UNICORE

H2020-ICT-2018-2-825377

UNICORE

UNICORE: A Common Code Base and Toolkit for Deployment of Applications to Secure and Reliable Virtual Execution Environments

Horizon 2020 - Research and Innovation Framework Programme

D2.1 Requirements

Due date of deliverable: 30 September 2019

Actual submission date: 30 September 2019

Start date of project	1 January 2019
Duration	36 months
Lead contractor for this deliverable	Accelleran NV
Version	1.0
Confidentiality status	"Public"

Abstract

This is the milestone 3 version of the UNICORE Requirements document.

The goal of the EU-funded UNICORE project is to develop a common code-base and toolchain that will enable software developers to rapidly create secure, portable, scalable, high-performance solutions starting from existing applications. The key to this is to compile an application into very light-weight virtual machines - known as unikernels - where there is no traditional operating system, only the specific bits of operating system functionality that the application needs. The resulting unikernels can then be deployed and run on standard high-volume servers or cloud computing infrastructure.

The technology developed by the project will be evaluated in a number of trials, spanning several application domains. This document describes the current state of the art in those application domains from the perspective of the project partners whose businesses encompass those domains. It then goes on to describe the specific target scenarios that will be used to evaluate the technology within each application domain, and how the success of each trial will be judged. Together, these descriptions give an early indication of the requirements for the UNICORE common code-base and toolchain.

This document then details the technical requirements for the Unikernel core (the common code-base), the technical requirements for the UNICORE toolchain and finishes with a section on the conclusions that can be drawn.

Target Audience

The target audience for this document is all project participants.

Impressum

Full project title	UNICORE: A Common Code Base and Toolkit for Deployment of Applications to Secure and Reliable Virtual Execution Environments
Title of the workpackage	D2.1 Requirements
Editor	Accelleran NV
Project Co-ordinator	Emil Slusanschi, UPB
Technical Manager	Felipe Huici, NEC
Copyright notice	© 2019 Participants in project UNICORE

Executive Summary

This is the milestone 3 version of the UNICORE D2.1 document, ”Requirements”.

The document focuses on the requirements arising from the application domains that are of business and academic interest to the various project partners. The approach taken is, for each appliction domain, to first describe how software in those domains is currently deployed, to identify the perceived benefits of using Unikernels and to to identify the high level requirements that would need to be met in order to deploy them using unikernels. Additionally, the performance expectations against which each deployment will be measured are detailed and a description of how unikernels are to be trialled is provided.

In chapter 3, Four application domains are described: Serverless Computing; Network Function Virtualisation; Home Automation and Internet of Things; and Smart Contracts.

In the Serverless Computing domain, CSUC describe the software used for institutional digital content repositories. This is currently deployed using containers and virtual machines, using several existing technologies: Docker; Rancher; Kubernetes; and Open Nebula. The expectation is that the UNICORE technology will significantly reduce the resources needed for certain processing functions, especially image and video media conversion. These are implemented using ImageMagick and ffmpeg respectively, so a unikernel-based deployment will have to support those utilities.

Several possible target deployment scenarios are proposed. The first involves developing a driver for OpenNebula to allow it to orchestrate unikernels instead of virtual machines, on the KVM hypervisor. The second is similar, but with Rancher being the orchestrator and using either KVM or Kubevirt as the hypervisor. The third is to investigate whether it would be possible to integrate unikernels with a Function-as-a-Service platform, such as OpenFaaS. In all cases, the evaluation will include checking whether unikernels give the following benefits: lower deployment time; greater number of concurrent running instances; reduced time to complete a task; and lower resource consumption.

Also in the Serverless Computing domain, CNW desribe the use of lambda-like services for packet processing, where customers will pay ”per-packet” rather than paying for the availability of infrastructure. Clients will implement their network functions by deploying extended BPF (Berkeley Packet Filter) code that will run per-packet. For this to be viable, efficient resource utilisation (both under load and when idle) and strong isolation and security guarantees are essential. It is anticipated that the UNICORE Decompostion Tool, Dependency Analysis Tool, Automatic Build Tool and Verification Tool will be key to achieving these characteristics in a unikernel-based solution.

CNW propose to deploy PacketCloud using two flavours of virtual machine, both running on the KVM hypervisor together with the Firecracker virtual machine monitor. The first runs a pre-compiled network function in a purpose-built unikernel. The second runs a slimmed down version of Linux, preconfigured with the enhanced BPF calls corresponding to the network function. They also propose to develop a solution for managing life cycle events of such PacketCloud virtual machines, that can be either run stand-alone, or in

conjunction with an orchestrator such as Kubernetes.

In the Network Function Virtualisation domain, Orange describe their current Broadband Network Gateway (BNG) solution, which is based on Nokia 7750 hardware, and their goal to replace the physical BNGs with virtualised BNGs based on unikernels. Key benefits of this are expected to be: a great reduction in the time taken to deploy a new service; reduced resource requirements; improved scalability; and improved security.

Orange describe a target deployment scenario in which a monolithic BNG is decomposed so that there is one unikernel per customer, running on the KVM hypervisor in an OpenStack or Docker-based environment. Orchestration (including service orchestration) will be provided using open source tools such as Open Source MANO, OpenStack's Heat or ONAP.

Also in the Network Function Virtualisation domain, Accelleran describe the move to virtualised Radio Access Network (vRAN) functionality for 5G networks. Accelleran's existing RAN products are based on specialised hardware, but they have started migrating some of their functionality to a virtualised environment using Docker containers, orchestrated using Kubernetes: this solution is known as dRAX(TM). The aim is to move this from its current Technology Readiness Level of 3/4 to 7/8 using the UNICORE technology. In order to do this, unikernel-hosted applications must be able to use the Stream Control Transmission Protocol (SCTP) that is available as the sctp kernel module in linux. The existing implementation depends on the following key third-party components, which should also be hostable using unikernels: the NATS messaging system; the Redis distributed data store; and Google's Protocol Buffers. In addition, unikernel support for the following libraries will be required: libc; zlog; sqlite3; openssl; libcurl; and libprotobuf. Benefits in terms of scalability, reliability and security are expected.

Accelleran's target deployment scenario is a lab-based, self-contained 4G/5G mobile network in which at least the dRAX control plane component will be implemented using the UNICORE technology. Initially, other components within the dRAX solution will be deployed in Docker containers, but it is expected that these could also be migrated to unikernels as the project progresses. The dRAX components will run on a relatively low cost platform e.g. Intel i7-based hardware, with container orchestration provided by Kubernetes. The performance of the control plane component will be evaluated by artificially generating high levels of control plane signalling.

Continuing in the Network Function Virtualisation domain, Ekinops describe how they have migrated their existing ONEOS Multi-Service Access Routers (MSAR) and Ethernet Access Devices (EAD) to ONEOS6 which supports the separation of the management, control and data plane. It also supports the decoupling of the software from the underlying hardware. However, the resulting ONEOS6 implementation, which is based on Linux and DPDK is still resource intensive.

The target deployment scenario envisages migrating ONEOS6 onto unikernels. The expectation is that this will lead to a significant improvement in system scalability, isolation and security.

In the Home Automation and Internet of Things domain, Nextworks describe *Symphony*, their Smart Home

and Smart Building Management platform. Symphony's functions are currently deployed in virtual machines and containers which communicate through a platform internal networking based on Layer 2 switching technologies and IPv4. The aim is to migrate this to a distributed micro-service architecture, using unikernels for some aspects, with orchestration using ProxMox or Kubernetes, on x86-based hardware. Unikernel support for the following libraries will be required: libc; sqlite3; openssl; libcurl; and libprotobuf.

Nextworks' target deployment scenario is to implement the Symphony IoT middleware funtions and gateways using unikernels, in the building automation and domotic control system that they already have in their premises in Pisa, Italy. The UNICORE Toolstack will be integrated into the Symphony build system. The target deployment will be used to evaluate: consistency and continuity of functionality; usability and flexibility of the toolchain; performance of the processes that have been migrated to unikernels; resource consumption; service reliability; and warm-upgrade of process images.

In the Smart Contracts domain, EPFL describe the use of blockchain technology as a distributed ledger to support smart contracts, which are written in a specially designed language called Solidity. Currently the use of virtual machines to implement this technology imposes some costs and limitations as to how a smart contract can be implemented. It is envisaged that replacing the virtual machines with unikernels would facilitate the writing in other languages (such as Rust, C, and C++) and the execution of smart contracts on diverse platforms. Key requirements are: fast boot; resource budget per execution; deterministic execution over any platform.

The first step in the evaluation of UNICORE with smart contracts will be to use the UNICORE Toolstack to verify that the tools reliably detect when the requirements for deterministic behaviour of a smart contract have not been met. Tests will also be done to ensure that a given smart contract implemented as a unikernel gives exactly the same result when run on platforms with different architectures. The target deployment for smart contracts is a live deployment of *Cothority*, which is the set of nodes involved in the DEDIS blockchain. The initial focus will be on x86 and ARM-based machines. The experiment will involve executing a set of smart contract transactions, written in a generic purpose language, on diverse hardware architectures (x86 and ARM) and diverse hypervisors (e.g. Xen and KVM) and checking that consensus is achieved and that performance is equivalent or better than that achieved using the existing Solidity-based approach.

Chapter 4 of this document details the core requirements that need to be put in place in order to meet the technical requirements of the various application domains. This chapter has sections covering general requirements, API requirements, orchestration environment integration requirements, security and isolation requirements and deterministic execution requirements.

Chapter 5 covers the technical requirements on the Unicore toolchain. Specifically, the requirements on several tools are detailed; A decomposition tool to help developers break down monolithic software blocks, a dependency analysis tool to help determine the minimum set of required libraries and OS primitives, an automatic build tool to help in building Unikernels, a verification tool to help ensure that the functionality

of applications runnning in Unikernels matches that of the equivalent application running on a standard OS. Finally, a performance tool that will help analyse and improve the performance of running applications, is described.

The final chapter of this document presents the conclusions reached during the requirements gathering phase of the project.

Contents

List of Figures

Acronyms

ABI	Application Binary Interface
API	Application Programming Interface
AMQP	Advanced Message Queuing Protocol
ARM	Advanced RISC Machines
ASLR	Address Space Layout Randomisation
AWS	Amazon Web Services
BPF	Berkley Packet Filter
BMS	Building Management System
BNG	Broadband Network Gateway
CLI	Command Line Interface
CPE	Customer Premises Equipment
CPU	Central Processing Unit
CNW	Correct Networks SRL
CSUC	Consorci de Serveis Universitaris de Catalunya
CVE	Common Vulnerabilities and Exposures
DALI	Digital Addressable Lighting Interface
DEDIS	Decentralized and Distributed Systems
DHCP	Dynamic Host Configuration Protocol
DMA	Direct Memory Access
DOS	Denial Of Service
DPDK	Data Plane Development Kit
dRIC	dRAX RAN Intelligent Controller
DSL	Digital Subscriber Line
DU	Distributed Unit
DUT	Device Under Test
EAD	Ethernet Access Devices
eBPF	extended Berkley Packet Filter
ELF	Executable and Linkable Format
EPC	Evolved Packet Core
EPFL	École Polytechnique Fédérale de Lausanne
EVM	Ethereum Virtual Machine
FPU	Floating Point Unit
GDOI	Group Domain of Interpretation
GPU	Graphics Processing Unit

HA	High Availability
HAL	Hardware Abstraction Layer
HVAC	Heating, Centilation, and Air Conditioning
IoT	Internet of Things
IP	Internet Protocol
IPSec	IP security
ISP	Internet Service Provider
KPI	Key Performance Indicator
KVM	Kernel-based Virtual Machine
LTE	Long Term Evolution
MANO	Management and Orchestration
MCAPI	Multicore Communications API
MPLS	Multiprotocol Label Switching
MSAR	Multi-Service Access Routers
MQTT	Message Queuing Telemetry Transport
NAT	Network Address Translation
NATS	Neural Autonomic Transport System
NIC	Network Intercace Card
NF	Network Function
NFV	Network Function Virtualisation
OCI	Open Containers Initiative
ODM	Original Design Manufacturer
ONAP	Open Network Automation Protocol
ONVIF	Open Network Video Interface Forum
OPC	Open Platform Communications
OPC-UA	OPC Unified Architecture
OS	Operating System
PBFT	Practical Byzantine Fault Tolerance
pCPE	physical CPE
PLR	Packet Loss Ratio
PNF	Physical Network Function
POS	Performance Oriented Scheduler
PTZ	Pan Tilt Zoom
QoS	Quality of Service
RAM	Random Access Memory

RAN	Radio Access Network
Redis	REmote Dictionary Server
REST	REpresentational State Transfer
RGB	Red Green Blue
RRD	Round Robin Database
RTU	Remote Terminal Unit
S3	Simple Storage Service
SCF	Smart Contract File
SCTP	Stream Control Transmission Protocol
SDN	Software Defined Networking
SDWAN	Software Defined Networking in a Wide Area Network
SIP	Session Initiation Protocol
SNMP	Simple Network Management Protocol
SQL	Structured Query Language
TCP	Transmission Control Protocol
TRL	Technology Readiness Level
UDP	User Datagram Protocol
UE	User Equipment
UI	User Interface
UIT	Universitat Internacional de Catalunya
pCPE	virtual CPE
vCPE	virtual CPU
VM	Virtual Machine
VMM	Virtual Machine Monitor
VoD	Video on Demand
VoIP	Voice Over Internet Protocol
VNF	Virtual Network Function
vRAN	virtualised Radio Access Network
XDP	eXpressive Data Path

1 Introduction

This document is deliverable 2.1 of the UNICORE project. Its purpose is to define the requirements for the UNICORE Core and for the UNICORE Toolstack, and for the trials that will be used to evaluate these implementations in practical applications. The Core implements the core functionality of UNICORE lightweight virtual machines (unikernels), namely the μlib library, together with implementations of security primitives and deterministic execution support. The Toolstack consists of a number of tools which together support the development of applications that can be deployed using the Core technology.

Chapter 1 provides an introduction to the rest of the document.

Chapter 2 explains the methodology by which the requirements in chapters 6 and 7 are obtained.

Chapter 3 describes the application domains that are in the scope of the project from the points of view of each of the relevant project partners. There is one section for each application domain and a sub-section for each partner-specific view of that domain. This sub-section also describes the target deployment scenarios (trials) that will be used to evaluate the UNICORE Core and the UNICORE Toolstack.

Chapter 4 specifies the technical requirements for the UNICORE Core, organised into several categories.

Chapter 5 specifies the technical requirements for the UNICORE Toolstack. There is one section for each tool.

Chapter 6 summarises what has been achieved in this document and any shortcomings that have been identified.

2 Methodology

This section describes the method that has been adopted for determining the requirements for both the UNICORE Core and the UNICORE Toolstack.

The initial focus of this document is on the requirements arising from the application domains that have been identified based on the business and academic interests of the various project partners.

The approach taken to identify the requirements is first to consider the application domains of interest. For each application domain there is more than one project partner with an interest in that domain, so each such partner describes the domain from its point of view. First, in chapter 3, the state of the art is described, based on the approach and architecture currently used by the partner for the software they develop for the domain in question. In general, this forms a baseline, against which, in order to be successful, the UNICORE approach will have to facilitate improvements in terms of cost, flexibility, security or performance. In most cases, when applying the UNICORE technology, the partners will want or need to retain certain aspects of their current approach, which gives rise to some essential requirements for UNICORE. Other requirements related to each application domain may be deemed to be desirable or nice to have, rather than essential.

The second step is for the partners to describe the specific target deployments in which they will apply and evaluate the Unicore Core and Toolstack. These proposals will give rise to specific requirements for the Core and Toolstack.

An analysis of the requirements identified by the application domain partners in conjunction with wider experience from years of building Unikernels and discussions with experts in the field, was used to identify additional requirements on the UNICORE Core and Toolstack i.e. apart from specific technical requirements, more general requirements also needed to be taken into consideration. This covers aspects such a functionality, portability, ease of use, maintainability and performance. In order to not artificially limit unikernel usage it was also necessary to consider Unikernel asage across multiple platforms. These platforms do not only include hypervisors such as Xen and KVM but also container based solutions such as OCI and Docker.

3 Application Domains

3.1 Serverless Computing

3.1.1 Digital Content Deployment Scenarios (CSUC)

CSUC has worked for years hosting, developing and implementing different digital repositories focused on digital content for the university community, in concrete these kind of repositories are called institutional repositories.

