memory fences), Seastar programs use the share-nothing programming model, i.e., the available memory is divided between the cores, each core works on data in its own part of memory, and communication between cores happens via explicit message passing (which itself happens using the SMP's shared memory hardware, of course).
.

1.1 Asynchronous programming

A server for a network protocol, such as the classic HTTP (Web) or SMTP (e-mail) servers, inherently deals with parallelism: Multiple clients send requests in parallel, and we cannot finish handling one request before starting to handle the next: A request may, and often does, need to block because of various reasons — a full TCP window (i.e., a slow connection), disk I/O, or even the client holding on to an inactive connection — and the server needs to handle other connections as well.

The most straightforward way to handle such parallel connections, employed by classic network servers such as Inetd, Apache Httpd and Sendmail, is to use a separate operating-system process per connection. This technique evolved over the years to improve its performance: At first, a new process was spawned to handle each new connection; Later, a pool of existing processes was kept and each new connection was assigned to an unemployed process from the pool; Finally, the processes were replaced by threads. However, the common idea behind all these implementations is that at each moment, each process handles exclusively a single connection. Therefore, the server code is free to use blocking system calls, such as reading or writing to a connection, or reading from disk, and if this process blocks, all is well because we have many additional processes ready to handle other connections.

Programming a server which uses a process (or a thread) per connection is known as *synchronous* programming, because the code is written linearly, and one line of code starts to run after the previous line finished. For example, the code may read a request from a socket, parse the request, and then piecemeal read a file from disk and write it back to the socket. Such code is easy to write, almost like traditional non-parallel programs. In fact, it's even possible to run an external non-parallel program to handle each request — this is for example how Apache HTTPd ran "CGI" programs, the first implementation of dynamic Web-page generation.

> NOTE: although the synchronous server application is written in a linear, non-parallel, fashion, behind the scenes the kernel helps ensure that everything happens in parallel and the machine's resources — CPUs, disk and network — are fully utilized. Beyond the process parallelism (we have multiple processes handling multiple connections in parallel), the kernel may even parallelize the work of one individual connection — for example process an outstanding disk request (e.g., read from a disk file) in parallel with handling the network connection (send buffered-but-yet-unsent data, and buffer newly-received data until the application is ready to read it).

But synchronous, process-per-connection, server programming didn't come without disadvantages and costs. Slowly but surely, server authors realized that starting a new process is slow, context switching is slow, and each process comes with significant overheads — most notably the size of its stack. Server and kernel authors worked hard to mitigate these overheads: They switched from processes to threads, from creating new threads to thread pools, they lowered default stack size of each thread, and increased the virtual memory size to allow more partially-utilized stacks. But still, servers with synchronous designs had unsatisfactory performance, and scaled badly as the number of concurrent connections grew. In 1999, Dan Kigel popularized "the C10K problem", the need of a single server to efficiently handle 10,000 concurrent connections — most of them slow or even inactive.

The solution, which became popular in the following decade, was to abandon the cozy but inefficient synchronous server design, and switch to a new type of server design — the *asynchronous*, or *event-driven*, server. An event-driven server has just one thread, or more accurately, one thread per CPU. This single thread runs a tight loop which, at each iteration, checks, using `poll()` (or the more efficient `epoll`) for new events on many open file descriptors, e.g., sockets. For example, an event can be a socket becoming readable (new data has arrived from the remote end) or becoming writable (we can send more data on this connection). The application handles this event by doing some non-blocking operations, modifying one or more of the file descriptors, and maintaining its knowledge of the *state* of this connection.

However, writers of asynchronous server applications faced, and still face today, two significant challenges:

- **Complexity:** Writing a simple asynchronous server is straightforward. But writing a *complex* asynchronous server is notoriously difficult. The handling of a single connection, instead of being a simple easy-to-read function call, now involves a large number of small callback functions, and a complex state machine to remember which function needs to be called when each event occurs.
- **Non-blocking:** Having just one thread per core is important for the performance of the server application, because context switches are slow. However, if we only have one thread per core, the event-handling functions must *never* block, or the core will remain idle. But some existing programming languages and frameworks leave the server author no choice but to use blocking functions, and therefore multiple threads.
  For example, `Cassandra` was written as an asynchronous server application; But because disk I/O was implemented with `mmap`ed files, which can uncontrollably block the whole thread when accessed, they are forced to run multiple threads per CPU.

Moreover, when the best possible performance is desired, the server application, and its programming framework, has no choice but to also take the following into account:

- **Modern Machines**: Modern machines are very different from those of just 10 years ago. They have many cores and deep memory hierarchies (from L1 caches to NUMA) which reward certain programming practices and penalizes others: Unscalable programming practices (such as taking

locks) can devastate performance on many cores; Shared memory and lock-free synchronization primitives are available (i.e., atomic operations and memory-ordering fences) but are dramatically slower than operations that involve only data in a single core's cache, and also prevent the application from scaling to many cores.

- **Programming Language:** High-level languages such Java, Javascript, and similar "modern" languages are convenient, but each comes with its own set of assumptions which conflict with the requirements listed above. These languages, aiming to be portable, also give the programmer less control over the performance of critical code. For really optimal performance, we need a programming language which gives the programmer full control, zero run-time overheads, and on the other hand — sophisticated compile-time code generation and optimization.

Seastar is a framework for writing asynchronous server applications which aims to solve all four of the above challenges: It is a framework for writing *complex* asynchronous applications involving both network and disk I/O. The framework's fast path is entirely single-threaded (per core), scalable to many cores and minimizes the use of costly sharing of memory between cores. It is a C++14 library, giving the user sophisticated compile-time features and full control over performance, without run-time overhead.

1.2 Seastar

Seastar is an event-driven framework allowing you to write non-blocking, asynchronous code in a relatively straightforward manner (once understood). Its APIs are based on futures. Seastar utilizes the following concepts to achieve extreme performance:

- **Cooperative micro-task scheduler**: instead of running threads, each core runs a cooperative task scheduler. Each task is typically very lightweight – only running for as long as it takes to process the last I/O operation's result and to submit a new one.
- **Share-nothing SMP architecture**: each core runs independently of other cores in an SMP system. Memory, data structures, and CPU time are not shared; instead, inter-core communication uses explicit message passing. A Seastar core is often termed a shard. TODO: more here https://github.com/scylladb/seastar/wiki/SMP
- **Future based APIs**: futures allow you to submit an I/O operation and to chain tasks to be executed on completion of the I/O operation. It is easy to run multiple I/O operations in parallel - for example, in response to a request coming from a TCP connection, you can issue multiple disk I/O requests, send messages to other cores on the same system, or send requests to other nodes in the cluster, wait for some or all of the results to complete, aggregate the results, and send a response.
- **Share-nothing TCP stack**: while Seastar can use the host operating system's TCP stack, it also provides its own high-performance TCP/IP stack built on top of the task scheduler and the share-nothing architecture. The stack provides zero-copy in both directions: you can process data directly from the TCP stack's buffers, and send the contents of your own data structures as part of a message without incurring a copy. Read more...
- **DMA-based storage APIs**: as with the networking stack, Seastar provides zero-copy storage APIs, allowing you to DMA your data to and from your storage devices.

This tutorial is intended for developers already familiar with the C++ language, and will cover how to use Seastar to create a new application.

2 Getting started

The simplest Seastar program is this:

```
#include "core/app-template.hh"
#include "core/reactor.hh"
#include <iostream>
```

```
int main(int argc, char** argv) {
    app_template app;
    app.run(argc, argv, [] {
            std::cout << "Hello world\n";
            return make_ready_future<>();
    });
}
```

As we do in this example, each Seastar program must define and run, an `app_template` object. This object starts the main event loop (the Seastar *engine*) on one or more CPUs, and then runs the given function - in this case an unnamed function, a *lambda* - once.

The "`return make_ready_future<>();`" causes the event loop, and the whole application, to exit immediately after printing the "Hello World" message. In a more typical Seastar application, we will want event loop to remain alive and process incoming packets (for example), until explicitly exited. Such applications will return a *future* which determines when to exit the application. We will introduce futures and how to use them below. In any case, the regular C `exit()` should not be used, because it prevents Seastar or the application from cleaning up appropriately.

To compile this program, first make sure you have downloaded and built Seastar. Below we'll use the symbol `$SEASTAR` to refer to the directory where Seastar was built (Seastar doesn't yet have a "`make install`" feature).

Now, put the above program in a source file anywhere you want, let's call the file `getting-started.cc`. You can compile it with the following command:

```
c++ `pkg-config --cflags --libs $SEASTAR/build/release/seastar.pc` getting-started.cc
```

Linux's pkg-config is a useful tool for easily determining the compilation and linking parameters needed for using various libraries - such as Seastar.

The program now runs as expected:

```
$ ./a.out
Hello world
$
```

3 Threads and memory

3.1 Seastar threads

As explained in the introduction, Seastar-based programs run a single thread on each CPU. Each of these threads runs its own event loop, known as the *engine* in Seastar nomenclature. By default, the Seastar application will take over all the available cores, starting one thread per core. We can see this with the following program, printing `smp::count` which is the number of started threads:

```
#include "core/app-template.hh"
#include "core/reactor.hh"
#include <iostream>

int main(int argc, char** argv) {
    app_template app;
    app.run(argc, argv, [] {
            std::cout << smp::count << "\n";
            return make_ready_future<>();
    });
}
```

On a machine with 4 hardware threads (two cores, and hyperthreading enabled), Seastar will by default start 4 engine threads:

```
$ ./a.out
4
```

Each of these 4 engine threads will be pinned (a la **taskset(1)**) to a different hardware thread. Note how, as we mentioned above, the app's initialization function is run only on one thread, so we see the ouput "4" only once. Later in the tutorial we'll see how to make use of all threads.