Each institutional repository stores its own documentation related to teaching, research and institutional documents. Concrete examples in this scenario are two university repositories: IRTA PubPro and UIC Open Access Archive.

These institutional repositories are based in DSpace. DSpace is an open source software that provides tools for the management of digital collections and it is used as an institutional repository solution and is adapted to the norms, standards, and good international practices.

The different repository institutional admins can upload documents in these repositories that can consist in: pdfs, videos, images, etc.

3.1.1.1 Architecture

The architecture behind these repositories is composed by different Docker containers and virtual machines as shown at figure 3.1 . At top of the stack is Rancher which is an open source software platform that enables organizations to run and manage Docker and Kubernetes in production and works as a container orchestration and scheduling.

The nodes of Rancher are:

- Bastion host which secures the internal network
- 3 Rancher Master nodes in High Availability
- 3 Kubernetes Master nodes in High Availability
- Kubernetes Worker nodes

All these nodes are virtual machines virtualized through KVM using OpenNebula as a orchestrator. OpenNebula is a cloud computing platform for managing hybrid distributed data center infrastructures, and is the solution at CSUC to build hybrid implementations of infrastructure as a service.

Kubernetes is a portable, extensible open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation, as well as scalability ecosystem. Kubernetes coordinates a highly available cluster of computers that are connected to work as a single unit.

The Rancher Master nodes manage the authentication, scheduling, and deploying of different Kubernetes Clusters. A Cluster in Rancher is a group of physical (or virtual) compute resources, in our case, the Ku-

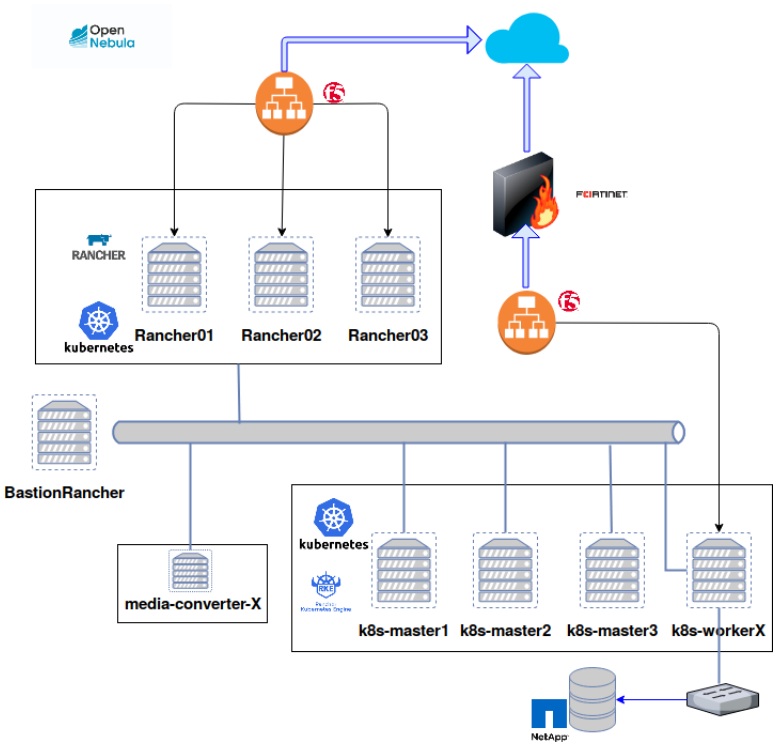

Figure 3.1: CSUC Deployment Scenario

bernetes Clusters are the resources. The Rancher Master nodes communicates with the Kubernetes Cluster through the kubectl CLI, in concrete with the kubernetes master nodes.

Kubernetes master nodes are stateless and are used to run the API server, scheduler, and controllers. They make global decisions about the cluster, and detect and respond to specific situations in order to start up a new pod for example.

The basic scheduling unit in Kubernetes is a pod. It adds a higher level of abstraction by grouping containerized components. A pod consists of one or more containers that are guaranteed to be co-located on the host machine and can share resources.

Kubernetes worker nodes run the different pods and provide the Kubernetes runtime environment. Every node in the cluster must run a container runtime such as Docker, as well as another components, for communication with master nodes for network configuration of these containers.

Once the description of the architecture is done, we can define a single repository application as composed by a pod running a DSpace instance, as well as running the database, and another virtual machine called ”media converter”.

The media converter node is not inside the Rancher/Kubernetes architecture, is an independent virtual machine managed by OpenNebula. The different repository institutional admins can upload the digital content to the repository, but some of them could be so huge that their visualization is not possible with their original raw format. The media converter executes a cron every night in order to detect different content that need to be compressed in smaller size so is not a heavy load to distribute its visualization from the server.

3.1.1.2 Benefits of Using Unikernels

When the digital content is upload, at night the media converter cron job look for new content. It has two steps, the first one converts the images to new reduced images, and the second one converts the videos to new video format compressed with mp4.

For the image convert step the media converter uses *ImageMagick* and for the video converter step it uses the *ffmpeg* utility. Both of them detect the format of image or video and then apply the corresponent template to convert to another format encoding less heavier and easily web broadcasted.

CSUC can take advantage of the unicore technology in order to do the job of the media converter to change the videos and images to another format. To obtain the dependencies libraries we'll use the decomposition tool and the dependency analysis tool.

The media converter virtual machine uses lot of resources (from 4 vcpus and 12Gb of RAM), and there is one media converter for each institutional repository, so we hope that with unicore the resources will be drastically reduced and that the time of conversion will be variable depends of the origin size video or image.

3.1.1.3 Requirements on Unikernels

The requirements for serverless computing use case on CSUC the *boto3* library will be necessary to manage the files by the S3 protocol. This library is written in Python so for this purpose is only needed to support Python3. For the image conversion a script written in Python will be developed using the *Pillow* library.

The *Pillow* library requirements are the following:

- python3
- python3:any
- mime-support
- python3-pil.imagetk
- libc6
- libfreetype6
- libjpeg8
- liblcms2-2
- libtiff5
- libwebp6
- libwebpdemux2
- libwebpmux3
- zlib1g
- python3-tk
- libgcc1
- libpng16-16
- libjpeg-turbo8
- libjbig0
- liblzma5

Also it is planned to use all components developed on the unicore toolkit in order to create the unikernel image which better fits on the purpose. The most important components would be the Automatic Build Tool in case we can achieve a way to build image on instantiation time, the verification tool and the performance

optimization to provide more information about how it works the used image.

3.1.1.4 Performance Expectations

The trials will consist on instantiating a specific number of unikernels, virtual machines and kubernetes, if it's affordable, on nodes with the same characteristics and measure the deployment time, see how many instances are possible to run and how many resources they consume and also check the time it takes to complete the same task one by one.

After that the results has to be compared and see if unikernels fulfill the following:

- Lower deployment time
- Higher number of instances or tasks a node can run
- Lower time to finish the task
- Lower resources consumption

3.1.1.5 Description of Proposed Trial

Following the previous domain application introduction the CSUC use cases will focus on the media converter service which is in charge of convert images to more lightweight formats. The goal is to change how CSUC converts these images by replacing the media converter service (now provided by virtuals machines) by an unikernel serverless solution. This new solution has to improve the behaviour of the virtual machines leveraging unikernel mainly characteristics like low deployment time, reducing resource consumption and a lifetime limited to the time it takes to convert a file unlike virtual machines which has a worst deployment time, bigger resource consumption and always are running independently if they are converting a file or idle.

Architecture

The figure 3.2 shows a possible architecture for the CSUC serverless solution. The components involved are the dspace repository app from where a user uploads the image. The input storage backend is the space where the files to be converted are placed and the output storage backend is where the already converted images are placed, in CSUC case will be both an on premise S3 protocol bucket solution. The queue service will monitor the conversion tasks to be started and also will send this information to the orchestrator and will check if the task is done. The orchestrator will receive the commands from the queue service to start as many unikernels processes as files would be in the input storage backend to convert it. The hypervisor, KVM, will run the unikernels and assure the isolation between them, considering unikernel processes as if they were virtual machines.

So the workflow starts by a user uploading an image. This image will be stored in the input bucket storage. The queue service will notice there is a file to convert and will inform the orchestrator to start a unikernel. The orchestrator will instantiate a unikernel over hypervisor and this unikernel will get the image from the input storage backend and will start to convert it. When the convert process finishes the orchestrator put the

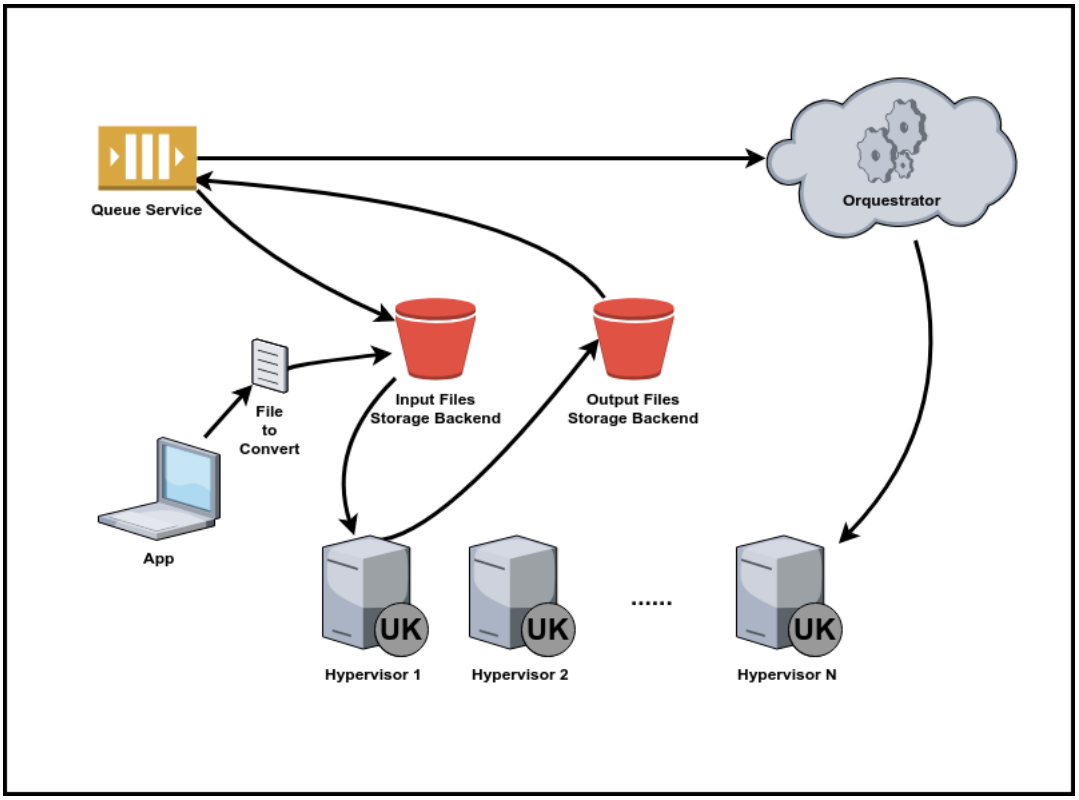

Figure 3.2: Serverless architecture of media converter service

image on the output storage backend and will die. After that, the queue system will check if the conversion has finished and will inform the dspace repository app that it has the image available.

Use Cases

As orchestrator CSUC uses OpenNebula to manage Virtual Machines over KVM and Rancher to manage kubernetes with Pods over OpenNebula. The main CSUC use case is focused on developing an OpenNebula driver to enable OpenNebula to manage the different kind of Unikernels responsible for converting the image files. OpenNebula will store in its datastores the different unikernels ”images” previously prepared with UNICORE and deploy it over KVM as if it were a Virtual Machine and run the task as explained previously, all integrated with the S3 buckets and the queue service. The communication between the queue service and OpenNebula can be done by the OneGate service (provided by OpenNebula) whose goal is to escalate the services depending on different parameters. Thus, when OneGate gathers the required information it will decide whether to instantiate more Unikernel processes via the KVM hypervisor.

Another desirable use case will be rancher over kubernetes but instead of launching docker containers over virtual machines the idea is to launch unikernels via the KVM hypervisor from a Rancher platform. It is intended to explore the possibility of using Rancher over Kubevirt and as in the previous use case, try to develop some connectors in order to launch unikernels. An additional use case would be to test if its possible to integrate unikernels on any Function-as-a-service open source platform such as openfaas.

3.1.2 Lambda Packet Processing Deployment Scenarios (CNW)

3.1.2.1 Existing Architecture

Serverless services are growing rapidly (28% per year expected growth) in usage and revenue. This is mainly due to its novel billing model: clients, instead of paying for the total time their infrastructure was up and running, pay for the amount of work that was completed. By work we refer to the number of API calls that were carried out. In our view this is a predictable evolution of cloud services. Historically, cloud took away infrastructure provisioning and maintenance duties from businesses and transferred those to the cloud provider. With lambda services, cloud providers also take over the task of streamlining their infrastructure operations. More concretely, businesses do not have to worry about idle instances, over provisioning, infrastructure life cycle management, since all these are implemented by the lambda framework and any costs due to inefficiency are accounted for by the cloud provider.

Having said this, we believe that lambda-like services for packet processing workloads are a natural evolution of existing cloud services.

To understand better the technical challenges, let us consider the current state of affairs in the field of Network Function Virtualisation (NFV).

The de-facto technical implementation of this scenario is to:

- Run every function within a single Virtual Machine (VM), to provide security and isolation.
- Within the VM, employ one of the available kernel bypass frameworks such as the industry standard DPDK, or others (Netmap, pfRing, snabb etc.).
- Spend one vCPU for polling for received packets and pulling these from NIC memory to user space buffers in order to be processed by another vCPU

Figure 3.3 depicts this deployment scenario.

From this setup there are a couple of lessons to be learned and areas to improve.

In terms of lessons to be learned, we state the following:

- VMs are the preferred context for deploying customer processing(NFs), as opposed to containers or plain processes. The fundamental reason to support this claim is the improved isolation model proposed by VMs.
- The Linux networking stack can be a bottleneck in the case of NFV, and even if it isn't, one can argue that for very specific packet processing workloads as proposed by NFV it can be an overkill.

3.1.2.2 Benefits of Using Unikernels

In order to make PacketCloud a valuable proposition we have to address the following:

- VM size:

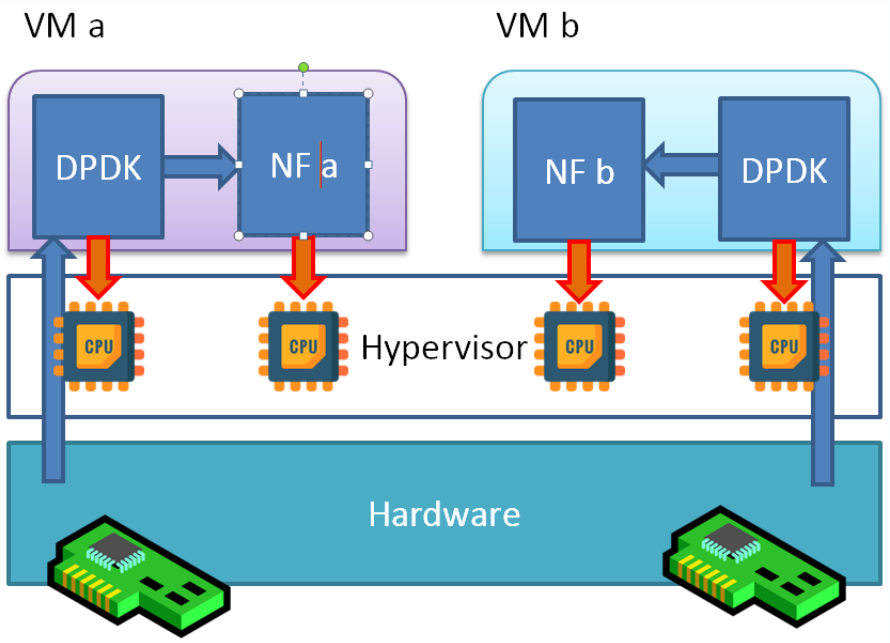

Figure 3.3: DPDK-based lambda service

(i) Kernel tailored for its NF instance.

(ii) Minimal runtime environment (tools and libraries)

- Flexible networking stack, instead of the take it (Linux stack) or leave it (kernel bypass). Ideally, the NF should benefit from the existing stack code for as much as needed.

We believe unikernels in general and Unikraft's ecosystem of libraries and tools deliver on all points mentioned above which we consider key to the success of a product such as PacketCloud.

3.1.2.3 Requirements on Unikernels

Our target scenario is to offer clients the ability to deploy per-packet network functions running in the ISP's cloud. This sections outlines the key technical challenges towards making PacketCloud a valuable product. For each such aspect, we highlight the means in which unikernel technology and ecosystem can aid achieving them.

In order for PacketCloud to become a valuable product, we strive for the following:

- Efficient resource utilization. The two halves of this challenge are:

  (i) High efficiency under load - packet lambdas running on PacketCloud should minimize the amount of resources required per call.

  (ii) High efficiency when idle - the time between two lambda calls should require little to no resources. To achieve this we require an efficient pause/resume mechanism in order to be aggressive with the passivization strategy employed by the lambda scheduler. In this regard, rapid boot of small VM images is critical, hence the potential value of unikernel-based VMs.

- Strong isolation and security guarantees.
- Flexible API Efficient and secure as it may be, PacketCloud will not be a success unless clients can implement their desired NF functionality. To achieve this, our aim is to support the eBPF computation model for NFs. Clients would, thus, deploy eBPF code that will run per-packet. However, great flexibility comes with great technical challenges:

  (i) Based on the type of processing deployed (inspected using **Dependency Analysis Tool**), the **Automatic Build Tool** will support the construction of small VM images, provisioned with an execution environment as frugal as possible, built to suit the deployed lambda - and nothing else.

  (ii) Since custom per-packet processing is a very open proposition, with very severe failure models (network black holes, traffic flooding, various DOS attacks, invalid packet modifications), we aim for strict security and correctness guarantees, a task where **Verification Tool** proposed by Unicore would definitely help.

3.1.2.4 Performance Expectations

In PacketCloud's case we characterize performance based on:

- Main memory footprint For a VM running a service in the cloud its physical memory requirements are dictated by:
  - The memory requirements of the service application itself and the libraries it uses.
  - The memory requirements of the kernel.

  The smaller these requirements, the better consolidation, thus efficiency, a cloud provider can achieve. While improving application memory usage is beyond the scope of PacketCloud, by using custom kernels, specific to the application we can improve the overall memory usage.

  In this respect, we expect VM sizes in the order of a few megabytes - *10-20MB*.

- Quick boot times If we can reduce boot times for PacketCloud VMs, we can develop a VM boot and shutdown cycle that follows traffic packets. In other words, when the VM is idle we can shut it down and boot it whenever the host detects traffic for the inactive VM.

  For this to be feasible in practice, the boot times should be small enough not to incur large packet processing delays and large packet buffer requirements.

  Having said this, we expect *sub second* boot times, depending on the application.

3.1.2.5 Description of Proposed Trial

The technical architecture intended for PacketCloud is based on the principles:

- Efficiency

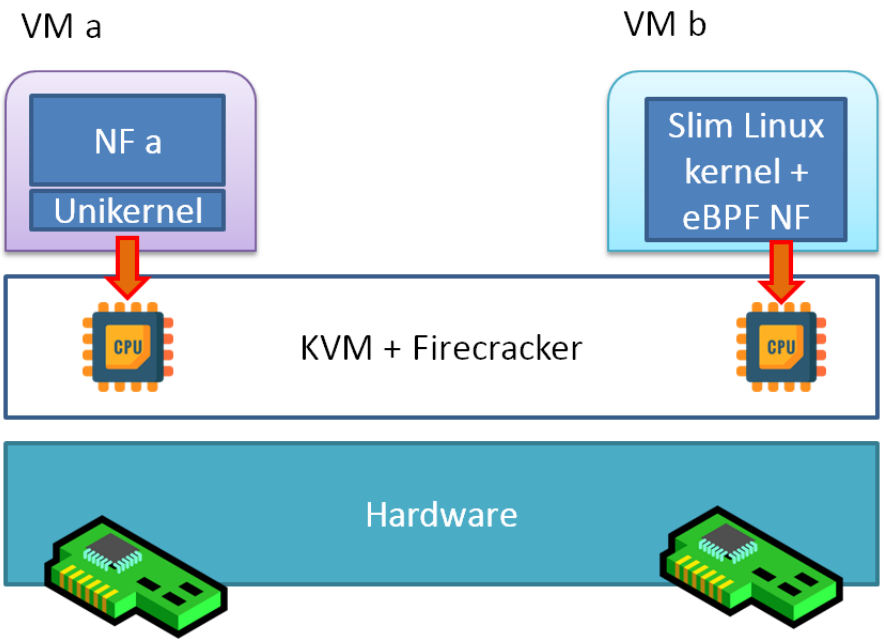

Figure 3.4: PacketCloud deployment overview

- Security
- Flexibility

Next we offer a deployment overview for PacketCloud (as depicted by 3.4):

- PacketCloud relies on the industry standard KVM hypervisor, and Firecracker efficient Virtual Machine Monitor (VMM) as the runtime environment for the minimalistic VMs that encapsulate the NFs.
- We will make available two runtime flavors:

  (i) VM a incorporates a pre-compiled NF, together with a purpose-built unikernel. VM b runs a slimmed down version of Linux, pre-configured with the eBPF calls corresponding to the NF code at various points in the stack, be it immediately after the DMA transfer from the NIC to main memory using Linux eXpressive Data Path (XDP) or up to Socket layer. Besides the kernel the VM will run no other code, since the NF is part of the kernel itself.

- For managing deployment, pause, suspend and other life cycle events of such PacketCloud VMs we plan on developing in house a solution such that it can be used standalone (for resource-constrained environments), or to be integrated with an already existing orchestrator, such as Kubernetes.

3.2 Network Function Virtualization

3.2.1 Broadband Network Gateway Scenarios (Orange)

Orange's target for UNICORE work is determined by the definition of a novel approach to the implementation of Broadband Network Gateways (BNGs). Different implementation models are considered:

(i) An evolution from the current physical monolithic implementation to a virtualized monolithic implementation.

(ii) A further evolution to a virtualised deployment using unikernel VMs.

There are expected to be several work streams defined for the Broadband Network Gateway (BNG) use case:

- **Use case workflow**, including design, preparation, execution, system and use case monitoring, analysis and validation for BNGs as unikernel's VMs and Key Performance Indicator (KPI) evaluation.
- **Implementation workflow**, including activities related to virtualized infrastructure implementation, unicore lightweight VM as network functions for every single (v)CPE
- **Transformation workflow**, a technical evaluation of:
  (i) A single Physical Network Function (PNF) BNG that provides a single network service function for several (v)CPEs.
  (ii) A single Virtual Network Function (VNF) BNG that provides a single network service function for several (v)CPEs.
- **Outcome workflow**, evaluating the unikernel BNG as NFV/NFV implementation in comparison against the other scenarios that are considered i.e. a monolithic NFV and a monolithic PNF

The main objective for ORANGE is to define and implement the BNG unikernel application in a virtualized environment, within the use-case life cycle approach for service requirements analysis, design, system requirements, overall architecture for implementation, including control and management for use case service and infrastructure resources. The technology developed should enable the seamless creation and deployment of any unikernel Unicore application with due consideration of performance, scalability, security, isolation, efficiency.

The entire system is based on an efficient NFV, relying on Unicore to develop the BNG lightweight network function for improved performance from the end-user perspective and efficient resource optimizations from the service-provider perspective. The entire system will run on dedicated lightweight VMs, instantiated in an automatic manner and orchestrated on a per customer application basis.

Today's ORANGE BNG implementation is based on several physical BNGs (Nokia 7750 hardware), deployed in different locations in the network, in ORANGE Data Centers. Customers from different regions are connected to their dedicated BNGs, for clarity, by defining a region, Region A, the clients from that specific region will connect to Region A BNGs, as shown in figure 3.5.

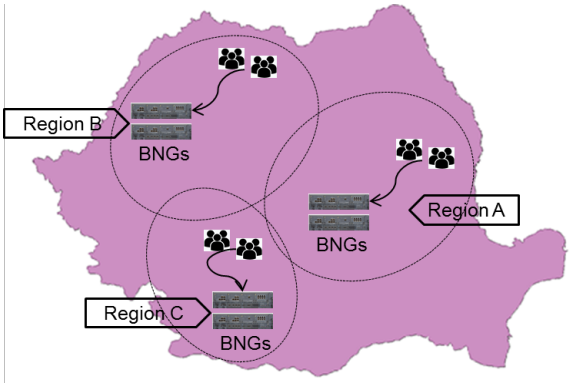

Figure 3.5: Orange BNG Use Case Scenario

Currently, a resilient system is deployed at each site location. This consists of two (active/standby) BNGs being deployed for High Availability (HA) purposes. This pair of the BNGs supports all traffic from the region, for multiple services (Internet, VoIP, etc). The network provides, connectivity for different customer applications with a capacity of up to 1Gbps along with QoS and user monitoring. The entire system is designed to cope with an estimated traffic load and number of customers, including specific authentication access and service allocation resources. Deployments using this monolithic approach can take up to several months, regardless the number of customers and specific service requirements and needs.

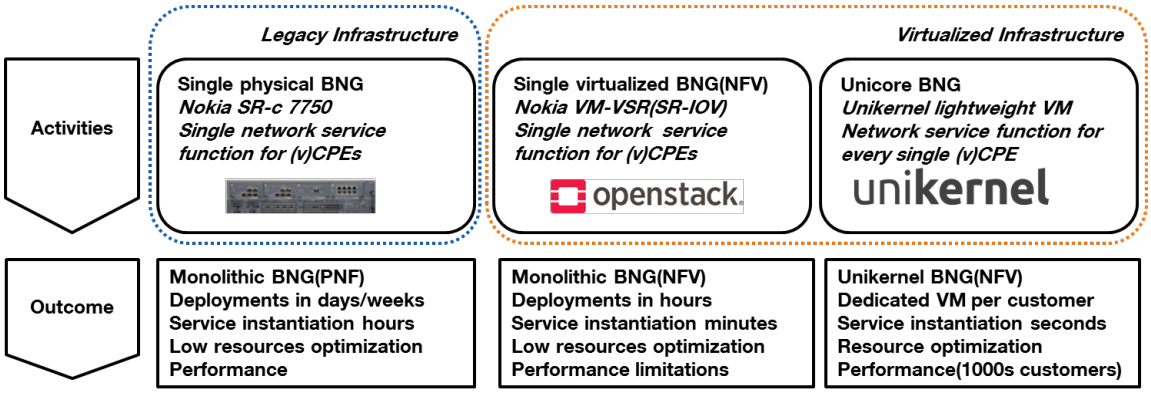

Figure 3.6: BNG Implementation Evolution

The next phase implementation will consist of replacing the physical BNGs (PNFs) in the service provider Data Centers, with virtualized infrastructure in an NFV/VNF deployment scenario. This approach of replacing the PNFs with VNFs is primarily designed to improve the deployment time and resource utilisation of the BNGs.

From a service perspective the functionality of both deployment approaches is the same, but there are

obvious improvements in deployment times and resource usage. However, the VNF/NFV approach does leave some questions open regarding VM performance, resource utilisation and security. This evolution path is shown in figure 3.6.

3.2.1.1 Existing Architecture

The legacy architecture, as shown in figure 3.7 is a compact physical platform which enables service delivery, high performance, dense interfaces and high user capacity equipment, with comprehensive features and different network functions, using common system modules, a network hardware architecture containing control processing modules, I/O modules for traffic forwarding programming, RIB APIs, filtering capabilities and media adapters modules providing physical interface connectivity. On top of the architecture, several features and protocols are supported, such as IP L3/L2 and MPLS features, Segment Routing and SDN, control interface, management and configuration interfaces as CLI, OpenFlow, Netconf, YANGmodels, SNMP, OAM fault and performance operation.

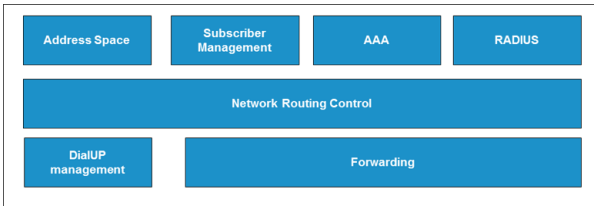

Figure 3.7: BNG Legacy Architecture

3.2.1.2 Benefits of Using Unikernels

The first version of BNGs were based on physical equipment handling all control, management and network functions. As mentioned previously, there are several drawbacks to this approach, such as cost and resource allocation, the time taken to deploy equipment, poor scalability and other general drawbacks of this monolithic solution approach.

As shown in figure 3.6, the approach of replacing the BNGs with vBNGs, whilst otherwise retaining the monolithic solution, results in improved deployment and service instantiation times, but still suffers from poor resource optimisation and performance limitations.

The BNG virtualized architecture using NFV and VNFS, will enable the decoupling of the software from the underlying hardware. This allows for the separation of the Control Plane (CP) and User Plane (UP). In this scenario, the CP takes responsibility for the user control management component and the UP is responsible for policy implementation and data forwarding.

This virtualized BNG will centralise management, is scalable for the management of subscribers and will enable flexible network resource allocation.

The functional components, as described in 3.8, are implemented as Virtual Network Functions (VNFs),

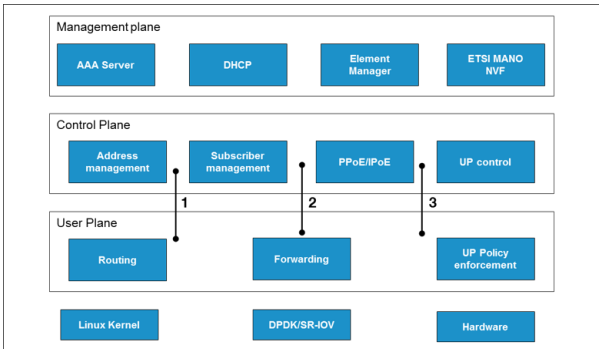

Figure 3.8: BNG Control and User Plane Separation

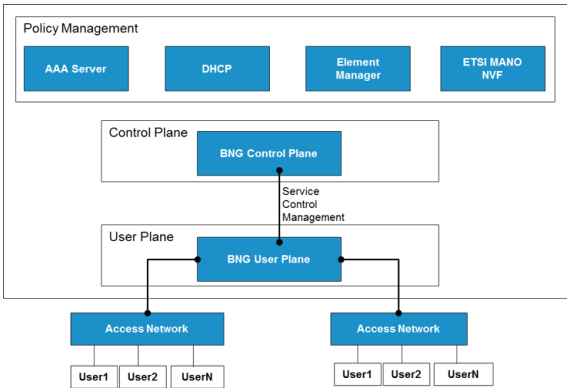

Figure 3.9: Virtualised BNG Deployment Scenario

hosted in Network Function Virtualization Infrastructure (NFVI). The proposed NFVI model is ETSI MANO NFV. ETSI MANO identifies Service (1), Control (2) and Management (3) interfaces and details the interactions between the CP and UP. This deployment is shown in figure 3.9.

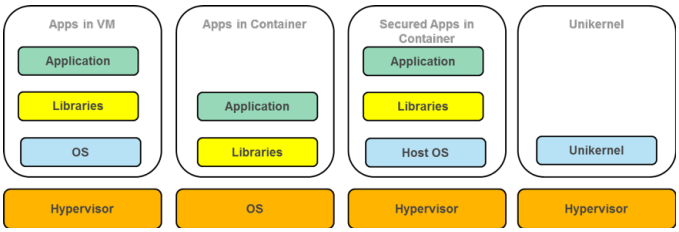

Figure 3.10: BNG Transformation Apps for Unikernel Implementation

A migration to a micro-services architecture facilitates more fine-grained distribution of BNG services and allows for more flexible deployment scenarios. In the industry, there is a movement away from the current use of VM and container technologies towards the introduction of smaller and more scalable NFVs with automation and orchestration and seamless scaling. BNGs deployed in Unikernels would lead to a smaller, faster approach to deploying the BNG application in a cloud infrastructure. As shown in figure 3.10, deploying BNG functionality as lightweight applications in unikernels will overcome the limitations outlined in deploying physical infrastructure.