The user can pass a command line parameter, `-c`, to tell Seastar to start fewer threads than the available number of hardware threads. For example, to start Seastar on only 2 threads, the user can do:

```
$ ./a.out -c2
2
```

When the machine is configured as in the example above - two cores with two hyperthreads on each - and only two threads are requested, Seastar ensures that each thread is pinned to a different core, and we don't get the two threads competing as hyperthreads of the same core (which would, of course, damage performance).

We cannot start more threads than the number of hardware threads, as allowing this will be grossly inefficient. Trying it will result in an error:

```
$ ./a.out -c5
terminate called after throwing an instance of 'std::runtime_error'
  what():  insufficient processing units
abort (core dumped)
```

The error is an exception thrown from app.run, which we did not catch, leading to this ugly uncaught-exception crash. It is better to catch this sort of startup exceptions, and exit gracefully without a core dump:

```
#include "core/app-template.hh"
#include "core/reactor.hh"
#include "util/log.hh"
#include <iostream>
#include <stdexcept>

int main(int argc, char** argv) {
    app_template app;
    try {
        app.run(argc, argv, [] {
            std::cout << smp::count << "\n";
            return make_ready_future<>();
        });
    } catch(...) {
        std::cerr << "Failed to start application: "
                  << std::current_exception() << "\n";
        return 1;
    }
    return 0;
}
```

```
$ ./a.out -c5
Couldn't start application: std::runtime_error (insufficient processing units)
```

Note that catching the exceptions this way does **not** catch exceptions thrown in the application's actual asynchronous code. We will discuss these later in this tutorial.

3.2 Seastar memory

As explained in the introduction, Seastar applications shard their memory. Each thread is preallocated with a large piece of memory (on the same NUMA node it is running on), and uses only that memory for its allocations (such as `malloc()` or `new`).

By default, the machine's **entire memory** except a small reservation left for the OS (defaulting to 512 MB) is pre-allocated for the application in this manner. This default can be changed by *either* changing the amount reserved for the OS (not used by Seastar) with the `--reserve-memory` option, or by explicitly giving the amount of memory given to the Seastar application, with the `-m` option. This amount of memory can be in bytes, or using the units "k", "M", "G" or "T". These units use the power-of-two values: "M" is a **mebibyte**, 2^20 (=1,048,576) bytes, not a **megabyte** (10^6 or 1,000,000 bytes).

Trying to give Seastar more memory than physical memory immediately fails:

```
$ ./a.out -m10T
Couldn't start application: std::runtime_error (insufficient physical memory)
```

4 Introducing futures and continuations

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

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

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

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

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

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

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

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

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

int main(int argc, char** argv) {
    app_template app;
    app.run(argc, argv, [] {
        std::cout << "Sleeping... " << std::flush;
```

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

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

The return value of `then()` is itself a future which is useful for chaining multiple continuations one after another, as we will explain below. But here we just note that we `return` this future from `app.run`'s function, so that the program will exit only after both the sleep and its continuation are done.

To avoid repeating the boilerplate "app_engine" part in every code example in this tutorial, let's create a simple main() with which we will compile the following examples. This main just calls function `future<> f()`, does the appropriate exception handling, and exits when the future returned by `f` is resolved:

```
#include "core/app-template.hh"
#include "util/log.hh"
#include <iostream>
#include <stdexcept>

extern future<> f();

int main(int argc, char** argv) {
    app_template app;
    try {
        app.run(argc, argv, f);
    } catch(...) {
        std::cerr << "Couldn't start application: "
                  << std::current_exception() << "\n";
        return 1;
    }
    return 0;
}
```

Compiling together with this `main.cc`, the above sleep() example code becomes:

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

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

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

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

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

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

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

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

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

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

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

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

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

4.1 Ready futures

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

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

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

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

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

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

5 Continuations

5.1 Capturing state in continuations

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

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

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

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

The future operation `incr(i)` takes some time to complete (it needs to sleep a bit first...), and in that duration, it needs to save the `i` value it is working on. In the early event-driven programming models, the programmer needed to explicitly define an object for holding this state, and to manage all these objects. Everything is much simpler in Seastar, with C++11's lambdas: The *capture syntax* "`[i]`" in the above example means that the value of i, as it existed when incr() was called() is captured into the lambda. The lambda is not just a function - it is in fact an *object*, with both code and data. In essence,

the compiler created for us automatically the state object, and we neither need to define it, nor to keep track of it (it gets saved together with the continuation, when the continuation is deferred, and gets deleted automatically after the continuation runs).

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

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

Using capture-by-reference in a continuation is usually a mistake, and can lead to serious bugs. For example, if in the above example we captured a reference to i, instead of copying it,

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

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

An exception to the capture-by-reference-is-usually-a-mistake rule is the `do_with()` idiom, which we will introduce later. This idiom ensures that an object lives throughout the life of the continuation, and makes capture-by-reference possible, and very convenient.

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

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

By using unique_ptr in this way, the caller passes an object to the function, but tells it the object is now its exclusive responsibility - and when the function is done with the object, it automatically deletes it. How do we use unique_ptr in a continuation? The following won't work:

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

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

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

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

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

5.2 Handling exceptions

An exception thrown in a continuation is implicitly captured by the system and stored in the future. A future that stores such an exception is similar to a ready future in that it can cause its continuation to be launched, but it does not contain a value – only the exception.

Calling `.then()` on such a future skips over the continuation, and transfers the exception for the input future (the object on which `.then()` is called) to the output future (`.then()`'s return value).

This default handling parallels normal exception behavior – if an exception is thrown in straight-line code, all following lines are skipped:

```
line1();
line2(); // throws!
line3(); // skipped
```

is similar to

```
return line1().then([] {
    return line2(); // throws!
}).then([] {
    return line3(); // skipped
});
```

Usually, aborting the current chain of operations and returning an exception is what's needed, but sometimes more fine-grained control is required. There are several primitives for handling exceptions:

1. `.then_wrapped()`: instead of passing the values carried by the future into the continuation, `.then_wrapped()` passes the input future to the continuation. The future is guaranteed to be in ready state, so the continuation can examine whether it contains a value or an exception, and take appropriate action.
2. `.finally()`: similar to a Java finally block, a `.finally()` continuation is executed whether or not its input future carries an exception or not. The result of the finally continuation is its input future, so `.finally()` can be used to insert code in a flow that is executed unconditionally, but otherwise does not alter the flow.

6 Advanced futures

7 Fibers

Seastar continuations are normally short, but often chained to one another, so that one continuation does a bit of work and then schedules another continuation for later. Such chains can be long, and often even involve loopings - see the following section, "Loops". We call such chains "fibers" of execution.

These fibers are not threads - each is just a string of continuations - but they share some common requirements with traditional threads. For example, we want to avoid one fiber getting starved while a second fiber continuously runs its continuations one after another. As another example, fibers may want to communicate - e.g., one fiber produces data that a second fiber consumes, and we wish to ensure that both fibers get a chance to run, and that if one stops prematurely, the other doesn't hang forever.

8 Semaphores

Seastar's semaphores are the standard computer-science semaphores, adapted for futures. A semaphore is a counter into which you can deposit units or take them away. Taking units from the counter may wait if not enough units are available.

8.1 Limiting parallelism with semaphores

A common use for a semaphore in Seastar is for limiting parallelism, i.e., limiting the number of instances of some code which can run in parallel. This can be important when each of the parallel invocations uses a limited resource (e.g., memory) so letting an unlimited number of them run in parallel can exhaust this resource.

Consider a case where an external source of events (e.g., incoming network requests) causes an asynchronous function `g()` to be called. Imagine that we want to limit the number of concurrent `g()` operations to 100. I.e., If g() is started when 100 other invocations are still ongoing, we want it to delay its real work until one of the other invocations has completed. We can do this with a semaphore:

```
future<> g() {
    static thread_local semaphore limit(100);
    return limit.wait(1).then([] {
        return slow(); // do the real work of g()
    }).finally([] {
        limit.signal(1);
    });
}
```

In this example, the semaphore starts with the counter at 100. The asynchronous operation (`slow()`) is only started when we can reduce the counter by one (`wait(1)`), and when `slow()` is done, either successfully or with exception, the counter is increased back by one (`signal(1)`). This way, when 100 operations have already started their work and have not yet finished, the 101st operation will wait, until one of the ongoing operations finishes and returns a unit to the semaphore. This ensures that at each time we have at most 100 concurrent `slow()` operations running in the above code.

Note how we used a `static thread_local` semaphore, so that all calls to `g()` from the same shard count towards the same limit; As usual, a Seastar application is sharded so this limit is separate per shard (CPU thread). This is usually fine, because sharded applications consider resources to be separate per shard.

Unfortunately, the above example code is actually *incorrect*. Counter-intuitively, `limit.wait(1)` can fail with an exception: when it runs out of memory to keep the list of waiters. In that case, the counter will not be decreased, but the `finally()` clause will still be run, and increase the counter!

To solve this problem, we need the `finally()` to chain to the `slow()` call only, not to `limit.wait(1)`:

```
future<> g() {
    static thread_local semaphore limit(100);
    return limit.wait(1).then([] {
        return slow().finally([] { limit.signal(1); });
    });
}
```

This version also has it's own subtle problem... What if `slow()` throws an exception before returning a future? Note that this is different from `slow()` returning an exceptional future (we discussed this difference in the section about exception handling). In this case, we decreased the counter, but the `finally()` will never be reached, and the counter will never be increased back...

To correctly support the case that `slow()` throws an exception, and also the case where `slow()` is not actually an asynchronous function, but rather a function which returns an non-future value, we can use the `futurize_apply()` function, which converts values and exceptions to the corresponding ready futures:

```
future<> g() {
    static thread_local semaphore limit(100);
    return limit.wait(1).then([] {
        return futurize_apply(slow).finally([] { limit.signal(1); });
    });
}
```

This is finally a bug-free, safe, version.

As we saw now, it is not easy to safely use the separate `semaphore::wait()` and `semaphore::signal()` functions while guaranteeing we never forget to call either one. C++ offers safer mechanisms for acquiring a resource (in this case seamphore units) and later releasing it: lambda functions, and RAII (*resource acquisition is initialization*):

The lambda-based solution is an exception-safe shortcut `with_semaphore()` that replaces exactly the last example above:

```
future<> g() {
    static thread_local semaphore limit(100);
    return with_semaphore(limit, 1, [] {
        return slow(); // do the real work of g()
    });
}
```

`with_semaphore()`, like the code above, waits for the given number of units from the semaphore, then runs the given lambda, and when the future returned by the lambda is resolved, `with_semaphore()` returns back the units to the semaphore. `with_semaphore()` returns a future which only resolves after all these steps are done.

The function `get_units()` provides a safer alternative to `semaphore`'s separate `wait()` and `signal()` functions, based on C++'s RAII philosophy: It returns an opaque units object, which while held, keeps the semaphore's counter decreased - and as soon as this object is destructed, the counter is increased back. With this interface you cannot forget to increase the counter, or increase it twice, or increase without decreasing: The counter will always be decreased once when the units object is created, and if that succeeded, increased when the object is destructed. When the units object is moved into a continuation, no matter how this continuation ends, when the continuation is destructed, the units object is destructed and the units are returned to the semaphore's counter. The above examples, written with `get_units()`, looks like this:

```
future<> g() {
    static thread_local semaphore limit(100);
    return get_units(limit, 1).then([] (auto units) {
        return futurize_apply(slow).finally([units = std::move(units)] {});
    });
}
```

8.2 Limiting resource use

Because semaphores support waiting for any number of units, not just 1, we can use them for more than simple limiting of the *number* of parallel invocation. For example, consider we have an asynchronous function `using_lots_of_memory(size_t bytes)`, which uses `bytes` bytes of memory, and we want to ensure that not more than 1 MB of memory is used by all parallel invocations of this function — and that additional calls are delayed until previous calls have finished. We can do this with a semaphore:

```
future<> using_lots_of_memory(size_t bytes) {
    static thread_local semaphore limit(1000000); // limit to 1MB
    return with_semaphore(limit, bytes, [bytes] {
```

```
        // do something allocating 'bytes' bytes of memory
    });
}
```

Watch out that in the above example, a call to `using_lots_of_memory(2000000)` will return a future that never resolves, because the semaphore will never contain enough units to satisfy the semaphore wait. `using_lots_of_memory()` should probably check whether `bytes` is above the limit, and throw an exception in that case.

8.3 Limiting parallelism of loops

Consider the following simple loop:

```
#include "core/sleep.hh"
future<> slow() {
    std::cerr << ".";
    return sleep(std::chrono::seconds(1));
}
future<> f() {
    return repeat([] {
        return slow().then([] { return stop_iteration::no; });
    });
}
```

This loop runs the `slow()` function (taking one second to complete) without any parallelism — the next `slow()` call starts only when the previous one completed. But what if we do not need to serialize the calls to `slow()`, and want to allow multiple instances of it to be ongoing concurrently?

Naively, we could achieve more parallelism, by starting the next call to `slow()` right after the previous call — ignoring the future returned by the previous call to `slow()` and not waiting for it to resolve:

```
future<> f() {
    return repeat([] {
        slow();
        return stop_iteration::no;
    });
}
```

But in this loop, there is no limit to the amount of parallelism — millions of `sleep()` calls might be active in parallel, before the first one ever returned. Eventually, this loop will consume all available memory and crash.

Using a semaphore allows us to run many instances of `slow()` in parallel, but limit the number of these parallel instances to, in the following example, 100:

```
future<> f() {
    return do_with(semaphore(100), [] (auto& limit) {
        return repeat([&limit] {
            return limit.wait(1).then([&limit] {
                slow().finally([&limit] {
                    limit.signal(1);
                });
                return stop_iteration::no;
            });
        });
    });
}
```

Note how in this code we do not wait for `slow()` to complete before continuing the loop (i.e., we do not `return` the future chain starting at `slow()`); The loop continues to the next iteration when a semaphore unit becomes available, while (in our example) 99 other operations might be ongoing in the background and we do not wait for them.

The above example is incomplete, because it has a never-ending loop and the future returned by `f()` will never resolve. In more realistic cases, the loop has an end, and at the end of the loop we need to wait for all the background operations which the loop started. We can do this by `wait()`ing on the original count of the semaphore: When the full count is finally available, it means that *all* the operations have completed. For example, the following loop ends after 456 iterations:

```
future<> f() {
    return do_with(semaphore(100), [] (auto& limit) {
        return do_for_each(boost::irange(0, 456), [&limit] (int i) {
            return limit.wait(1).then([&limit] {
                slow().finally([&limit] { limit.signal(1); });
            });
        }).finally([&limit] {
            return limit.wait(100);
        });
    });
}
```

The last `finally` is what ensures we wait for the last operations to complete: After the `repeat` loop ends (whether successfully or prematurely because of an exception in one of the iterations), we do a `wait(100)` to wait for the semaphore to reach its original value 100, meaning that all operations that we started have completed. Without this `finally`, the future returned by `f()` will resolve *before* all the iterations of the loop actually completed (the last 100 may still be running).

In the idiom we saw in the above example, the same semaphore is used both for limiting the number of background operations, and later to wait for all of them to complete. Sometimes, we want several different loops to use the same semaphore to limit their total parallelism. In that case we must use a separate mechanism for waiting for the completion of the background operations started by the loop. The most convenient way to wait for ongoing operations is using a gate, which we will describe in detail later. A typical example of a loop whose parallelism is limited by an external semaphore:

```
thread_local semaphore limit(100);
future<> f() {
    return do_with(seastar::gate(), [] (auto& gate) {
        return do_for_each(boost::irange(0, 456), [&gate] (int i) {
            return limit.wait(1).then([&gate] {
                gate.enter();
                slow().finally([&gate] {
                    limit.signal(1);
                    gate.leave();
                });
            });
        }).finally([&gate] {
            return gate.close();
        });
    });
}
```

In this code, we use the external semaphore `limit` to limit the number of concurrent operations, but additionally have a gate specific to this loop to help us wait for all ongoing operations to complete.

Note that in the above examples, we could not use the `with_semaphore()` shortcut. `with_semaphore()` returns a future which only resolves after the lambda's returned future resolves. But in the above examples, the loop needs to know when just the semaphore units are available, to start the next iteration, and not wait for the previous iteration to complete. We could not achieve that with `with_semaphore()`.

9 Pipes

Seastar's `pipe<T>` is a mechanism to transfer data between two fibers, one producing data, and the other consuming it. It has a fixed-size buffer to ensures a balanced execution of the two fibers, because the producer fiber blocks when it writes to a full pipe, until the consumer fiber gets to run and read from the pipe.

A `pipe<T>` resembles a Unix pipe, in that it has a read side, a write side, and a fixed-sized buffer between them, and supports either end to be closed independently (and EOF or broken pipe when using the other side). A `pipe<T>` object holds the reader and write sides of the pipe as two separate objects. These objects can be moved into two different fibers. Importantly, if one of the pipe ends is destroyed (i.e., the continuations capturing it end), the other end of the pipe will stop blocking, so the other fiber will not hang.

The pipe's read and write interfaces are future-based blocking. I.e., the write() and read() methods return a future which is fulfilled when the operation is complete. The pipe is single-reader single-writer, meaning that until the future returned by read() is fulfilled, read() must not be called again (and same for write). Note: The pipe reader and writer are movable, but *not* copyable. It is often convenient to wrap each end in a shared pointer, so it can be copied (e.g., used in an std::function which needs to be copyable) or easily captured into multiple continuations.

10 Shutting down a service with a gate

Consider an application which has some long operation `slow()`, and many such operations may be started at any time. A number of `slow()` operations may even even be active in parallel. Now, you want to shut down this service, but want to make sure that before that, all outstanding operations are completed. Moreover, you don't want to allow new `slow()` operations to start while the shut-down is in progress.

This is the purpose of a `seastar::gate`. A gate `g` maintains an internal counter of operations in progress. We call `g.enter()` when entering an operation (i.e., before running `slow()`), and call `g.leave()` when leaving the operation (when a call to `slow()` completed). The method `g.close()` *closes the gate*, which means it forbids any further calls to `g.enter()` (such attempts will generate an exception); Moreover `g.close()` returns a future which resolves when all the existing operations have completed. In other words, when `g.close()` resolves, we know that no more invocations of `slow()` can be in progress - because the ones that already started have completed, and new ones could not have started.

The construct