3.2.1.3 Requirements on Unikernels

- Deployments: Kubernetes or Openstack orchestration, bare metal, x86/per-customers
- Management plane service, resource and service orchestration OSMv5/ONAP, in Openstack/KVM context
- Control plane service APIs
- User Plane forwarding services:
  - deploy on the top of infrastructure up to 1000 vBNGs, per instance an average traffic of 100Mbps per session flow, 24.000packest/s
- Unikernel based services on top:
  - DPDK
  - Openssl
- Network Libraries
- Protocols:
  - Network routing protocols: static; BGP; OSPF; ISIS (this list is open, at least one of the dynamic protocols should be supported)
  - Management: AAA Server; DHCP
- Scalability
- Service isolation
- Security

3.2.1.4 Performance Expectations

Control Plane and User Plane performance:

- services deploy on the top of infrastructure up to 1000 vBNGs for 1000 vCPEs
- instance an average traffic of 100Mbps per session flow, 24.000 packet/s

3.2.1.5 Description of Proposed Trial

The Unicore unikernels implementation assumes the decomposition of monolithic BNGs into a number of unikernels, with one unikernel per customer. The decomposition starts from the commercial monolithic BNGs service approach, based on the unikernel BNG provided by NEC. The Unicore lightweight BNGs VMs should provide, per customer, at least the same performance as the monolithic one. The change is provided

through the physical separation of client's application domains, providing the capability and flexibility to provide customer specific implementation and resource allocation whilst allowing for optimization and fast deployment of the services.

A per customer unikernel VM is a completely different deployment model. It will enable per customer and network monitoring and resource control, improved capabilities and capacity for customers and the possible application of several alternate subscription models.

From a high level perspective, the Unicore unikernel implementations should provide the possibility of:

- Lightweight, dedicated VMs deployments (per customer or per CPE).
- Automated service instantiation.
- Scalability capabilities.
- Per region service provisioning.
- Tools for monitoring resource allocation, service KPIs and metrics performance.

The entire Unicore system is expected to be supported on a virtualized environment, VNFs with VMs Openstack (KVM hypervisor) based on containerised infrastructure, with the possibility of introducing several open tools for orchestration and service instantiation. Infrastructure platform deployments will support specific integration of lightweight BNG's VMs. The scenario should be extended with the purpose of supporting the testing and use case validation of more than 1000s (v)CPEs, instantiated and connecting to the proper light BNG, into an end-to-end scenario. The performance of this proposed system should respect customer service KPIs and network KPIs, in terms of bandwidth, delay, provisioning time, capacity, resource consumption and efficiency.

A successful overall implemention of unikernel applications should be evaluated when deploying more than 1000s Unicore vBNGs (vCPEs instances), defining several service capabilities and characteristics, providing ranges of speed and QoS profiles, authentication mechanism and traffic restrictions for user, into an isolated and secured light process. For service instantiation and resource configuration on the deployed infrastructure, orchestration tools are mandatory to be used, tools used from the open tool community, such as OSMv5 for resources orchestration, ONAP for resources and services orchestration, in a virtualized (Openstack based) or containerised (Docker based) scenario. The use case implementation is not limited to any orchestration tool and can be integrated to any other component, adopting also service orchestration capabilities. The orchestration process is intended to be adapted and integrated accordingly into the testbed infrastructure, automation and software programmability of the system being seen as a mandatory resource apps block and different APIs implementation for control, management and service instantiation, with some measurable cost reduction in case of development.

The ORANGE BNG use case, as implemented on Unikernels, requires advanced security features, at the BNG application level, to minimize the exposure to different security attacks (Unicore by design). Hardware and

platform security level are more relevant for a telco operator and this is achieved through the decomposition of a monolithic application into smaller building blocks which can be executed in isolated environments.

3.2.2 5G vRAN Scenarios (Accelleran)

Accelleran has for several years been developing software for small cells (base stations) for 4G mobile networks. Although this software has always been architected for independence from hardware, operating system and third party protocol stacks, to date commercial releases have always been in the form of embedded software running on specialised Original Design Manufacturer (ODM) hardware. This is exemplified by the E1000 series of small cell products. The E1000 provides a single, low-powered Long Term Evolution (LTE) (4G) cell and is intended for deployment in enterprise, public urban and suburban scenarios as well as remote and rural scenarios. Its hardware is based around the Marvell Octeon Fusion-M CNF7130 quad-core baseband processor, and the Accelleran software on it runs under the linux operating system.

With the advent of 5G mobile networks, the concepts of NFV and Radio Access Network (RAN) disaggregation have come to the fore. These concepts break down the existing network elements into smaller building blocks, many of which can be deployed in a NFV environment.

3.2.2.1 Existing Architecture

Accelleran has taken its existing embedded software and demonstrated that it can be adapted to run in virtual computing environments with relative ease, leading to its planned dRAX^{TM} product offering, which is currently at Technology Readiness Level (TRL) 3-4.

The outline dRAX architecture is shown in figure 3.11.

The initial dRAX solution is composed of the following services:

- dRAX RAN Intelligent Controller (dRIC)
- dRAX Information Base
- dRAX Data Bus
- Cell Control Plane

which run on a general compute server.

To form one small cell, one Cell Control Plane instance connects to remote unit over a 3GPP standard interface (F1 for 5G, W1 for 4G). The remote unit consists of baseband hardware that runs layer 2 and layer 1 software, and a radio front end. The cell also has a user plane component which currently also runs on the remote unit - offloaded user plane will be supported in the future. The dRIC manages a set of such small cells. If the dRIC sees a new remote unit start up, it automatically spins up a Cell Control Plane instance to be paired with it.

The dRIC and the Cell Control Plane instances communicate over the dRAX Data Bus which is implemented using the open source message broker NATS. To guarantee platform and language independence, messages are encoded using Google's Protocol Buffers framework. The dRAX Information Base holds configuration data and state information in a distributed datastore that is implemented using Redis.

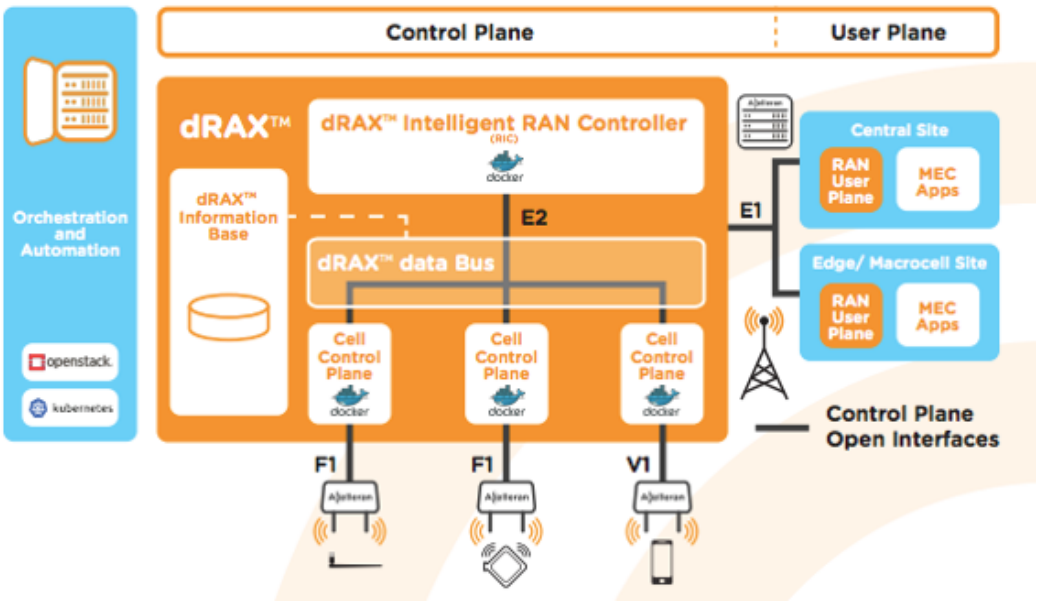

Figure 3.11: Outline dRAX Architecture

The various dRAX services run in Docker containers. There is one Docker container for the dRIC, one for the dRAX Data Bus, one for the dRAX Information Base and one for each instance of the Cell Control Plane service. The container orchestration is provided using Kubernetes: each Docker container runs in a Kubernetes pod. Depending on customer demand, however, OpenStack or similar could be used for orchestration instead of, or in conjunction with, Kubernetes.

3.2.2.2 Benefits of Using Unikernels

While the existing dRAX solution makes it possible to deploy a large part of the small cell software and the dRIC in a virtualised environment, the breakdown into different services is rather coarse. In particular, the Cell Control Plane service consists of several components that could run as separate services. Furthermore, some of these services serve multiple UEs (a UE is a user equipment, such as a mobile phone, or a mobile-broadband dongle). It should be possible to break these down further so that each instance of a service serves a single UE. Deploying these instances as unikernels would allow them to spin up much more quickly, potentially giving the responsiveness that is needed when a new UE connects to a cell. This would result in certain resources being allocated in proportion to the number of connected UEs, rather than each service having a fixed capacity as it does now.

In general the lower overhead of unikernels compared to a Docker container in a Kubernetes pod should make it feasible to decompose the system into smaller - and simpler - services. This should bring a significant benefit in terms of reliability, as there will be less coupling between services compared to when they are implemented together.

The level to which Accelleran's current components can be broken down and assigned to separate unikernels - and hence how many unikernel instances will be needed to run the application - will be evaluated during the project.

3.2.2.3 Requirements on Unikernels

Typically, we would expect to deploy our unikernel-based services using Kubernetes or OpenStack orchestration on linux-based or bare-metal x86 servers or similar.

In order to minimise changes to existing Accelleran software, it must be possible for the services implemented in unikernels to continue to use NATS, Google's Protocol Buffers and Redis.

The dRAX services are dependent on `libc`.

The Cell Control Plane service uses the linux kernel module `sctp`, so it is essential that this, or a mature viable alternative, be available in the UNICORE unikernel implementation.

Third party dependencies include:

- zlog
- cnats (client side only)
- redis (client side only)
- sqlite3
- openssl
- libcurl
- libprotobuf

3.2.2.4 Performance Expectations

A small cell that is implemented (partly or wholly) using unikernels must have the same - or better - performance compared to a fully embedded small cell with the same functionality. In the Accelleran case, the baseline for comparison would be our fully embedded E1000 product. The KPIs will depend on which dRAX components are migrated to unikernels.

3.2.2.5 Description of Proposed Trial

The target deployment scenario used to evaluate the UNICORE technology in the Virtualised RAN application domain is a lab-based, self-contained 4G/5G mobile network. As a minimum, this will consist of:

- An Evolved Packet Core (EPC) from Attocore
- Accelleran's dRAX
- At least two Accelleran E1000 series small cell units
- Accelleran's in-house target test system consisting of up to 16 commercial LTE modems together with controlling software

Within the timescale of the project we envisage being able to extend this to include 5G RAN components.

Accelleran will take its existing dRAX solution described in section 3.2.2 and apply the UNICORE tools to one or more of the components that are currently deployed in Docker containers, in order to migrate them to unikernels. We will take a phased approach to this, first migrating the Cell Control Plane component to a unikernel and, if that is successful, then migrating the other dRAX components. In the first phase, therefore, dRAX will consist of both Docker-based components and unikernel-based components.

In order to demonstrate that the UNICORE technology can be deployed effectively on low cost hardware, we will run the dRAX components on an Intel NUC (e.g. Intel i7 processor with 32 GB of RAM) rather than a high performance Xeon-based server. Container orchestration will be provided using Kubernetes. OpenStack orchestration could also be considered if there is sufficient time.

In order to evaluate the performance of the unikernel based Cell Control Plane, we plan to artificially generate a high level of signalling for paging. Depending on the test equipment available, it may also be possible to arrange for many UEs to connect within a very short space of time, or to trigger handovers from one cell to another. Both of these scenarios would also place a high signalling load on the control plane.

Consideration will be given to developing a software simulation of additional small cells that can connect to the dRAX to provide additional load for performance testing.

If available at the time, a UE simulator such as the TM500 from Viavi, or similar, may be used to simulate up to 256 UEs, again for load testing.

If a software based cell Distributed Unit (DU) simulator is available, performance evaluation of the dRIC could be based on the time taken to spin up many Cell Control Plane instances.

3.2.3 MSAR and EAD (Ekinops)

Ekinops has, for several years, been developing branch office and mid range MSAR as well as EAD. The first versions of these products are characterised by a tight coupling between the software and the underlying hardware. Four years ago, in response to new market requirements and trends, Ekinops has deeply re-architected its core software named ONEOS. The new version (v6) meets the Software Defined Networking (SDN) and NFV principles:

(i) SDN: separating the management plane, the control plane and the data plane.

(ii) NFV and Virtualization: decoupling the software from the hardware.

Indeed, the new management plane is now based on a confd server that handles both the Command Line Interface (CLI) and netconf operations. The control-plane is based on a standard Linux distribution (debian 9) whereas the data-planes are implemented as separate Data Plane Development Kit (DPDK) processes. The communication between the control-plane and the data-plane is based on a shared memory and handled by a standard Multicore Communications API (MCAPI) library. Within this new architecture, the OneOS6 (including all the legacy routing function) could be run on:

(i) Standard x86 or ARM platforms

(ii) Virtual machine running of the top of legacy hypervisor such as KVM or VMWARE.

The current implementation of OneOS6 based on Linux and DPDK is resource hungry. It requires at least two Central Processing Unit (CPU) cores and 2 GB of Random Access Memory (RAM). This means that on a standard 8 cores machine we can only run three vRouters assuming that the hypervisor and the management plane require two CPUs. Ekinops is also developing an SDWAN solution for service providers. As shown in figure 3.12, the solution is basically made of three components:

(i) edge device: The SDWAN Edge is where the SDWAN tunnels are initiated or terminated. It creates and terminates secured (encrypted) tunnels over different types of wired or wireless underlay networks, such as broadband Internet (DSL, Fiber), LTE and Multiprotocol Label Switching (MPLS)

(ii) controller: The SDWAN Controller is responsible for the management of the Edge devices. This includes, but is not limited to, authentication, activation, underlay and overlay IP configuration, IP security (IPSec) key management and traffic policy distribution. The controller provides an open Netconf-based Application Programming Interface (API) on its northbound interface. It is natively managed through the Director but could be also managed by a third-party management system. From an architectural point of view, the controller is made of several micro-services called micro-controllers. Each micro-controller is responsible for a specific management function. This includes, but is not limited to, Route Reflector, Key Server, Bootstrap Server

(iii) Director: The director is a centralized web portal that offers a multitenant, multi-user and role-based User Interface (UI). It allows Service Providers, Partners and Customers to run and operate an SDWAN network. The Director can be managed by a third-party management system such as NSO or blue planet through its REST-based northbound interface. It manages the edge devices through the controller.

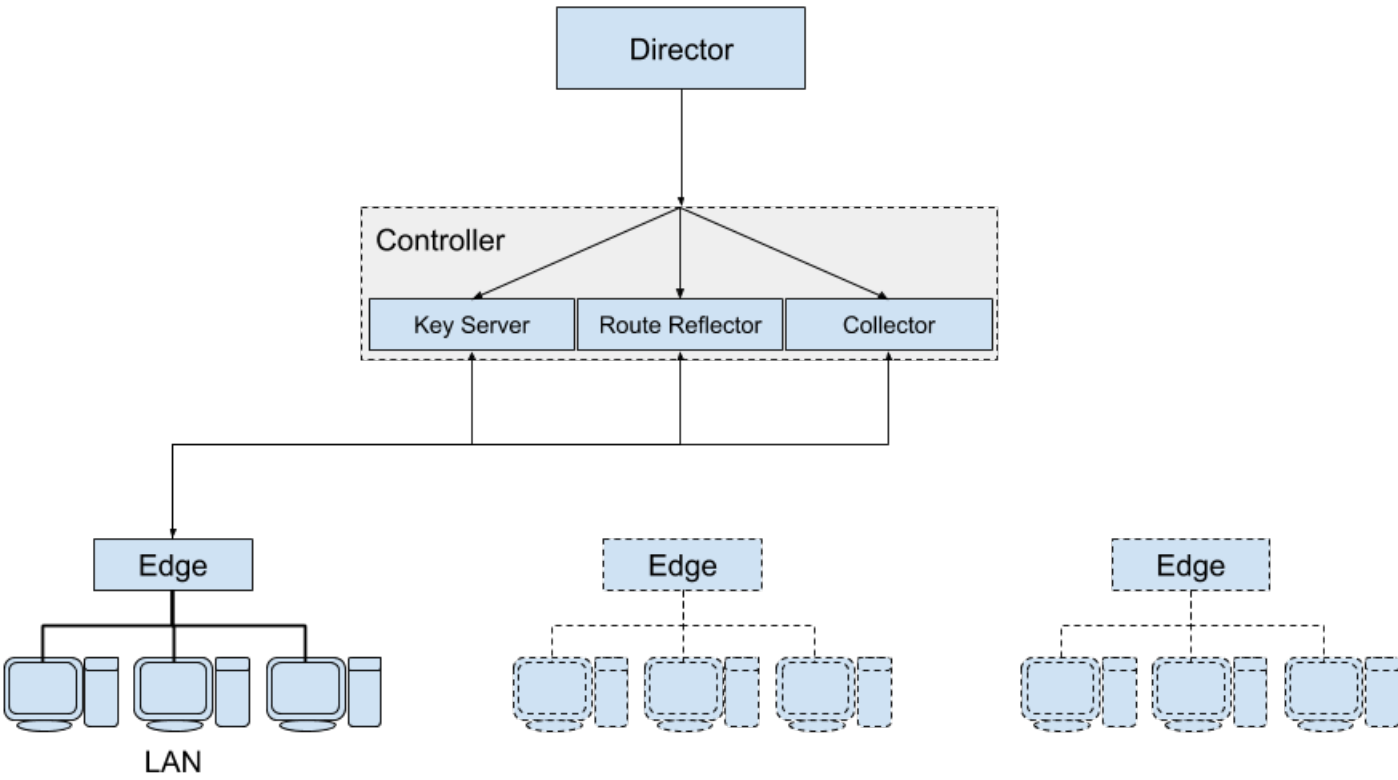

Figure 3.12: Ekinops SDWAN Components

3.2.3.1 Existing Architecture

3.2.3.1.1 OneOS6 Figure 3.13 illustrates the current architecture of OneOS6 on a physical platform. The software is basically made of three planes:

(i) The management plane

(ii) The control plane

(iii) The data plane

The hardware platform is managed by two different operating systems

(i) A linux OS handling the management and the Control plane.

(ii) A Performance Oriented Scheduler (POS) that handles the data plane.

The management plane is based on a Netconf server that provides a Netconf, command line and graphical interfaces. The control plane is made of several control daemons. The control plane daemons receive their

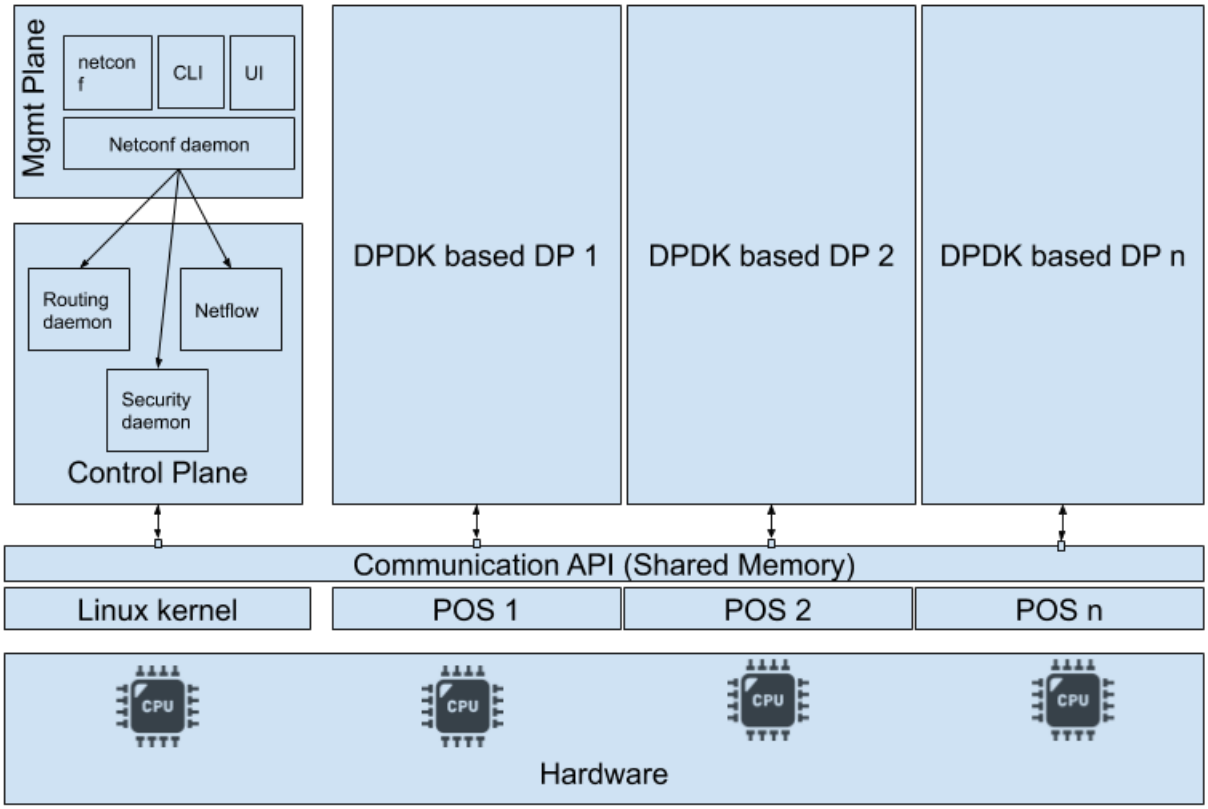

Figure 3.13: OneOS6 physical architecture

configuration from the netconf server through a Transmission Control Protocol (TCP) channel (i.e. socket) and configure the data plane processes through an MCAPI. The data plane is made of several processes. Each process is DPDK based and runs on a TOP of a POS and an isolated CPU.

For the virtualized version of the software, the design has been almost kept unchanged (Figure 3.14). The POS has been removed and replaced by the existing Linux OS. The Data plane processes runs as legacy linux threads pinned to isolated CPUs.

3.2.3.1.2 SDWAN As shown in figure 3.15, the very first version of the SDWAN controller was based on physical CPE (pCPE). Each pCPE is handling a subset of the controlling functions (e.g. NetFlow collector, routing, security, bootstrap...). A set of pCPEs is associated to a given customer. This solution provides a very good isolation and security levels however it suffers from a big scalability problem. To enhance the scalability and lower the cost the controller we rapidly moved toward a VNF based approach. We have simply replaced pCPEs with virtual CPE (pCPE)s, but even with such an architecture the scalability of the whole solution remains very basic. To overcome this problem, we are currently moving toward a container-based architecture so that each controlling function is implemented as a separate docker container. This solution provides the required level for the scalability but suffers from a serious isolation problem since all the Dockers run on the top of the same OS.

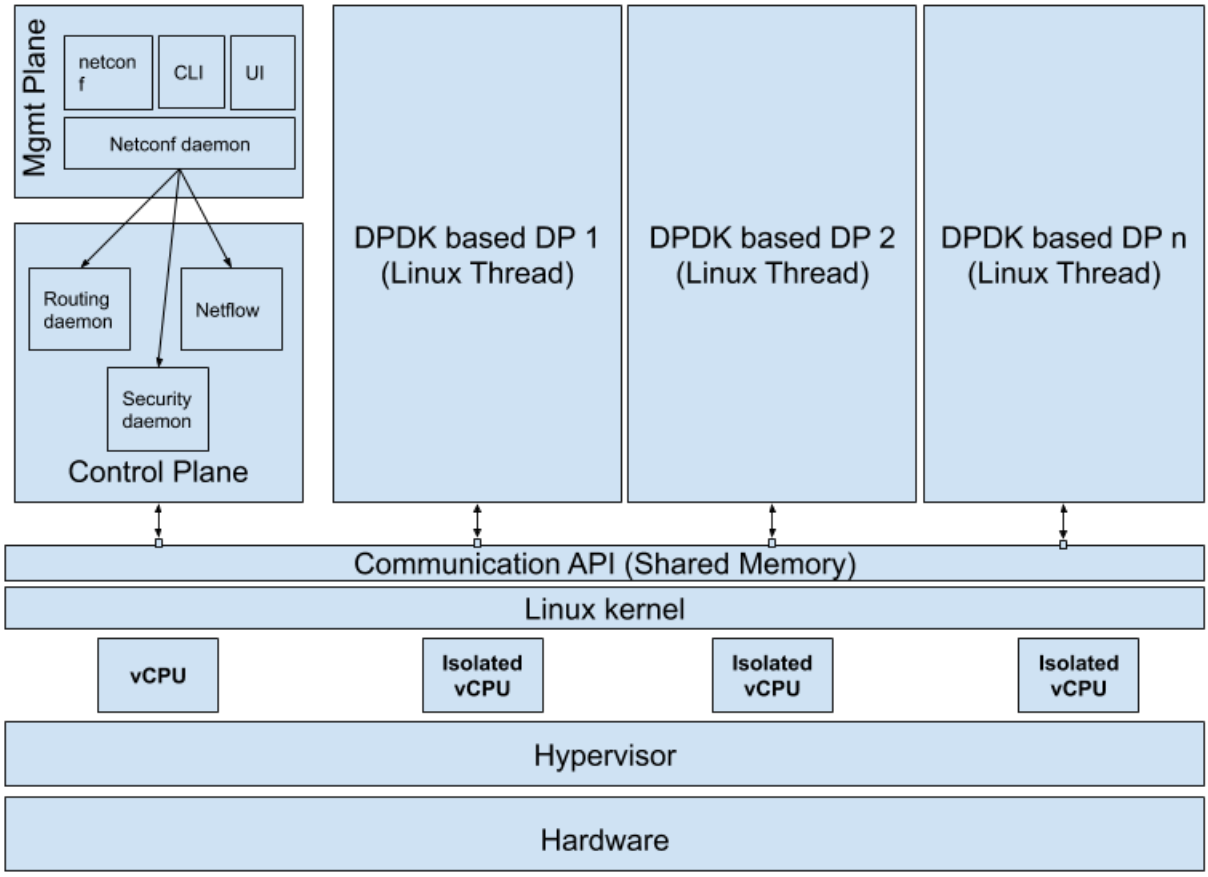

Figure 3.14: OneOS6 virtual architecture

3.2.3.2 Benefits of Using Unikernels

As a first use-case, we will adopt the Unikernel architecture to explode our OneOS6 software into smaller micro services. Each micro-service handles a very specific network function such as Network Address Translation (NAT), Dynamic Host Configuration Protocol (DHCP), Quality of Service (QoS), Routing etc. Obviously, the micro-services are running on unikernels. This allows to instantiate multiple VNFs on the same CPU core and increase the hardware efficiency by a factor of x10 and thus boosting the competitiveness of our products

The second use case consists in using unikernels to build a Group Domain of Interpretation (GDOI) key server for our SDWAN controller. This will enhance the performance and the boot time of the keyserver and provide a better isolation between the different key servers (one per customer). The Key server is responsible for authenticates group members and handles the distribution and renewal of encryption keys.

3.2.3.3 Requirements on Unikernels

We expect our unikernel-based services to run on the top of our pCPE (Linux based) on an X86 platform. Third party dependencies include:

(i) Dpdk

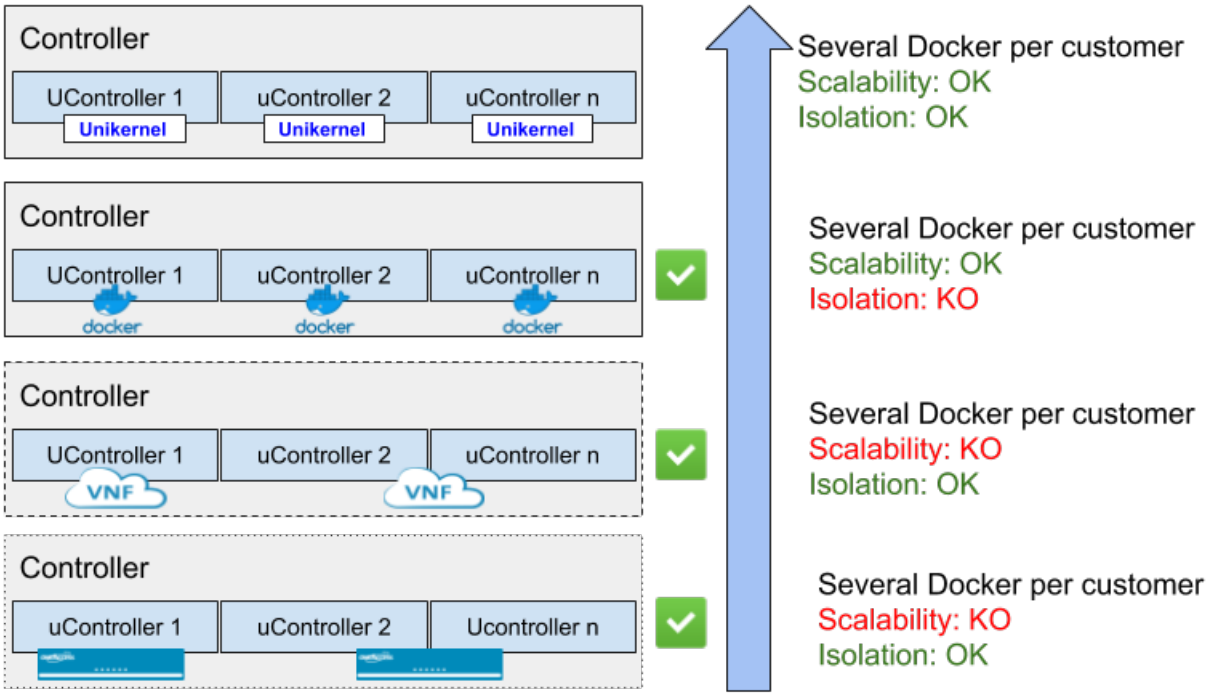

Figure 3.15: Controller Architecture Evolution

(ii) linux-vdso

(iii) Libpthread

(iv) Libdl

(v) libc

(vi) libfreeradius-client

(vii) Openssl

(viii) librt

Other requirements may be identified as the project and software evolve.

3.2.3.4 Performance Expectations

OneOS6 packet forwarding use case: 3 Gbps bidirerctional / 350 B UDP packets / 1 Core Skylake Intel CPU

SD-WAN use case: 1000 Edge devices per key server

3.2.3.5 Description of Proposed Trial

3.2.3.5.1 OneOS6 The target deployment scenario used to evaluate the performance of the UNICORE technology in the Virtualised OneOS6 dataplane is lab-based. It consist of a OneOS6 dataplane process connected to a TRex machine (https://trex-tgn.cisco.com/). TRex is an open source, low cost, stateful and stateless traffic generator fuelled by DPDK. It generates L4-7 traffic based on pre-processing and smart replay

of real traffic templates. TRex is used to measure the maximum sustainable throughput of a Device Under Test (DUT) under certain conditions. The DUT here is the OneOS6.
We configure TRex to replay the pcap described in figure 3.16. The traffic is basically User Datagram Protocol (UDP). The frame size is approximately 350 Bytes. With with pcap, a bi-directional throughput of 100 Mbps consists in 31250 connections per second and 31250 packets per second.

```
01:00:00.000000 IP 21.0.0.2.1030 > 22.0.0.12.56: UDP, length 341
01:00:00.020944 IP 22.0.0.12.56 > 21.0.0.2.1030: UDP, length 357
```

Figure 3.16: test traffic

For each test we measure the PLR, the latency and the throughput. The acceptance criteria are:

(i) Max Latency $<$ 1ms

(ii) Average Latency $<$ 0.3ms

(iii) Packet Loss Ratio (PLR) $< 10^{-5}$

3.3 Home Automation and Internet of Things

3.3.1 IoT Scenarios based on Symphony (Nextworks)

Symphony is the Smart Home and Smart Building Management platform by Nextworks which integrates home/building control functionalities, devices and heterogeneous sensing and actuation subsystems. The system allows creation of scenarios for the control of lighting, doors, climate in all the rooms, various automation sensors and actuators, entertainment devices (e.g. TV, VoD, music players) and phone calls specifically in residential environments.

Symphony can communicate with any automation controls, both standard protocols and proprietary systems. It integrates different protocols under a coordinated, unified management level with an open and modular approach.