```
seastar::with_gate(g, [] { return slow(); })
```

can be used as a shortcut to the idiom

```
g.enter();
slow().finally([&g] { g.leave(); });
```

Here is a typical example of using a gate:

```
#include "core/sleep.hh"
#include "core/gate.hh"
#include <boost/iterator/counting_iterator.hpp>

future<> slow(int i) {
    std::cerr << "starting " << i << "\n";
    return sleep(std::chrono::seconds(10)).then([i] {
        std::cerr << "done " << i << "\n";
    });
}
```

```
future<> f() {
    return do_with(seastar::gate(), [] (auto& g) {
        return do_for_each(boost::counting_iterator<int>(1),
                boost::counting_iterator<int>(6),
                [&g] (int i) {
            seastar::with_gate(g, [i] { return slow(i); });
            // wait one second before starting the next iteration
            return sleep(std::chrono::seconds(1));
        }).then([&g] {
            sleep(std::chrono::seconds(1)).then([&g] {
                // This will fail, because it will be after the close()
                g.enter();
                seastar::with_gate(g, [] { return slow(6); });
            });
            return g.close();
        });
    });
}
```

In this example, we have a function `future<> slow()` taking 10 seconds to complete. We run it in a loop 5 times, waiting 1 second between calls, and surround each call with entering and leaving the gate (using `with_gate`). After the 5th call, while all calls are still ongoing (because each takes 10 seconds to complete), we close the gate and wait for it before exiting the program. We also test that new calls cannot begin after closing the gate, by trying to enter the gate again one second after closing it.

The output of this program looks like this:

```
starting 1
starting 2
starting 3
starting 4
starting 5
WARNING: exceptional future ignored of type 'seastar::gate_closed_exception': gate closed
done 1
done 2
done 3
done 4
done 5
```

Here, the invocations of `slow()` were started at 1 second intervals. After the "`starting 5`" message, we closed the gate and another attempt to use it resulted in a `seastar::gate_closed_exception`, which we ignored and hence this message. At this point the application waits for the future returned by `g.close()`. This will happen once all the `slow()` invocations have completed: Immediately after printing "`done 5`", the test program stops.

As explained so far, a gate can prevent new invocations of an operation, and wait for any in-progress operations to complete. However, these in-progress operations may take a very long time to complete. Often, a long operation would like to know that a shut-down has been requested, so it could stop its work prematurely. An operation can check whether its gate was closed by calling the gate's `check()` method: If the gate is already closed, the `check()` method throws an exception (the same `seastar::gate_closed_exception` that `enter()` would throw at that point). The intent is that the exception will cause the operation calling it to stop at this point.

In the previous example code, we had an un-interruptible operation `slow()` which slept for 10 seconds. Let's replace it by a loop of 10 one-second sleeps, calling `g.check()` each second:

```
future<> slow(int i, seastar::gate &g) {
    std::cerr << "starting " << i << "\n";
    return do_for_each(boost::counting_iterator<int>(0),
                       boost::counting_iterator<int>(10),
            [&g] (int) {
        g.check();
        return sleep(std::chrono::seconds(1));
    }).finally([i] {
        std::cerr << "done " << i << "\n";
    });
}
```

Now, just one second after gate is closed (after the "starting 5" message is printed), all the `slow()` operations notice the gate was closed, and stop. As expected, the exception stops the `do_for_each()` loop, and the `finally()` continuation is performed so we see the "done" messages for all five operations.

11 Introducing Seastar's network stack

We begin with a simple example of a TCP network server written in Seastar. This server repeatedly accepts connections on TCP port 1234, and returns an empty response:

```
#include "core/seastar.hh"
#include "core/reactor.hh"
#include "core/future-util.hh"
#include <iostream>

future<> f() {
    return do_with(listen(make_ipv4_address({1234})), [] (auto& listener) {
        return keep_doing([&listener] () {
            return listener.accept().then(
                [] (connected_socket s, socket_address a) {
                    std::cout << "Accepted connection from " << a << "\n";
                });
        });
    });
}
```

This code works as follows:

1. The `listen()` call creates a `server_socket` object, `listener`, which listens on TCP port 1234 (on any network interface).
2. To handle one connection, we call `listener`'s `accept()` method. This method returns a `future<connected_socket, socket_address>`, i.e., is eventually resolved with an incoming TCP connection from a client (`connected_socket`) and the client's IP address and port (`socket_address`).
3. To repeatedly accept new connections, we use the `keep_doing()` loop idiom. `keep_doing()` runs its lambda parameter over and over, starting the next iteration as soon as the future returned by the previous iteration completes. The iterations only stop if an exception is encountered. The future returned by `keep_doing()` itself completes only when the iteration stops (i.e., only on exception).
4. We use `do_with()` to ensure that the listener socket lives throughout the loop.

Output from this server looks like the following example:

```
$ ./a.out -c1
Accepted connection from 127.0.0.1:47578
Accepted connection from 127.0.0.1:47582
...
```

Note how we ran this Seastar application on a single thread, using the `-c1` option. Unintuitively, this options is actually necessary for running this program, as it will *not* work correctly if started on multiple threads. To understand why, we need to understand how Seastar's network stack works on multiple threads:

For optimum performance, Seastar's network stack is sharded just like Seastar applications are: each shard (thread) takes responsibility for a different subset of the connections. In other words, each incoming connection is directed to one of the threads, and after a connection is established, it continues to be handled on the same thread. But in our example, our server code only runs on the first thread, and the result is that only some of the connections (those which are randomly directed to thread 0) will get serviced properly, and other connections attempts will be ignored.

If you run the above example server immediately after killing the previous server, it often fails to start again, complaining that:

```
$ ./a.out -c1
program failed with uncaught exception: bind: Address already in use
```

This happens because by default, Seastar refuses to reuse the local port if there are any vestiges of old connections using that port. In our silly server, because the server is the side which first closes the connection, each connection lingers for a while in the "`TIME_WAIT`" state after being closed, and these prevent `listen()` on the same port from succeeding. Luckily, we can give listen an option to work despite these remaining `TIME_WAIT`. This option is analogous to `socket(7)`'s `SO_REUSEADDR` option:

```
    listen_options lo;
    lo.reuse_address = true;
    return do_with(listen(make_ipv4_address({1234}), lo), [] (auto& listener) {
```

Most servers will always turn on this `reuse_address` listen option. Stevens' book "Unix Network Programming" even says that "All TCP servers should specify this socket option to allow the server to be restarted". Therefore in the future Seastar should probably default to this option being on — even if for historic reasons this is not the default in Linux's socket API.

Let's advance our example server by outputting some canned response to each connection, instead of closing each connection immediately with an empty reply.

```
#include "core/seastar.hh"
#include "core/reactor.hh"
#include "core/future-util.hh"
#include <iostream>

const char* canned_response = "Seastar is the future!\n";

future<> f() {
    listen_options lo;
    lo.reuse_address = true;
    return do_with(listen(make_ipv4_address({1234}), lo), [] (auto& listener) {
        return keep_doing([&listener] () {
            return listener.accept().then(
                [] (connected_socket s, socket_address a) {
                    auto out = s.output();
                    return do_with(std::move(s), std::move(out),
                        [] (auto& s, auto& out) {
                            return out.write(canned_response).then([&out] {
                                return out.close();
                    });
                });
                });
        });
```

```
    });
}
```

The new part of this code begins by taking the `connected_socket`'s `output()`, which returns an `output_stream<char>` object. On this output stream `out` we can write our response using the `write()` method. The simple-looking `write()` operation is in fact a complex asynchronous operation behind the scenes, possibly causing multiple packets to be sent, retransmitted, etc., as needed. `write()` returns a future saying when it is ok to `write()` again to this output stream; This does not necessarily guarantee that the remote peer received all the data we sent it, but it guarantees that the output stream has enough buffer space to allow another write to begin.

After `write()`ing the response to `out`, the example code calls `out.close()` and waits for the future it returns. This is necessary, because `write()` attempts to batch writes so might not have yet written anything to the TCP stack at this point, and only when close() concludes can we be sure that all the data we wrote to the output stream has actually reached the TCP stack — and only at this point we may finally dispose of the `out` and `s` objects.

Indeed, this server returns the expected response:

```
$ telnet localhost 1234
...
Seastar is the future!
Connection closed by foreign host.
```

In the above example we only saw writing to the socket. Real servers will also want to read from the socket. The `connected_socket`'s `input()` method returns an `input_stream<char>` object which can be used to read from the socket. The simplest way to read from this stream is using the `read()` method which returns a future `temporary_buffer<char>`, containing some more bytes read from the socket — or an empty buffer when the remote end shut down the connection.

`temporary_buffer<char>` is a convenient and safe way to pass around byte buffers that are only needed temporarily (e.g., while processing a request). As soon as this object goes out of scope (by normal return, or exception), the memory it holds gets automatically freed. Ownership of buffer can also be transferred by `std::move()`ing it. We'll discuss `temporary_buffer` in more details in a later section.

Let's look at a simple example server involving both reads an writes. This is a simple echo server, as described in RFC 862: The server listens for connections from the client, and once a connection is established, any data received is simply sent back - until the client closes the connection.