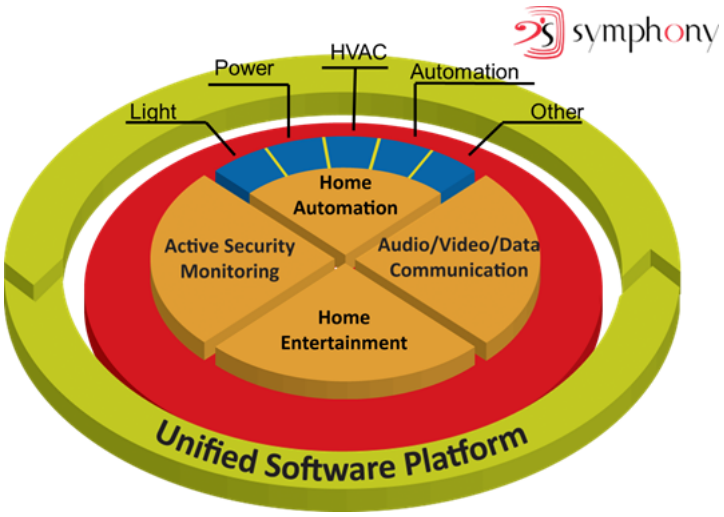

Figure 3.17: Symphony by Nextworks: the integrated Smart IoT platform concept

The Symphony Insight management station in the cloud allows operations, administration and management of the Building Management System (BMS) from any authorized remote terminal. It provides a scalable service architecture, data security and privacy, customized dashboards and business intelligence. To guarantee maximum confidentiality, BMS can be deployed on a private cloud infrastructure.

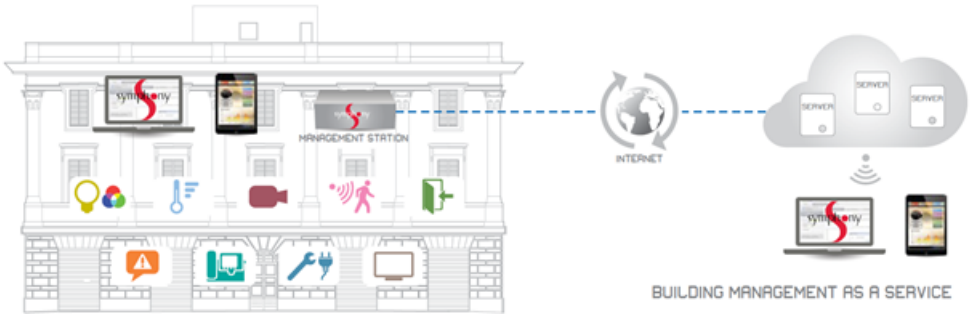

Figure 3.18: Symphony's Building Management as a Service concept

3.3.1.1 Existing Architecture

The Symphony platform consists of a series of modules which implements an IoT middleware capable of interfacing via specific protocol drivers to a series of domotic and automation field buses, and to model various automation technologies into a generalized abstract and unified model for control. Symphony's functions are currently deployed in various VMs and containers which communicate through a platform internal networking based on Layer 2 switching technologies and IPv4.

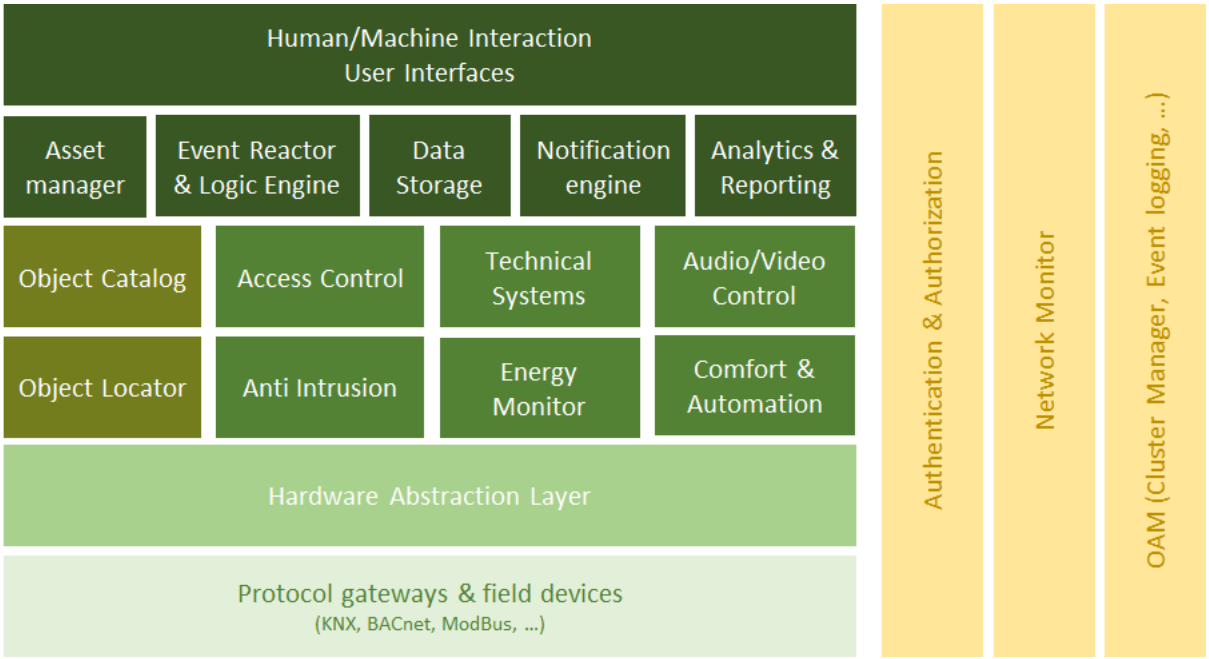

Figure 3.19: Symphony's high level architecture

Information data model. The Symphony data model covers most of the "objects" commonly found in a smart environment platform:

- Comfort related objects (e.g. RGB lights, dimmers, on/off switches, HVAC systems and fan coils, shades, curtains, lifts and motors)
- Environment sensors (presence, light, humidity, temperature, CO2).
- Energy measurement (energy meters, smart plugs, energy producers).
- Security related objects (anti-intrusion sensors, video cameras, audio monitors, access control).
- Media, player and adaptation devices (including audio/video indexes and metadata).
- User profiles for authentication and authorization actions.

These objects are stored in an Object Catalog function which is coupled with an Object Locator where exact references to communication mechanism and placement information are maintained. Each object has an abstract interface which is independent of the specific brand and model of the actual device being controlled.

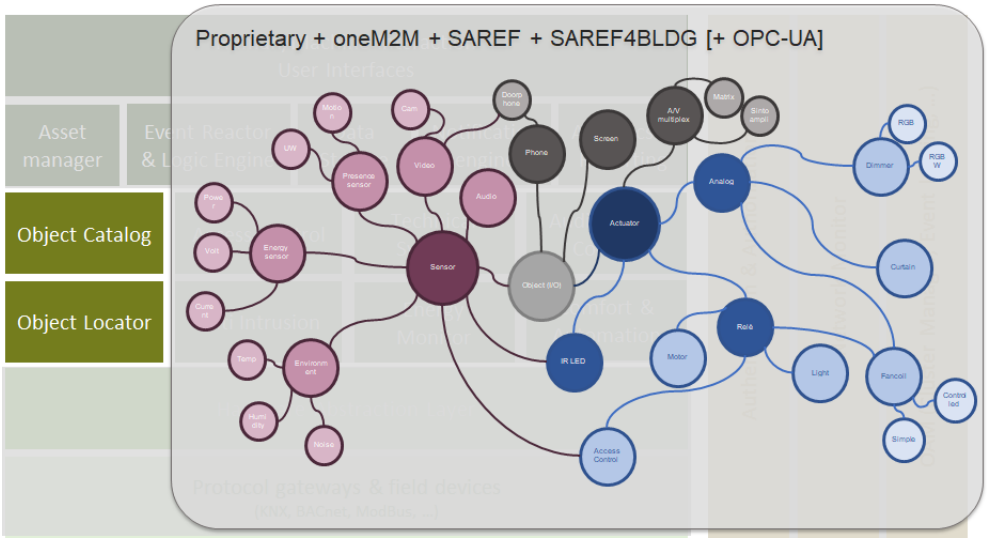

Figure 3.20: Symphony's Information Model

Hardware Abstraction Layer (HAL) and protocol gateways. The Symphony HAL device adapter is part of a hierarchical, scalable and redundant architecture providing object abstraction, data modelling and protocol translation for several automation-related protocols. The zone dispatcher is used to coordinate a number of device adapters, each one covering one or more areas. Device adapters implement the middleware data model and the translation functions for the supported protocols.

The supported protocols include Konnex, Digital Addressable Lighting Interface (DALI), Z-Wave, several proprietary protocols over ModBus/TCP, ModBus/RTU, serial connection, Simple Network Management Protocol (SNMP). The external interfaces of the module are string-based TCP protocol, REpresentational State Transfer (REST) and protocol buffers. Drivers for Zigbee, M-bus, BACnet, Cisco EnergyWise, OPC Unified Architecture (OPC-UA) are planned for future releases.

Audio/Video Control. The Audio/Video Control is an extension of the main device adapter module, focused on audio/video devices. It works as a room function processor, providing a gateway between Nextworks' middleware and devices such as televisions, monitors, video projectors, sinto-amplifiers, audio processors, satellite/cable receivers, set-top-boxes.

Anti-intrusion controller. The surveillance controller handles events and video streams generated by IP camera objects within the middleware. It supports streams distribution to terminals and to full-fledged control stations, which can perform Pan Tilt Zoom (PTZ) control and display multiple cameras by accessing a single, camera-independent interface. The supported devices include Open Network Video Interface Forum (ONVIF) cameras and proprietary cameras (e.g.: Mobotix,FLIR).

Communications controller. An Asterisk-based IP switchboard for voice and video communications. It supports any Session Initiation Protocol (SIP) phones (hard and soft), videophones, doorphones, over IP and analog channels. Besides handling incoming and outgoing calls, the controller can propagate notifications upon call-related events (call setup, tear-down and dismissal) and exposes an interface to the middleware to

generate calls.

Energy Monitor. The energy management module collects information from measurement sensors and performs actions based on behavioural policies aimed at energy saving. The module contains a complete representation of the energy distribution and measurement topology (which can seamlessly span from a single centralized measurement point, to a dense branched tree covering each single plug in the building).

Notification engine. The notification engine provides the middleware modules with an opaque asynchronous message bus, operating according to the publish/subscribe paradigm.

Data storage. The data storage module allows each application to save configuration, state parameters and historical data. It currently supports an Structured Query Language (SQL) back-end (to store static data) and an Round Robin Database (RRD) back-end (to log real-time data coming from sensors and events). The module is capable of implementing data collection, storage, queries and processing from 1M+ distributed sensors, retrieving data from multiple data sources (i.e. proprietary CORBA and text-based interface, Advanced Message Queuing Protocol (AMQP) and Message Queuing Telemetry Transport (MQTT) exchanges, etc.) which are then stored into a number of configurable data sinks (e.g. PostgreSQL DB; NOSQL backends like Cassandra, Elastic Search, InfluxDB, etc.; AMQP and MQTT exchanges). The stored data are then processed with basic analytics engines for aggregation, rate limiting and sub-sampling, with configurable data retention policies.

Event Reactor. The event reactor module allows implementers to write simple yet powerful recipes that associate one or more middleware events (i.e. notifications and module-specific events) to one or more actions (i.e. any operation performed through a CORBA IDL), possibly based on activation rules.

User interfaces. Graphical user interfaces provide programmable, flexible views to middleware objects. They represent the service to end user, but are usually decoupled from the actual service implementation.

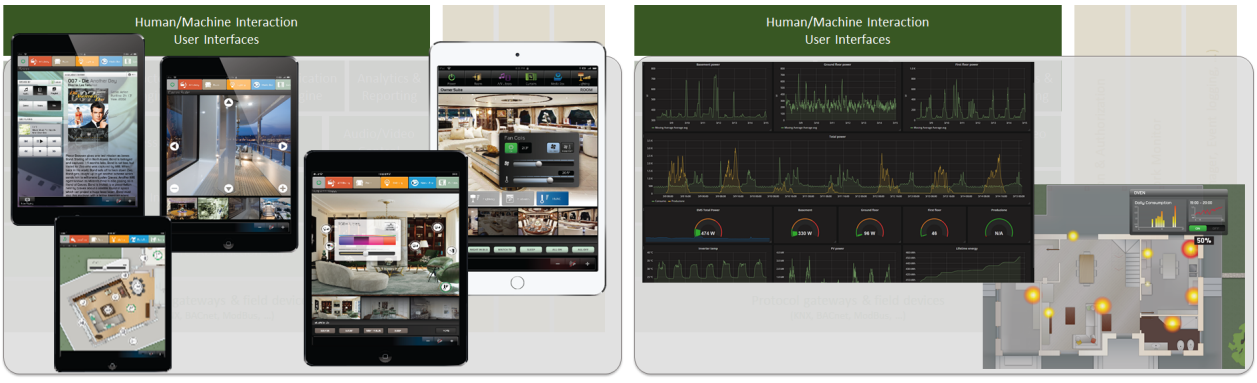

Figure 3.21: Symphony's User Interfaces

Available interfaces are:

- Web interface, based on HTML5 and running inside a browser.
- Native PC interface, running on Windows and Linux

- Tablet/smartphone application

Cloud services. The Symphony systems allow for remote management and proxying of the core functionalities through the activation of functional backend in Cloud from which it is possible to:

- Access Data Storage and Notifications from multiple distributed instances
- Implement Analytics and Reporting on multiple local instances
- Implement Event Reactor configurations and manage energy control policies
- Access local objects and interact with them (read/write, depending on user profile and roles)

The high level architecture of the Symphony cloud system is depicted in Figure 3.22.

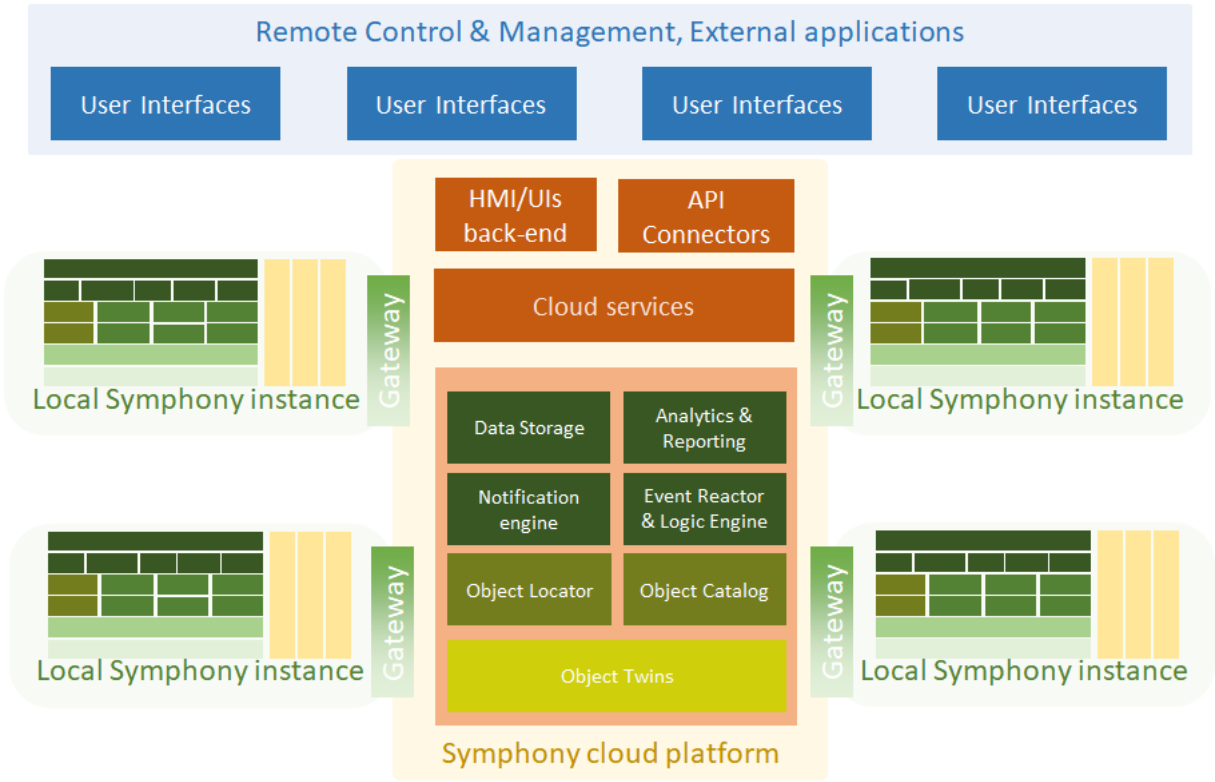

Figure 3.22: Symphony's high level architecture

3.3.1.2 Benefits of Using Unikernels

Nextworks is currently evolving the Symphony platform to further implement highly decomposed and distributed low service modules to be distributed out of the stand-alone system over a wide area (i.e. local cloud, public cloud), in a truly flexible IoT paradigm.