```
#include "core/seastar.hh"
#include "core/reactor.hh"
#include "core/future-util.hh"

future<> handle_connection(connected_socket s, socket_address a) {
    auto out = s.output();
    auto in = s.input();
    return do_with(std::move(s), std::move(out), std::move(in),
        [] (auto& s, auto& out, auto& in) {
            return repeat([&out, &in] {
                return in.read().then([&out] (auto buf) {
                    if (buf) {
                        return out.write(std::move(buf)).then([] {
                            return stop_iteration::no;
                        });
                    } else {
                        return make_ready_future<stop_iteration>(stop_iteration::yes);
                    }
                });
            }).then([&out] {
```

```
                return out.close();
            });
        });
}

future<> f() {
    listen_options lo;
    lo.reuse_address = true;
    return do_with(listen(make_ipv4_address({1234}), lo), [] (auto& listener) {
        return keep_doing([&listener] () {
            return listener.accept().then(
                [] (connected_socket s, socket_address a) {
                    // Note we ignore, not return, the future returned by
                    // handle_connection(), so we do not wait for one
                    // connection to be handled before accepting the next one.
                    handle_connection(std::move(s), std::move(a));
                });
        });
    });
}
```

The main function `f()` loops accepting new connections, and for each connection calls `handle_connection()` to handle this connection. Our `handle_connection()` returns a future saying when handling this connection completed, but importantly, we do ***not*** wait for this future: Remember that `keep_doing` will only start the next iteration when the future returned by the previous iteration is resolved. Because we want to allow parallel ongoing connections, we don't want the next `accept()` to wait until the previously accepted connection was closed. So we call `handle_connection()` to start the handling of the connection, but return nothing from the continuation, which resolves that future immediately, so `keep_doing` will continue to the next `accept()`.

This demonstrates how easy it is to run parallel *fibers* (chains of continuations) in Seastar - When a continuation runs an asynchronous function but ignores the future it returns, the asynchronous operation continues in parallel, but never waited for.

It is often a mistake to silently ignore an exception, so if the future we're ignoring might resolve with an except, it is recommended to handle this case, e.g. using a `handle_exception()` continuation. In our case, a failed connection is fine (e.g., the client might close its connection will we're sending it output), so we did not bother to handle the exception.

The `handle_connection()` function itself is straightforward — it repeatedly calls `read()` read on the input stream, to receive a `temporary_buffer` with some data, and then moves this temporary buffer into a `write()` call on the output stream. The buffer will eventually be freed, automatically, when the `write()` is done with it. When `read()` eventually returns an empty buffer signifying the end of input, we stop `repeat`'s iteration by returning a `stop_iteration::yes`.

12 Command line options

12.1 Standard Seastar command-line options

All Seastar applications accept a standard set of command-line arguments, such as those we've already seen above: The `-c` option for controlling the number of threads used, or `-m` for determining the amount of memory given to the application.

Every Seastar application also accepts the `-h` (or `--help`) option, which lists and explains all the available options — the standard Seastar ones, and the user-defined ones as explained below.

12.2 User-defined command-line options

Seastar parses the command line options (`argv[]`) when it is passed to `app_template::run()`, looking for its own standard options. Therefore, it is not recommended that the application tries to parse `argv[]` on its own because the application might not understand some of the standard Seastar options and not be able to correctly skip them.

Rather, applications which want to have command-line options of their own should tell Seastar's command line parser of these additional application-specific options, and ask Seastar's command line parser to recognize them too. Seastar's command line parser is actually the Boost library's `boost::program_options`. An application adds its own option by using the `add_options()` and `add_positional_options()` methods on the `app_template` to define options, and later calling `configuration()` to retrieve the setting of these options. For example,

```
#include <iostream>
#include <core/app-template.hh>
#include <core/reactor.hh>
int main(int argc, char** argv) {
    app_template app;
    namespace bpo = boost::program_options;
    app.add_options()
        ("flag", "some optional flag")
        ("size,s", bpo::value<int>()->default_value(100), "size")
        ;
    app.add_positional_options({
       { "filename", bpo::value<std::vector<sstring>>()->default_value({}),
         "sstable files to verify", -1}
    });
    app.run(argc, argv, [&app] {
        auto& args = app.configuration();
        if (args.count("flag")) {
            std::cout << "Flag is on\n";
        }
        std::cout << "Size is " << args["size"].as<int>() << "\n";
        auto& filenames = args["filename"].as<std::vector<sstring>>();
        for (auto&& fn : filenames) {
            std::cout << fn << "\n";
        }
        return make_ready_future<>();
    });
    return 0;
}
```

In this example, we add via `add_options()` two application-specific options: `--flag` is an optional parameter which doesn't take any additional agruments, and `--size` (or `-s`) takes an integer value, which defaults (if this option is missing) to 100. Additionally, we ask via `add_positional_options()` that an unlimited number of arguments that do not begin with a "`-`" — the so-called *positional* arguments — be collected to a vector of strings under the "filename" option. Some example outputs from this program:

```
$ ./a.out
Size is 100
$ ./a.out --flag
Flag is on
Size is 100
$ ./a.out --flag -s 3
Flag is on
Size is 3
$ ./a.out --size 3 hello hi
```

```
Size is 3
hello
hi
$ ./a.out --filename hello --size 3 hi
Size is 3
hello
hi
```

`boost::program_options` has more powerful features, such as required options, option checking and combining, various option types, and more. Please refer to Boost's documentation for more information.

13 Debugging a Seastar program

handle SIGUSR1 pass noprint
handle SIGALRM pass noprint

14 Promise objects

As we already defined above, An **asynchronous function**, also called a **promise**, is a function which returns a future and arranges for this future to be eventually resolved. As we already saw, an asynchronous function is usually written in terms of other asynchronous functions, for example we saw the function `slow()` which waits for the existing asynchronous function `sleep()` to complete, and then returns 3:

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

The most basic building block for writing promises is the **promise object**, an object of type `promise<T>`. A `promise<T>` has a method `future<T> get_future()` to returns a future, and a method `set_value(T)`, to resolve this future. An asynchronous function can create a promise object, return its future, and the `set_value` method to be eventually called - which will finally resolve the future it returned.

CONTINUE HERE. write an example, e.g., something which writes a message every second, and after 10 messages, completes the future.

15 Seastar::thread

Seastar has support for threads [http://docs.seastar-project.org/master/group__thread-module.html], but they are not source-compatible with POSIX threads.

In general, seastar futures and threads can coexist. Here is an example using blocking I/O in seastar and converting it back to a future:

```
future<sstring> read_file(sstring file_name) {
    return seastar::async([file_name] () {  // lambda executed in a thread
         file f = open_file_dma(file_name).get0();   // get0() call "blocks"
         auto buf = f.dma_read(0, 512).get()0;  // "block" again
         return sstring(buf.get(), buf.size());
    });
};
```

The `async()` call starts a thread, executes the function, and returns a future that will be resolved with the function's return value, when it completes.

Seastar threads provide an execution environment where blocking is tolerated; you can issue I/O, and wait for it in the same function, rather then establishing a callback to be called with future<>::then().

Seastar threads are not the same as operating system threads:

- seastar threads are cooperative; they are never preempted except at blocking points (see below)
- seastar threads always run on the same core they were launched on

Like other seastar code, seastar threads may not issue blocking system calls.

A seastar thread blocking point is any function that returns a future<>. you block by calling future<>::get(); this waits for the future to become available, and in the meanwhile, other seastar threads and seastar non-threaded code may execute.

Example:

```
seastar::thread th([] {
   sleep(5s).get();  // blocking point
});
```

An easy way to launch a thread and carry out some computation, and return a result from this execution is by using the seastar::async() function. The result is returned as a future, so that non-threaded code can wait for the thread to terminate and yield a result.

9 Appendix B: Capstan User's Manual

Application package management presented in Section 5 promises to greatly simplify management and composition of unikernel applications. The following pages provide detailed user's guide, accompanying the M24 release of the Capstan tool. The documentation consists of an introduction of Capstan and its capabilities, guide to installing it and a step-by-step introduction into getting the first unikernel up and running.

Capstan

This is an upgraded version of the original Capstan and is maintained by MIKELANGELO consortium. Although it is still possible, joining this repository with the original Capstan is not very likely. We're looking towards the first stable release at the moment.

Capstan is a command-line tool for rapidly running your application on OSv unikernel. It focuses on improving user experience during building the unikernel and attemts to support not only a variety of runtimes (C, C++, Java, NodeJS etc.), but also a variety of ready-to-run applications (Hadoop, MySQL, SimpleFOAM etc.).

Philosophy

Buildning unikernels is generally a nightmare! It is a non-trivial task that requires deep knowledge of unikernel implementation. It depends on a lot of installation tools and takes somewhat 10 minutes to complete once configured correctly. But an application-oriented developer is not willing to take a load of new knowledge about unikerel specifics, nor wait long minutes to compile! And that's where Capstan comes in.

Capstan tends to be a tool that one configures with *application-oriented settings* (Where is application entry point? What environment variables to pass? etc.) and then runs a command or two to quickly boot up a new unikernel with her application. Measured in seconds.

To achieve this, Capstan uses **precompiled** stuff: precompiled OSv kernel, precompiled Java runtime, precompiled MySQL, and many more. All you have to do is to name what precompiled packages you want to have available in your unikernel and that's it.

Features

Capstan is designed to prepare and run OSv unikernel for you. With Capstan it is possible to:

- prepare OSv unikernel without compiling anything but your application, in seconds
- use any precompiled package from the MIKELANGELO package repository, or a combination of them
- set arbitrary size of the target unikernel filesystem
- run OSv unikernel using one of the supported providers

But Capstan is not a magic tool that could solve all the problems for you. Capstan does **not**:

- compile your application. If you have Java application, you need to use `javac` compiler and compile the application yourself prior using Capstan tool!
- inspect your application. Capstan does nothing with your application but copies it into the unikernel
- overcome OSv unikernel limits. Consult OSv documentation about what these limits are since they all still apply. Most notably, you can only run single process inside unikernel (forks are forbidden).

Getting started

Capstan can be installed using precompiled binary or compiled from source.
Step-by-step Capstan Installation Guide

Using Capstan is rather simple: open up your project directory, create Capstan configuration files there and, being in project root directory, use Capstan command to create unikernel:

```
$ capstan package compose {unikernel-name}
```

At this point, you have your unikernel built. It contains all your project files plus all the precompiled stuff that you asked for. In other words, the unikernel contains everything and is ready to be started! As you might have expected, there is Capstan command to run unikernel for you:

```
$ capstan run {unikernel-name}
```

Congratulations, your unikernel is up-and-running! Press CTRL + C to stop it.

Documentation

- Step-by-step Capstan Installation Guide
- Running My First Application Inside Unikernel
- Configuration Files
- CLI Reference
- Under the Hood

License

Capstan is distributed under the 3-clause BSD license.

Acknowledgements

This project has been conducted within the RIA MIKELANGELO project (no. 645402), started in January 2015, and co-funded by the European Commission under the H2020-ICT- 07-2014: Advanced Cloud Infrastructures and Services programme.

Step-by-step Installation Guide

You can install Capstan either by downloading pre-built binaries or building it from sources. You need to install QEMU in both cases.

Install Prerequisites

QEMU

Capstan needs QEMU hypervisor being installed on your system, even if you don't intend to run unikernels with QEMU provider. So go ahead and install it:

On Fedora:

```
$ sudo yum install qemu-system-x86 qemu-img
```

On Ubuntu

```
$ sudo apt-get install qemu-system-x86 qemu-utils
```

On OS X:

```
$ brew install qemu
```

On FreeBSD:

```
$ sudo pkg install qemu
```

Install Capstan

Run this script to download Capstan binary into `$HOME/bin` directory:

```
$ ./scripts/download
```

You can then use Capstan tool with `$HOME/bin/capstan --help` or include `$HOME/bin` into `PATH` and use it simply with `capstan --help`.

Install Capstan from source (advanced)

Install Go 1.6

Capstan is a Go project and needs to be compiled first. But heads up, compiling Go project is trivial,

as long as you have Go installed. Consult official documentation to learn how to install Go, or use this bash snippet to do it for you:

```
curl https://storage.googleapis.com/golang/go1.6.2.linux-amd64.tar.gz | sudo tar xz
-C /usr/local
sudo mv /usr/local/go /usr/local/go1.6
sudo ln -s /usr/local/go1.6 /usr/local/go

export GOPATH=$HOME/go
export PATH=$GOPATH/bin:$PATH
export PATH=/usr/local/go1.6/bin:$PATH
```

Compile Capstan

Since Capstan is hosted on GitHub, the compilation process is as simple as:

```
go get github.com/mikelangelo-project/capstan
go install github.com/mikelangelo-project/capstan
```

That's it, we have Capstan installed. The binary resides in $GOPATH/bin, so make sure to add it to your path. You should probably start using Capstan with:

```
capstan --help
```

Configure Capstan (advanced)

Capstan uses optimized default values under the hood. But you are allowed to overwrite them with your own values and this section describes how. Actually, there are three ways to overwrite them (first non-empty value is taken), although not every variable can be set using all three ways:

1) using command-line arguments

You can overwrite some variables using command-line arguments. Please note that you need to repeat the argument for every command you use, Capstan doesn't memorize it. Also please pay attention of the location of the argument. Capstan command must look like this:

```
$ capstan {command-line-configuration} other sub commands and args
# For example:
$ capstan -u https://mikelangelo-capstan.s3.amazonaws.com/ package compose img1
--size 10GB
          |-------------- here --------------------------|
```

List of supported arguments:

- `-u <repo-URL>` overwrites the default remote repository URL that is used to fetch precompiled

packages from.

2) using configuration file

Capstan supports configuration file to permanently overwrite some internal defaults. This file is located in `$HOME/.capstan/config.yaml`. It is not created by default, so you need to create the file and folder if they do not exist yet. The file is nothing but a simple yaml containing "key: value" pairs e.g.

```
repo_url: https://mikelangelo-capstan.s3.amazonaws.com/
disable_kvm: false
```

List of supported keys:

- `repo_url` overwrites the default remote repository URL that is used to fetch precompiled packages from.
- `disable_kvm` by default KVM acceleration is turned on to speed up unikernel creation, but in certain circumstances this results in error. Set this to `true` if you have problems using KVM.

Please note that if command line argument is used to overwrite the same value (e.g. -u for repository URL), then the value from configuration file is ignored.

3) using environment variables

Capstan reads execution environment to overwrite internal variables. Just set environment variable prior calling Capstan commands, for example:

```
$ export CAPSTAN_REPO_URL=https://mikelangelo-capstan.s3.amazonaws.com/
$ capstan package compose ...
```

List of supported environment variables:

- `CAPSTAN_REPO_URL` overwrites the default remote repository URL that is used to fetch precompiled packages from.

Please note that environment variables have the lowest priority - if same variable is set using either command-line argument either configuration file, then environment variable is ignored.

Double-check your configuration

There is a Capstan command to double-check which configuration values are eventually used:

```
capstan config print
```

Capstan Configuration Files

You must provide two configuration files to run your application using Capstan:

```
{application-root}
└── meta
       |- package.yaml
       |- run.yaml
| ... (application files and directories)
```

Both files must be placed inside directory named `meta/` that you create directly inside your project's root directory. The package.yaml file tells Capstan how to properly *create* unikernel i.e. what precompiled packages to put in it besides your application files. And the run.yaml file gives Capstan all the neccessary information about your application (the runtime, the main file, ...) to *run* it.

meta/package.yaml

Below please find sample content of meta/package.yaml configuration file:

```
name: my-super-application
title: DEMO App
author: lemmy (lemmy@email.com)
require:
    - eu.mikelangelo-project.osv.cli
```

The first three parameters are simple metadata and are used later if you decide to publish your application as a package to make it available for everyone. Be careful, though, that you pick unique name for your package to avoid collisions with other users.

The most interesting attribute is the one named `require`. This is where you list all the packages that you would like to have them included in your final unikernel. In this example we've listed only one, `eu.mikelangelo-project.osv.cli`. You can omit the whole attribute if you don't need any precompiled packages. A list of all packages in the remote repository can be obtained by executing:

```
$ capstan package search
Name                                            Description
Version                   Created
eu.mikelangelo-project.app.hadoop-hdfs          Hadoop HDFS
2.7.2                     0001-01-01T00:00:00Z
eu.mikelangelo-project.app.hello-node           NodeJS-4.4.5
4.4.5                     0001-01-01T00:00:00Z
eu.mikelangelo-project.app.mysql-5.6.21         MySQL-5.6.21
5.6.21                    0001-01-01T00:00:00Z
eu.mikelangelo-project.app.node-4.4.5           NodeJS-4.4.5
4.4.5                     0001-01-01T00:00:00Z
eu.mikelangelo-project.erlang                   Erlang
```

```
18.0                      0001-01-01T00:00:00Z
eu.mikelangelo-project.ompi                         Open MPI
1.10                      0001-01-01T00:00:00Z
eu.mikelangelo-project.openfoam.core                OpenFOAM Core
2.4.0                     0001-01-01T00:00:00Z
eu.mikelangelo-project.openfoam.pimplefoam          OpenFOAM pimpleFoam
2.4.0                     0001-01-01T00:00:00Z
eu.mikelangelo-project.openfoam.pisofoam            OpenFOAM pisoFoam
2.4.0                     0001-01-01T00:00:00Z
eu.mikelangelo-project.openfoam.poroussimplefoam    OpenFOAM porousSimpleFoam
2.4.0                     0001-01-01T00:00:00Z
eu.mikelangelo-project.openfoam.potentialfoam       OpenFOAM potentialFoam
2.4.0                     0001-01-01T00:00:00Z
eu.mikelangelo-project.openfoam.rhoporoussimplefoam OpenFOAM rhoPorousSimpleFoam
2.4.0                     0001-01-01T00:00:00Z
eu.mikelangelo-project.openfoam.rhosimplefoam       OpenFOAM rhoSimpleFoam
2.4.0                     0001-01-01T00:00:00Z
eu.mikelangelo-project.openfoam.simplefoam          OpenFOAM simpleFoam
2.4.0                     0001-01-01T00:00:00Z
eu.mikelangelo-project.osv.bootstrap                OSv Bootstrap
v0.24-216-g1cf8972        0001-01-01T00:00:00Z
eu.mikelangelo-project.osv.cli                      OSv Command Line Interface
v0.24-216-g1cf8972        0001-01-01T00:00:00Z
eu.mikelangelo-project.osv.cloud-init               cloud-init
v0.24-216-g1cf8972        0001-01-01T00:00:00Z
eu.mikelangelo-project.osv.httpserver               OSv HTTP REST Server
v0.24-216-g1cf8972        0001-01-01T00:00:00Z
eu.mikelangelo-project.osv.java                     Java JRE 1.7.0
v0.24-216-g1cf8972        0001-01-01T00:00:00Z
eu.mikelangelo-project.osv.nfs                      OSv NFS Client Tools
v0.24-216-g1cf8972        0001-01-01T00:00:00Z
```

Then to download desired package into your local repository execute:

```
$ capstan package pull {package-name}
```

Alternatively, you can make use of `--pull-missing` flag when composing unikernel.

Please note that packages are copied to the unikernel in the same order that you specify here in meta/package.yaml file. So if two packages contain file with same name and inside same folder path, then the one that was copied last will remain. Only after all the packages are copied to the unikernel, your application files are copied too. So your application can never get overwritten. To verify the final content one can execute:

```
$ capstan package collect
```

A folder `mpm-pkg` appears containing exact content as it will be baked into unikernel during compose.

meta/run.yaml

Content of run.yaml file depends on runtime that this package is about to use. File is structured as follows:

```
runtime: node

# runtime-specific configuration
...
```

Key `runtime` is always required - it defines what runtime we will be using to run our application. A list of all runtimes can be obtained by executing:

```
$ capstan runtime list

RUNTIME             DESCRIPTION                                         DEPENDENCIES
native              Run arbitrary command inside OSv                    []
node                Run JavaScript NodeJS 4.4.5 application
[eu.mikelangelo-project.app.node-4.4.5]
java                Run Java 1.7.0 application
[eu.mikelangelo-project.osv.java]
```

And then Capstan can tell us what settings are supported for each runtime. For example, for NodeJS one can view expected content of the run.yaml file by executing:

```
$ capstan runtime preview -r node

--------- meta/run.yaml ---------
runtime: node

# REQUIRED
# Filepath of the NodeJS entrypoint (where server is defined).
# Note that package root will correspond to filesystem root (/) in OSv image.
# Example value: /server.js
main: <filepath>
---------------------------------
```

Lets sum up. To prepare appropriate run.yaml file, we must first select one of the supported runtimes. If we are about to be running NodeJS application, then we opt-in to use runtime named *node*. We get the details on how to prepare run.yaml for *node* by using Capstan command.

Note that run.yaml also supports named configurations. You can read more about this feature here: Advanced Usage of meta/run.yaml Configuration File

Automatic generation of configuration files

You can create configuration files manually or generate them using Capstan. The latter option does

not only create empty files; Capstan pre-fills them with default values and detailed self-description in form of yaml comments. You are therefore advised to use Capstan to initialize configuration files for you.

To initialize meta/package.yaml file, use:

```
$ capstan package init \
   --name "my-super-application" \
   --title "DEMO App" \
   --author "lemmy (lemmy@email.com)" \
   --require eu.mikelangelo-project.osv.cli
```

This will create a meta subdirectory and `meta/package.yaml` file with the given content. Then to initialize meta/run.yaml file, use:

```
$ capstan runtime init -r {runtime-name}
```

This will create `meta/run.yaml` file with documentation for the selected runtime.

Running My First Application Inside Unikernel

Lets walk through all the steps needed to run your NodeJS application inside OSv unikernel using Capstan. This step-by-step tutorial assumes that you have Capstan already installed, but you know absolutely nothing about how to use it.

OK, suppose I'm a web developer and I've developed some awesome NodeJS server. I ran it hundreds of times on my local machine during the development, but now it's time to go live! Typically I would ask our administrator guy to launch another Ubuntu VM on our OpenStack and I would provision it through ssh. But I've heard a lot about unikernels that are consuming less resources and everything, so I want to give it a try and deploy my NodeJS application in unikernel.

Summary for the impatient

Checkout example application, navigate to it's root direcotry and create Capstan configuration files there:

```
$ git clone https://github.com/amirrajan/word-finder.git word-finder
$ cd word-finder
$ capstan package init --name com.example.word-finder --title "Word Finder"
--author "Lemmy"
$ touch meta/run.yaml && cat > meta/run.yaml <<EOL
runtime: node
config_set:
   word_count:
      main: /start.js
      env:
         PORT: 4000
config_set_default: word_count
EOL
```

Then compose and run your unikernel:

```
capstan package compose com.example.word-finder
capstan run compose com.example.word-finder -f 4004:4000
```

Open up your browser, navigate to `http://localhost:4004` and start using the application that runs in unikernel.

STEP 0: Develop your application locally

Suppose we've developed this NodeJS application and put it on the GitHub. Let's clone it:

```
$ git clone https://github.com/amirrajan/word-finder.git word-finder
$ # PROJECT_ROOT points to our NodeJS project root
$ PROJ_ROOT="$(pwd)/word-finder"
```

and run it locally:

```
$ cd $PROJECT_ROOT
$ npm install
$ node start.js
Listening on port: 3000
```

We can then open browser and navigate to http://localhost:3000 and see that the web server works. Great! But wait, that's running on our development machine, not inside unikernel. Use CTRL + C to stop the server.

Just out of curiosity, lets count number of files in this NodeJS project:

```
$ find . -type f | wc -l
46104
```

Wait what? 46k? Yes. This project is rather simple, but it uses some fancy libraries that contain a lot of files. Just pointing out here that our NodeJS server is not a small one - and yet it will run in the unikernel. Excited yet?

STEP 1: Add meta information for Capstan

Capstan will seek for some configuration files when you ask it to prepare a unikernel. More precisely, there are two configuration files `meta/package.yaml` and `meta/run.yaml` and you need to prepare them for each unikernel. Think of them as a recipe that Capstan needs in order to know what unikernel will suit your needs best.

a) prepare meta/package.yaml

First things first. We need to give some meaningful name to our future unikernel. Navigate to your project diretory and use Capstan utility command to generate this file for you:

```
$ cd $PROJECT_ROOT
$ capstan package init --name com.example.word-finder --title "Word Finder"
--author "Lemmy"
Initializing package in /home/miha/git-repos/word-finder/meta
```

There you go, a folder named `meta` was generated inside project directory containing a file named `package.yaml`. That's where information regarding unikernel content is now stored. You needn't modify anything in this file. Yet, you can take a look at what it contains:

```
$ cd $PROJECT_ROOT
$ cat meta/package.yaml
```

```
name: com.example.word-finder
title: Word Finder
author: Lemmy
```

b) prepare meta/run.yaml

Great! The unikernel has a name. Now we need to somehow inform Capstan that those 46k files are written for NodeJS runtime. We do this by running:

```
$ cd $PROJECT_ROOT
$ capstan runtime init --runtime node
meta/run.yaml stub successfully added to your package. Please customize it in
editor.
```

As the last line suggests, a file `meta/run.yaml` was created with some stubs and now has to be edited manually. Go ahead open it up. It looks like this:

```
$ cd $PROJECT_ROOT
$ cat meta/run.yaml
runtime: node
config_set:
   ###################################################################
   ### This is one configuration set (feel free to rename it).  ###
   ###################################################################
   myconfig1:
      # REQUIRED
      # Filepath of the NodeJS entrypoint (where server is defined).
      # Note that package root will correspond to filesystem root (/) in OSv image.
      # Example value: /server.js
      main: <filepath>

      # OPTIONAL
      # Environment variables.
      # A map of environment variables to be set when unikernel is run.
      # Example value:  env:
      #                    PORT: 8000
      #                    HOSTNAME: www.myserver.org
      env:
         <key>: <value>

   # Add as many named configurations as you need

# OPTIONAL
# What config_set should be used as default.
# This value can be overwritten with --runconfig argument.
config_set_default: myconfig1
```

What we see is a bunch of comments that try to explain what input is needed from you. Don't get scared, you only need to type a few words in here. Final content of the file that is needed to run our NodeJS application (all comments were removed for clarity) is:

```
# meta/run.yaml
runtime: node
config_set:
   word_count:
      main: /start.js
      env:
         PORT: 4000
config_set_default: word_count
```

This reads as follows: *Our application needs to be run using NodeJS runtime with entrypont file /start.js*. Note that we could specify multiple configuration sets here (each e.g. with different entrypoint file or different environment variables) and could then easily switch between them just before unikernel is run. But in this case we only provide one config_set and name it *word_count*.

That's it! Capstan has all the information needed and we can go and create the unikernel at last.

STEP 3: Compose unikernel

Once our NodeJS application has been equipped with appropriate configuration files, Capstan is able to create the unikernel. Execute:

```
$ cd $PROJECT_ROOT
$ capstan package compose com.example.word-finder --size 200MB
(1)  Resolved runtime into: node
(2)  Using named configuration: 'word_count'
(3)  Prepending 'node' runtime dependencies to dep list:
[eu.mikelangelo-project.app.node-4.4.5]
(4)  Importing com.example.word-finder...
(5)  Importing into
...capstan/repository/com.example.word-finder/com.example.word-finder.qemu
(6)  Uploading files to
...capstan/repository/com.example.word-finder/com.example.word-finder.qemu...
(7)  Setting cmdline: --norandom --nomount --noinit /tools/mkfs.so; /tools/cpiod.so
--prefix /zfs/zfs; /zfs.so set compression=off osv
(8)  Uploading files 49319 / 49319 [==============================================]
100.00 % 1m9s
(9)  All files uploaded
(10) Command line set to: --env=PORT=4000 node /start.js
```

Let's observe what the output says. First it confirms that runtime "node" was detected (what a surprise;) and that named configuration named "word_count" is being used (what a surprise - just like we've specified in meta/run.yaml file).

The output then in line (3) says that node runtime dependencies were prepended to the dependencies list. Luckily, Capstan is smart enough to figure out that if we've told him that the 46k files are written for NodeJS runtime, then precompiled package containing NodeJS runtime is required.

Line (5) tells where the resulting unikernel will be created. Wait what, `.qemu` extension? Yup, the

unikernel that we get is nothing but qemu image (oficially the format is called QCOW2). Which is really great since you can just grab it as any other virtual machine image and run it on hipervisor. That's one of the best Capstan features: it produces a ready-to-run VM image.

Then in line (8) you see a progress bar. Capstan has to upload all your project files and required packages into the target unikernel and here you see how it's doing. Note that we're uploading nearly 50k files so it takes around a minute to complete. If you later make a small change to your application code you needn't upload 50k files again, but rather update the existing unikernel by adding --update flag.

Finally, in line (10), Capstan tells you what command will unikernel be booted with by default*. This command is calculated based on the content of your meta/run.yaml. To be honest, this is the *only* command that will ever be executed in unikernel. Aren't unikernels simple?

* *It is possible to change the boot command even for the composed unikernel. See documentation for more details.*

STEP 4: Run unikernel

Once we have unikernel composed, we can run it. There's a bunch of possibilities here - you needn't use Capstan tool for this. You can upload the `.qemu` to the OpenStack and run it there thru Horizon dashboard. Or you can use local installation of qemu and run it. Let's pick the third option, Capstan utility function:

```
$ capstan run com.example.word-finder -f 4004:4000
(1) Resolved runtime into: node
(2) Using named configuration: 'word_count'
(3) Created instance: com.example.word-finder
(4) Setting cmdline: --env=PORT=4000 node /start.js
(5) OSv v0.24-116-g73b38d8
(6) eth0: 192.168.122.15
(7) Listening on port: 4000
```

We passed port forwarding rule `-f 4004:4000` to make unikernel's port 4000 accessible from our localhost:4004. Go ahead, open your browser and navigate to `http://localhost:4004`. There it is, our NodeJS application, running inside OSv unikernel!

Line (1) and (2) inform you what runtime is set and what run configuration is used. As we've mentioned earlier, it is possible to change the boot command even after the unikernel is composed already. Capstan therefore evaluates your meta/run.yaml once again here and makes sure that the correct boot command is set.

Line (3) informs you about your instances's name and line (4) shows what boot command was set. You needn't care about boot command, but the information is printed anyway to feed your curiosity.

All that's prineted after line (4) is captured from unikernel's stdin and stderr. OSv unikernel always prints its kernel version (line 5) and network configuration (line 6) followed by whatever your application is printing to the stdin/stderr. Line (7) was, for example, produced by our NodeJS application. Will it print anything more while the unikernel is running, the text will appear here.

Congratulations! You've composed and run your first OSv unikernel.

CLI Reference

Here we describe Capstan CLI in detail. Please note that this very same information can be obtained by adding --help flag to any of the listed commands.

Packaging Application

These commands are useful when packaging my application into Capstan package.

capstan package init

```
NAME:
   capstan package init - initialise package structure

USAGE:
   capstan package init [command options] [path]

OPTIONS:
   --name value, -n value     package name
   --title value, -t value    package title
   --author value, -a value   package author
   --version value, -v value  package version
   --require value            specify package dependency
   --runtime value            runtime to stub package for. Use 'capstan runtime
list' to list all
```

capstan package collect

```
NAME:
   capstan package collect - collects contents of this package and all required
packages

USAGE:
   capstan package collect [command options] [arguments...]

OPTIONS:
   --pull-missing, -p           attempt to pull packages missing from a local
repository
   --runconfig value, -r value  specify config_set name (specified in meta/run.yaml)
```

capstan package compose

```
NAME:
   capstan package compose - composes the package and all its dependencies into OSv
image
```

```
USAGE:
   capstan package compose [command options] image-name

OPTIONS:
   --size value, -s value       total size of the target image (use M or G suffix)
(default: "10G")
   --update                     updates the existing target VM by uploading only
modified files
   --verbose, -v                verbose mode
   --run value                  the command line to be executed in the VM
   --pull-missing, -p           attempt to pull packages missing from a local
repository
   --runconfig value, -r value  specify config_set name (specified in meta/run.yaml)
```

Dependent Packages

These commands are useful when we intend to use package from remote repository.

capstan package list

```
NAME:
   capstan package list - lists the available packages

USAGE:
   capstan package list [arguments...]
```

capstan package search

```
NAME:
   capstan package search - searches for packages in the remote repository (partial
name matches are also supported)

USAGE:
   capstan package search [package-name]
```

capstan package pull

```
NAME:
   capstan package pull - pulls the package from remote repository and imports it
into local package storage

USAGE:
   capstan package pull [package-name]
```

Runtime

Runtime-related commands.

capstan runtime list

```
NAME:
   capstan runtime list - list available runtimes

USAGE:
   capstan runtime list [arguments...]
```

capstan runtime preview

```
NAME:
   capstan runtime preview - prints runtime yaml template to the console

USAGE:
   capstan runtime preview [command options] [arguments...]

OPTIONS:
   --runtime value, -r value  Runtime name. Use 'capstan runtime list' to see
available names.
   --plain                    Remove comments
```

capstan runtime init

```
NAME:
   capstan runtime init - prepares meta/run.yaml stub for selected runtime

USAGE:
   capstan runtime init [command options] [arguments...]

OPTIONS:
   --runtime value, -r value  Runtime name. Use 'capstan runtime list' to see
available names.
   --named                    Use named configurations format
   --plain                    Remove comments
   --force, -f                Override existing meta/run.yaml
```

Run

Commands used to run composed package.

capstan run

```
NAME:
   capstan run - launch a VM. You may pass the image name as the first argument.

USAGE:
   capstan run [command options] instance-name

OPTIONS:
   -i value                     image_name
   -p value                     hypervisor: qemu|vbox|vmw|gce (default: "qemu")
   -m value                     memory size (default: "1G")
   -c value                     number of CPUs (default: 2)
   -n value                     networking: nat|bridge|tap (default: "nat")
   -v                           verbose mode
   -b value                     networking device (bridge or tap): e.g., virbr0,
vboxnet0, tap0
   -f value                     port forwarding rules
   --gce-upload-dir value       Directory to upload local image to: e.g.,
gs://osvimg
   --mac value                  MAC address. If not specified, the MAC address will
be generated automatically.
   --execute value, -e value    set the command line to execute
   --runconfig value, -r value  specify config_set name (specified in meta/run.yaml)
```

OpenStack Integration

Commands used to compose unikernel, upload it to OpenStack Glance and run it with OpenStack Nova.

capstan stack push

```
NAME:
   capstan stack push - composes OSv image and pushes it to OpenStack

USAGE:
   capstan stack push [command options] image-name

DESCRIPTION:
   Compose package, build .qcow2 image and upload it to OpenStack under nickname
<image-name>.

OPTIONS:
   --size value, -s value       minimal size of the target user partition (use M or
G suffix).
                                NOTE: will be enlarged to match flavor size.
(default: "10G")
   --flavor value, -f value     OpenStack flavor name that created OSv image should
fit to
   --run value                  the command line to be executed in the VM
   --keep-image                 don't delete local composed image in
```

```
.capstan/repository/stack
   --verbose, -v                verbose mode
   --pull-missing, -p           attempt to pull packages missing from a local
repository
   --runconfig value, -r value  specify config_set name (specified in meta/run.yaml)
   --OS_AUTH_URL value          OpenStack auth url (e.g. http://10.0.2.15:5000/v2.0)
   --OS_TENANT_ID value         OpenStack tenant id (e.g.
3dfe7bf545ff4885a3912a92a4a5f8e0)
   --OS_TENANT_NAME value       OpenStack tenant name (e.g. admin)
   --OS_PROJECT_NAME value      OpenStack project name (e.g. admin)
   --OS_USERNAME value          OpenStack username (e.g. admin)
   --OS_PASSWORD value          OpenStack password (*TODO*: leave blank to be
prompted)
   --OS_REGION_NAME value       OpenStack username (e.g. RegionOne)
```

capstan stack run

```
NAME:
   capstan stack run - runs image that was previously pushed to OpenStack

USAGE:
   capstan stack run [command options] image-name

DESCRIPTION:
   Run image that you've previously uploaded with 'capstan stack push'.
   Please note that image size CANNOT be changed at this point (wont' boot on
   too small flavor, wont use extra space on too big flavor), but feel free
   to adjust amount of memory (RAM).

OPTIONS:
   --flavor value, -f value  OpenStack flavor to be run with
   --mem value, -m value     MB of memory (RAM) to be run with
   --name value, -n value    instance name
   --count value, -c value   number of instances to run (default: 1)
   --verbose, -v             verbose mode
   --OS_AUTH_URL value       OpenStack auth url (e.g. http://10.0.2.15:5000/v2.0)
   --OS_TENANT_ID value      OpenStack tenant id (e.g.
3dfe7bf545ff4885a3912a92a4a5f8e0)
   --OS_TENANT_NAME value    OpenStack tenant name (e.g. admin)
   --OS_PROJECT_NAME value   OpenStack project name (e.g. admin)
   --OS_USERNAME value       OpenStack username (e.g. admin)
   --OS_PASSWORD value       OpenStack password (*TODO*: leave blank to be prompted)
   --OS_REGION_NAME value    OpenStack username (e.g. RegionOne)
```

Configuration

Commands used to configure Capstan.

capstan config print

```
NAME:
   capstan config print - print current capstan configuration

USAGE:
   capstan config print [arguments...]
```