The next planned step for Symphony is to migrate to a micro-service architecture and become highly distributed on a variety of hosting systems (e.g. domestic NAS or micro servers hosted at home or at providers' curbs or in the cloud) and highly flexible to incorporate more and more technology drivers for sensor/actuators.

The migration to unikernels may be an interesting step in this direction in various specific areas, like

- functions for the automatic resource discovery and dynamic configuration of services;
- functions for data storage;
- specific domotic or automation protocol gateways (e.g. Zigbee, Z-Wave, Bluetooth LE);
- specific network functions (e.g. local routers for NAT/Firewall);
- specific media service gateways and/or voice/video communication handlers.

The envisaged scenarios for this use case are briefly introduced in the following.

- **Scenario 1.** A first stage of validation of functional aspects for a selection of specific unikernel-based functions of the ones specified above. In particular, in this stage the focus is on the validation of the process of automated compilation, building, packaging and deployment via Unikraft of a Symphony function for target deployment nodes (e.g. NUC, or Kubernetes cluster).
- **Scenario 2.** A second stage of use case evaluation will consider performance aspects. This is particularly relevant for the building management use case, where scalability and distribution of the functions across the platform are critical, as well as automated scale out/in procedures.
- **Scenario 3.** A final stage of experimentation will focus on automatic deployment of the Symphony Building Management System through distributed controller nodes in which unikernels are generated at run-time, taking into consideration characteristics, constraints and location of the available hardware nodes.

3.3.1.3 Requirements on Unikernels

We intend to deploy unikernel-based services in containers or light VMs to be orchestrated and managed via ProxMox (https://www.proxmox.com) or Kubernetes (https://kubernetes.io).
The target reference processor architecture for all the selected functions is x86, with the option to support

- libc
- sqlite3
- openssl
- libcurl
- libprotobuf

As requirements may evolve this list might not be exhaustive.

3.3.1.4 Performance Expectations

We expect to implement functions via unikernels which are capable to

- implement the same functionality across the current interfaces (at least string-based TCP protocol interfaces);
- support the same number of messages on AMQP and MQTT exchanges with respect to standard containers or VM solutions;
- support the same number of network flows in network functions (e.g. NAT translations, firewall rules, etc.) and packets processed per second in similar conditions of assigned resources;
- allow via unikraft automated packaging of unikernel functions with variable configuration profiles, to allow automatic on-demand spawning of unikernel-based functions for service scaling or event-based processing;
- lower resources consumption for the Symphony middleware to allow it to fit into small-scale computing elements (e.g. domestic NAS);
- lower time for delivering software upgrades in field installations to reduced footprint images (unikernel-based) and automated build procedures.

As performance requirements may evolve this list might not be exhaustive.

3.3.1.5 Description of Proposed Trial

The evaluation of the UNICORE solutions for Symphony (i.e. IoT middleware functions and gateways implemented with unikernels, and integration with Unikraft) will take place at Nextworks' premises in Pisa, where Symphony is currently deployed to implement the overall building automation and domotic control of the office.

The location does not need any specific preparation, apart from the preparation of the unikernel images to deploy and the deployment of the UNICORE toolchain in Symphony build servers.

The types of devices involved in the trial will include lamps, dimmers, RGB lights and curtains, all controlled via Symphony.

The possible installations inside the building are depicted in Figure 3.23:

It is planned to conduct this trial with a group of Nextworks researchers who will monitor the performance of the deployed unikernels in the overall end-to-end service chain.

The evaluation process will test the three scenarios described above with the primary goals of

- verifying consistency and continuity of functionality;
- verifying usability and flexibility of the Unikraft-based toolchain;

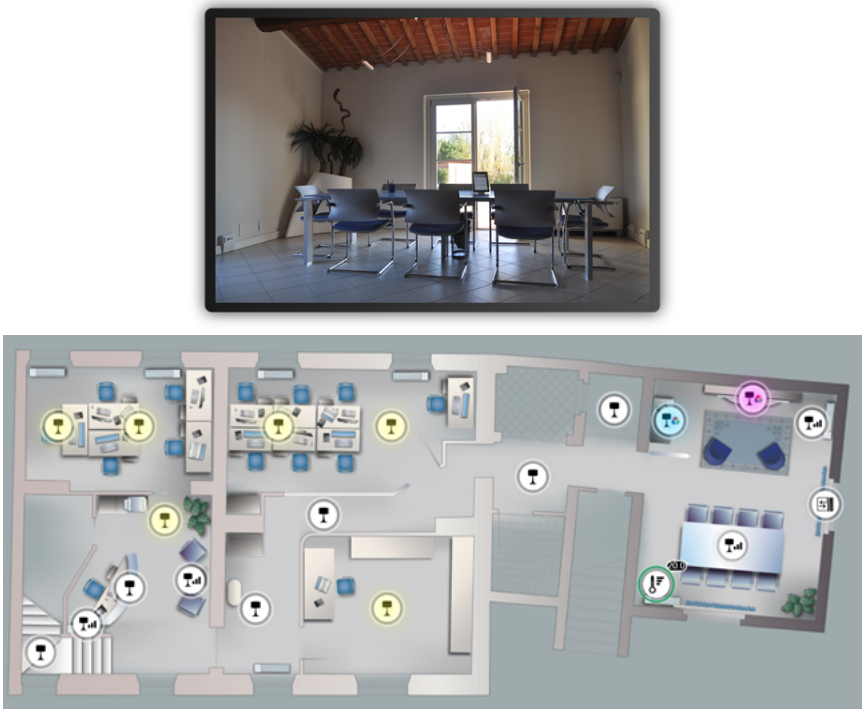

Figure 3.23: Nextworks trial for home automation use case: IoT devices at ground floor.

- evaluate performances of the various processes migrated into unikernel functions;
- evaluate resource consumption in the platform to quantify potential benefits with respect to standard container-based approaches;
- evaluate service reliability and mechanisms of warm-upgrade of process images.

3.4 Smart Contracts

3.4.1 Smart Contracts (EPFL)

Smart contracts are digital contracts between multiple parties that enable an irreversible execution of the contract without the need of a trusted third party. To provide decentralized trust, smart contracts are typically implemented on top of a blockchain implementation like Ethereum[1]. A blockchain is a distributed immutable ledger technology that is usually, but not always, used in cryptocurrency applications (e.g. Bitcoin[2]). The simplest representation of a blockchain is a chain of blocks where the links are immutable.

3.4.1.1 Existing Architecture

The DEDIS lab at EPFL has developed its own decentralized framework called Cothority, which stands for Collective Authority. It enables a set of decentralized and distributed authorities to collaborate towards a common goal, without placing trust in any of these individual authorities. Within this framework, the lab has developed a blockchain implementation, Byzcoin, where several peers (called conodes in the documentation) collaborate to reach consensus on the next block.

The Byzcoin implementation is a permissioned blockchain, which means that the blockchain conodes are known entities added by administrator. In contrast to permissioned blockchains, permission-less blockchains define their consensus participants through a Sybil-resistant concept, such as proof-of-work, proof-of-stake, proof-of-personhood, etc. Byzcoin reaches consensus through an optimized implementation of Practical Byzantine Fault Tolerance (PBFT) that scales to thousands of participants. Specifically, conodes run a collective signing protocol using a cryptographic algorithm that aggregates the signatures of all conodes into a single one and that proves that a threshold of honest peers have verified and agreed to append the block to the chain.

Byzcoin provides the API necessary to create transactions and to send them to a peer that will propagate the proposal to be verified and included in the next block if enough conodes agreed. In simple terms, a transaction contains a list of instructions and each of them contains the key of the value to update and enough arguments to update it accordingly. Each instruction also contains the contract identifier stored alongside. The conode can then look up the contract implementation and execute the instruction to move to the next step of the transaction. In comparison with Ethereum, where external developers can write their own contract and upload it to be available to clients, Cothority's smart contracts are pre-compiled and packed with the conode binary.

3.4.1.2 Benefits of Using Unikernels

The previous section introduced the current state of the Cothority to give a hint about the limitations. External partners cannot write their own contract on-the-fly because it must be packed with the conode binary, which means that they have to modify the conode software. It also means that future updates to a contract require a

[1] https://www.ethereum.org/
[2] https://bitcoin.org/en/

full Cothority update because a threshold of honest conodes need to know about the new version.

Ethereum allows anyone to write a smart contract and upload it so that any holder of Ether coins can use it. Doing so is, however, not so simple. First, because malicious developers and users alike can create and interact with contracts, Ethereum needs to contain the damage they can inflict. As an example, the execution of transaction must be sand-boxed so that it cannot alter another user's data. Second, the result of the transaction must be the same for every peer in order for the system to reach a consensus. The latter is especially hard to ensure even in the absence of malicious parties, when the distributed system is heterogeneous (i.e., nodes are running on different architectures such as x86, ARM, MIPS).

Given these issues, the Ethereum team with has developed an interpreted language that runs on the Ethereum Virtual Machine (EVM) and with limited features to further ensure deterministic execution. The drawback is that one cannot execute the smart contract with the native machine performance as the EVM interprets the code, but the language produces deterministic results by definition. For instance, there is no floating point support because of the unpredictability of the precision, but rather a fixed precision support that is more predictable. Through these mechanisms, Solidity solves the deterministic behavior required for consensus, but it is also necessary to sandbox the execution to prevent unauthorized actions or direct attack on the nodes, like a Denial Of Service (DOS) with a simple infinite loop in the code. For this, Ethereum introduces the concept of gas, where each instruction has an associated cost and a contract's execution stops when it runs out of gas.

Ideally, we want to replace the usage of a virtual machine by Unikernels that would let anyone write smart contracts using an existing language, supported by Unikernels (e.g., Rust, Go, C / C++). Each blockchain conode can then execute a smart contract inside a unikernel created for that purpose. The machine the conode is running on does not affect the result of the execution because the Unikernel itself ensures a deterministic execution.

The direct benefit of using Unikernels to increase the number of potential distributed applications/smart contracts that can be written, because one problem that developers are currently facing with Solidity is the limited computational power/available libraries that prevents them from writing complex cryptographic functions. Those functions are often needed in use cases that require security and/or privacy. The cryptographic domain is also evolving quickly when new attacks are discovered and then smart contracts need to be adapted to new standards. Unikernels can help in that direction because one can use open source libraries available from the largest community.

A second benefit is the increase in system robustness through heterogeneity. The more heterogeneous the system is, the more difficult and expensive it is to build an attack against it. Concretely, imagine that a failure in a hypervisor implementation leads to allowing a smart contract to make incorrect ledger modifications. If the unikernel-based smart contract is executed on other validators via diverse hypervisor implementations, then the flaw is collectively detected and the integrity of the ledger is preserved.

3.4.1.3 Requirements on Unikernels

The challenge is to provide a similar environment as the EVM but with a wider range of expressiveness. In other words, smart contracts should be written using the most appropriate language and libraries to build the distributed application. Thus, diverse executions of the same Unikernel should result in the same changes to the distributed ledger.

In summary, the important aspects required are:

- Execution of an intermediate or low-level representation of a smart contract
- A budget in terms of resources for a given execution
- Deterministic execution over any supported platform

A Smart Contract File (SCF) has to be verified for executions that are non-deterministic or infinite. The verification for non-determinism can be performed either at runtime, when the unikernel executes, or before the contract is stored in the distributed ledger and is, thus, publicly available. On the one hand, the former option is faster because it analyzes only parts of the program that are indeed executed, but it detects violations late, when the contract is already on the chain. On the other hand, the latter option saves resources as it updates the ledger only when the SCF is correct, but it is likely to incur a costly static analysis. The challenge is to determine a sweet spot between these two options suitable for SCFs.

In order to compete against native execution, it is important that execution depends mostly on the Unikernel runtime, so that the boot time is not part of the execution. It is then expected that the loading time of the SCF will be small enough so that the overall execution time will lie in the defined range.

3.4.1.4 Performance Expectations

There are two main references we will benchmark our performance against, which are the current native performance of the pre-compiled smart contracts written inside the Cothority and the EVM interpretation of the byte code. We expect that Unikernels have a performance close to the native execution and the of course better than that of the EVM interpreter, especially noticeable for complex operations.

One aspect of Unikernel that can significantly affect performance concerns floating-point numbers. Given these are frequent operation, if the approach used by Unikernels is much slower than a native execution, optimization are likely necessary, such as using the native Floating Point Unit (FPU) for a specific platform and switch to a software implementation for the others.

As mentioned previously, the verification of the SCF can be done either statically or during runtime. In the case of the later solution, it is important to reduce the overhead of that procedure to the maximum to stay in the range defined at the beginning of the section. Runtime checking is costly and this could negatively affect the execution time.

3.4.1.5 Description of Proposed Trial

There are multiple goals that will need to assert:

(i) Efficiency and truthfulness of the static/dynamic checker.

(ii) Deterministic execution of an SCF over different architectures/platforms/OS.

(iii) Performance of an execution against EVM.

(iv) Interactions with the blockchain.

The checker has to insure that a program does not allow an execution path that uses nondeterministic instruction, including but not limited to system calls that are known to be forbidden in the context of deterministic execution. In order to ensure this behavior, we will test this checker with different programs, both deterministic and nondeterministic, and make sure that it correctly returns errors when required. Depending on which approach is chosen for the development of the checker, i.e., white/black list, it is possible to write programs that will verify the behavior according to the set of authorized/unauthorized instructions.

As for the checker, we expect specific logic to be translated to different instructions depending on the architecture the unikernel is running on, so it is possible to create an SCF with known nondeterministic behavior (e.g., floating-point numbers) and ensure that the result is the same over different platforms like x86- or ARM-based machines.

For the next goal, it is necessary to write a smart contract on Cothority, Solidity and one of the supported languages. With this smart contract, it is then possible to compute benchmarks for the three different environments and to compare them to make sure that the unikernel execution at least lies between Cothority and Ethereum.

Finally and as a proof of concept, the integration of a unikernel-based execution of a smart contract to the framework developed by DEDIS will be tested through different use cases that cannot be performed efficiently on Ethereum. For instance, it would be interesting to demonstrate how external libraries can be integrated to a smart contract like cryptographic libraries.

4 Unikernel Core Technical Requirements

Having thoroughly described all of the UNICORE application domains in the last chapter, we now proceed to outline all of the technical requirements that are needed in order for UNICORE's core to meet them. We split the content into multiple sections, beginning with general and API requirements, followed by integration with major, existing orchestration frameworks, continuing with security and isolation requirements, and finishing with deterministic execution requirements crucial to the smart contracts use case.

4.1 General Requirements

The status quo of unikernels mostly focuses on time-consuming, manually built images that are specific not only to a single application but also to a specific platform (e.g., ClickOS, HalVM, Minicache, etc.). As such, they tend to target only a few requirements (e.g., performance) while ignoring many other important ones. In this section we provide a comprehensive list of such requirements derived from the UNICORE use cases but also wider experience from years of building unikernels and speaking with experts in the area. For ease of presentation we cover each of these in turn.

Functionality: First and foremost, the unikernel has to be able to execute the target application or set of applications. This functional requirement is not limited to just the application, but extends also to all of its dependencies. For instance, building a SQLite unikernel requires support for that software package, but also items such as a block device and file system (for permanent storage) and a network driver and networking stack (for receiving queries over the network).

Consequently, UNICORE should support a wide base of functionality common to a wide range of applications. Obvious candidates are standard C libraries (e.g., glibc, musl, etc.) and libcxx for supporting C++ files.

In addition, UNICORE should support a number of programming language environments. For example, supporting the Python interpreter would mean that UNICORE users could take their unmodified Python projects and have UNICORE's tools automatically build them into an efficient unikernel.

Finally, UNICORE should clearly support a number of application, especially those needed by the project's use cases. For example, for the NFV use cases, UNICORE should at least support one networking framework such as the Click modular router, a BPF-based implementation or a DPDK-based one.

Portability and Ease of Use: Beyond building a common code base, there will be cases for which such an approach would be too time consuming: certain applications would have too long a list of esoteric dependencies that would be a one-off, meaning that they would only ever be used for a particular application and would not, as a result, justify the porting effort. Further, we cannot assume that source code will be available for all applications, or that all applications will depend only on libraries for which we can have access to source code.

For such scenarios, the requirement would be to be able to run binary code, likely in the form of an Executable

and Linkable Format (ELF) file. The UNICORE system should be able to wrap such an ELF in a unikernel, be able to automatically load it, and meet any system call or dynamically-loaded library dependencies (these also provides as binaries).

A middle ground for when source code is available but a project is too complex to port would be to use cross-compilation techniques as used by Rump kernels; while strictly speaking not a requirement, the project will seek to provide such a mechanism as well.

Deployment Platforms: One of the major showstoppers of many previous unikernel projects was the fact that, at least initially, they were limited to a single platform (e.g., Solo5 for KVM, ClickOS for Xen). Adding another platform would require substantial amounts of work, work that would then be duplicated across unikernel projects.

To address this, UNICORE should provide the ability to transparently support a number of different platforms without the end user having to do any additional work. These platforms should not only include hypervisors such as Xen and KVM, but also container-based solutions such as Open Containers Initiative (OCI) and Docker.

Finally, to have further impact and deployment flexibility, time permitting UNICORE should aim to be able to support cloud-based platforms such as Amazon Web Services (AWS) and Google Cloud Platform.

Maintenance: The UNICORE common code base should be relatively easy to maintain: upgrading to newer micro-library versions should not be as time-consuming as the original porting effort. Upgrades to new minor versions should consist of minimum or no effort.

Performance: Given that one of unikernels' main features is specialization and the performance that is derived from that, UNICORE should not sacrifice these performance gains when achieving the previously mentioned requirements. KPIs such as fast boot times, low memory consumption, high consolidation and high requests per second should still be feasible.

4.2 API Requirements

In order to define common interfaces to support decomposition and modularization of OS components, and automated unikernel construction, it is necessary to define several API categories. Such categories will provide the common glue between compliant OS components, enabling the UNICORE tools to mix and match these components.

As stated previously, there exist several use-cases to consider. UNICORE shall handle these use-cases by providing a high-level interface to interact with low-level features independently of the underlying platform and architecture. To support the use-case requirements several API categories must be considered:

Networking: The network APIs provide entry points to protocols and re-usable software libraries. Such APIs allow to exchange information through the network and thus to support web servers, web databases, and many other web services.

Storage: The storage APIs are designed to provide compatibility to existing applications, and flexibility to

purpose built I/O intensive systems. They offer the file and directory abstraction to applications by leveraging interfaces exposed by the block device layer.

Memory management: Unikernels are single-address space. Therefore, there is no longer any separation between user and kernel address space. Such APIs should provide low-level (e.g., managing pages withing address space) and high-level operations (e.g., dynamic allocation of heap storage).

Scheduler/Process: The process or Scheduler APIs provide abstraction/API for all schedulers (e.g, cooperative round-robin scheduler, pre-emptive scheduler) and multi-task synchronization primitives.

Thread management: The thread APIs include two abstractions: thread management (e.g., creation of threads), and scheduling primitives for inter-thread synchronization and coordination.

Console: The console API is designed to provide compatibility to existing applications by providing an interface to log error and informative messages and to retrieve user input. This last one is especially useful for debugging purpose and should not be included to avoid common attacks.

Time management: For several kinds of applications, time and date management are useful. Therefore, it is necessary to define a group of functions to manage both. The time management APIs provide support for time acquisition, conversion between date formats and formatted output to strings.

System & Miscellaneous: Besides managing host resources, system APIs contain miscellaneous features such as exception handling and random number generation.

According to these categories, UNICORE will decompose operating system primitives and libraries into fine-grained modules called μ-libs. These can be arbitrarily small or as large as standard libraries like libc. Large micro-libs that give a higher-level API use other lower-level μ-libs, and also support a decomposition level. E.g., a libc could use several different variants of μ-libs for threading or storage. Three sub-categories should be defined:

Internal Libraries: Provide functionality typically found in operating systems and are part of the UNICORE core.

External Libraries: Consist of existing software projects external to UNICORE. For example, these include libraries such as openssh and libuuid, but also language environments such as Javascript/v8 and Python

Platforms libraries: Provide bits and pieces necessary to run unikernels on the target platforms. For example, platforms are Linux userspace, KVM and XEN.

4.3 Orchestration Environment Integration Requirements

For UNICORE to be successful, it is crucial that it integrates seamlessly with at least one major orchestration framework. While many candidates exist, potential targets are Docker, because of its wide deployment base and easy-to-use ecosystem and Kubernetes. Supporting such orchestration frameworks should allow existing users to painlessly adopt UNICORE unikernels.

4.4 Security and Isolation Requirements

For many years, unikernels have been touted as solutions to a number of security problems, and have been of particular interest to security-minded domains of application. Part of the reason is certainly justified: unikernels have a number of fundamental properties that make them ideally suited to such environments, and particularly resistent to attack:

- **Specialized code base**: The higher number of lines of code, the greater the attack surface and Common Vulnerabilities and Exposures (CVE) related to such code. Unikernels defend against this problem through extreme specialization, providing only the parts that are strictly necessary for the target application to run, and nothing more. Static compilation and techniques such as dead code elimination further help to keep the final image as lean (and secure) as possible.

- **No shell**: A large number of applications do not need a shell to actually run. Unikernels forego this common attack channel by simply not including a shell in the final image where not needed.

- **No system calls**: Another common vector of attack are well-knowns Application Binary Interface (ABI) and system calls. Unikernels provide a custom operating system and ABI, removing this vector of attack.

- **Immutability**: For the most part, unikernels do not include the ability to modify/reconfigure a running instance, preventing yet another common form of attack.

Beyond the above security requirements, which come almost for free, unikernels in the past have largely ignored a number of commonplace, yet fundamental, security features. UNICORE should at least attempt to address them:

- **Address Space Layout Randomization**: to protect against memory corruption exploits, Address Space Layout Randomisation (ASLR) randomizes the position of key data areas such as the positions of the stack, heap and libraries.

- **Stack Overflow Protection** through the use of stack canaries.

- **Entropy**: The UNICORE base should provide a good source of entropy.

Having the above mechanisms in place would mean that UNICORE could potentially be the first unikernel project/software to be able to truely generate unikernel images with security in mind.
Finally, regarding isolation, UNICORE should be able to provide strong isolation through the support of hypervisor-based platforms and virtual machines. In addition, for deployments where virtualization is deemed too heavy-weight, or the underlying hardware does not support virtualization extensions, UNICORE should support lighter-weight isolation mechanisms such as containers.

4.5 Deterministic Execution Requirements

UNICORE is to provide support for running smart contracts in a blockchain infrastructure. Smart contracts are executable pieces of software requiring consensus in a distributed environment. Each smart contract will run as a unikernel instance. Reaching consensus requires deterministic execution of a smart contract, i.e. execution that yields the same results irrespective of the hardware platform and environment it is running on. UNICORE aims to provide support for writing smart contracts in higher level languages (such as Go or C/C++).

A smart contract running as part of unikernel will not be allowed to use CPU instructions that result in non-deterministic behavior (such as floating point operations), instructions that behave differently on distinct architectures (such as ARM or x86) or actions connected to timing and scheduling, i.e. waiting/pausing and multithreading are disabled.

A unikernel instance will comprise of the smart contract code and core libraries. The smart contract code and library code invoked are required to be deterministic as defined above. UNICORE will provide a validator that vets smart contract code for deterministic execution. UNICORE will provide the core libraries as a package; the library API will be provided with annotation regarding deterministic execution. Smart contract unikernels will be created and run from core libraries and vetted smart contract code.

The smart contract developer will be provided a simple interface where the smart contract code is loaded and validated. A valid smart contract will be then fed to the blockchain infrastructure that will tie the UNICORE core libs and smart contract code and then run the smart contract in a unikernel.

5 Unicore Toolchain Technical Requirements.

This section will contain technical requirements for the Unikernel toolchain. The requirements are derived from the deployment scenarios and use cases, together with requirements elicited from other sources or by other means.

5.1 Overall Toolchain Requirements

The main objective of the UNICORE project is to provide a toolchain to automatically build images of operating systems targeting application(s) that are optimized to run on bare metal or as virtual machines. This toolchain includes the following subcomponents:

(i) **Decomposition tool**: Help developers to break down existing monolithic software into smaller components.

(ii) **Dependency Analysis tool**: Analyse existing applications to determine which set of libraries and OS primitives are absolutely required by the application.

(iii) **Automatic build tool**: Match the requirements computed by the dependency analysis tool to automatically build a unikernel.

(iv) **Verification tool tool**: Ensure that the functionality of the resulting, specialized application matches that of the application running on a standard OS.

(v) **Performance tool**: Analyse the running specialized application and gather detailed information to generate even more optimized images.

The requirements of each tool are explained in the following sections. In general, the toolchain has been designed to run on a specific platform. Indeed, a UNIX system is needed by the toolchain. In future versions, another platform(s) can also be considered to support the toolchain. Concerning the underlying architecture, the toolchain requires x86 (32-bits/64-bits) or ARM (32-bits/64-bits) architectures.

5.2 Decomposition Tool Requirements

The decomposition tool will be used to break down monolithic libraries such as libc and operating system primitives (e.g., memory allocators, network stack, ...) into a set of small modules that can be selected from a libraries pool to build unikernels. The tool will help developers in decomposing the software, and is targeted at the UNICORE consortium and not the software community at large.
The decomposition tool is still in research phase therefore only some assumptions are established. These are the following ones:

(i) The tool should perform incremental decomposition isolating one kernel subcomponent at a time;

(ii) When kernel subsystems have been isolated, patterns matching /recognition techniques can be used to extract relevant files and blocks of code;

(iii) These components are then integrated with each other where unknown functions and symbols are replaced by stubs.

The first objective of the tool is thus to help experts to understand the interactions between different components and to obtain a first skeleton of a micro-library. After this automatic extraction, developers will have to work on their own by implementing and verifying all stub functions to have fully functional modules.
Concerning the requirements, this tool should be able to retrieve packages and repositories from the Internet and then extract relevant subcomponent(s). The extraction part should not require specific requirements. Indeed, it consists of patterns recognition/extraction methods. At first, a Python/bash script can be designed to create a first functional prototype. After receiving feedback, the tool can be redesigned with different languages and technologies.
Another approach is to use tools like Clang Static Analyzer which would enable an analysis of the flow code and automatically extract the identified subsets.

5.3 Dependency Analysis Tool Requirements

A first prototype of the dependency analysis tool has been developed. The prototype is a console tool that gathers system calls, library calls and shared libraries (dependencies) from a binary application. The tool, written in golang, requires the following third-party dependencies:

- argparse
- psutil
- gographviz
- color

As the tool is written in golang, a golang runtime is required. The desired runtime is golang1.12.7 nevertheless lower versions ($\geq$ 1.10) should be compatible. Furthermore, several basic command tools such as `nm`, `ldd`, `lsof` and `readelf` should be available on the underlying host system. In addition, the tool ebables the visualisation of dependencies by showing graphs. The only requirement is to install graphviz on the host machine. Finally, as the tool analyses binaries, it is necessary to have the binary of the application.

5.4 Automatic Build Tool Requirements

The automatic build tool will automatically build an OS specialized image from the outputs of the dependency analysis tool. UNICORE will develop a build tool that relies upon the existing autoconfigure/automake combination to build binaries and the existing package management tools such as apt on Ubuntu or yum on Redhat. The following packages are required:

- make
- gcc
- g++
- libncurses-dev

Host systems should also contain various basic utilities tools such as `sed` and `awk` to filter output. In a general way, the automatic tool should use traditional components in order to be easily ported to different UNIX distributions.

5.5 Verification Tool Requirements

The verification tool will ensure the correctness and security of unikernels. For example, the tool will ensure that the newly built application is equivalent to the initial one. In that case, heuristic methods will be used in order to check if the old and new application behave the same. The tool will also ensures security in ring 0 and if micro-libs are correctly implemented.
At this date, the tool has still not been developed, only the prerequisites have been analysed. Among these, the following assumptions can be made:

(i) As for the dependency analysis tool, a Python script can be used initially to develop a functional prototype; The script will run two different images of a same application: one as a unikernel and the other as a traditional application. It will then provide various inputs to both and analyse their behaviour and respective outputs. Then, a matching score will be computed in order to ensure the correctness.

(ii) Even if the built application has the same behaviour as the original one, there can still remain bugs (e.g., buffer overflows). To protect against such attacks, UNICORE will use privileged processor instructions to implement highly efficient sandboxing mechanisms to contain possible attacks.

Since the UNICORE project has been designed for different platforms and architectures, the host system should be able to execute and verify a unikernel on qemu-kvm or/and XEN. In the same way, it would be optimal to be able to test the resulting unikernel on several architectures (e.g., ARM, x86, MIPS). Depending on the resources available, it may be appropriate for the toolchain to be hosted on remote servers in order to test all possible platforms and architectures.

5.6 Performance Optimisation Tool Requirements

The last component of the toolchain concerns the performance optimisation tool. This tool aims to help developers automatically improve the performance of their application for a given target platform and workload. When building a unikernel, the developer (or the build system) has to choose several parameters that can influence application performance:

- Which μ-libs to use, when several μ-libs are available for a given functionality. E.g., would the application be more efficient which a preemptive scheduling μ-lib, or a cooperative threading one?
- Configuration parameters of the μ-libs and the application. E.g. The performance of a packet-processing application will vary with the batch size (i.e how many packets are fetched at once).
- Resource allocation parameters, e.g., how much RAM to allocate to the unikernel?

The proposed approach consists in asking the developer to identify test cases (how to run the application, with which workload) and performance metrics (how to measure and quantify performance), and let the system derive a good set of μ-libs and parameters as automatically as possible, through trial and error.
Given an initial set of parameters (including which μ-libs to use), the performance optimisation tool would cycle through the following loop:

(i) Build a unikernel for target platform with the current parameter set.

(ii) Run that unikernel with the test workload, measuring performance.

(iii) Update the application performance model.

(iv) Choose a new parameter set to try next.

There could be multiple exit criteria, e.g., a parameter set that reaches a target performance, a parameter set which is expected to be close to the optimum, or exceeding a time/energy budget for optimisation. On exit, the tool would output the *best* parameter set found so far, and the corresponding unikernel.
The tool is still in research phase, and it remains to be seen how efficient we can make such an automatic optimisation. The tool will use machine-learning technique to build a performance model. To minimize the number of experiments required, we will try to make use of *active learning* (and maybe *bayesian optimisation*).
The tool might also make use of profiling tools, either to guide the automatic optimisation, or to help the developer optimise the application manually.
The requirements for the tool will thus consist of machine-learning frameworks such as Scikit-Learn, TensorFlow or PyTorch and profiling tools such as gprof. As machine learning can require a lot of computational power, a suitable hardware is required. A good Graphics Processing Unit (GPU) might be particularly useful to accelerate model training.
Once individual components (such as μ-libs) and a few applications will have been profiled and modeled, it might be possible to reuse gathered information to accelerate the performance modeling of new applications (e.g., via *transfer learning*). However, this is still an open research question at this stage.

6 Conclusions

In this deliverable we presented a large survey and analysis of all the potential features, both functional and in terms of performance, that the different use cases and business verticals might need from the underlying UNICORE technologies. The aim of the exercise is fundamental to the success of the project, since it establishes a sound set of requirements for the technical work packages of the project to meet to ensure that the use cases can not only be implemented, but also deployed at the technology readiness level promised. Beyond this, the requirements are there to maximize innovation impact, one of the main goals of the project.

Towards this goal, we have gathered requirements in a number of ways. First and foremost, we have looked at the project's use cases: Serverless computing, NFV, vRAN, and Home Automation and IoT and Smart Contracts. For each, we have not only identified what the crucial value added would be for these sectors if unikernel technologies were leveraged for them, but also derived a set of functional requirements (often in terms of software dependencies) along with a description of performance KPIs that would be needed to achieve that value added. For instance, for the NFV case, we have identified not only the need for a packet processing framework (e.g., Click or DPDK), but also performance metrics (e.g., 3Gb/s peak throughput per server).

In addition, we have derived a set of "platform" requirements, which, if met, would make UNICORE an enticing platform not only for these use cases, but also to establish it as **the** unikernel technology to be used when trying to develop business solutions in the European market. Examples of these include support for multiple virtualization as well as container technologies (e.g., Docker, KVM, Xen, etc.) in order to be applicable to the largest possible range of sectors; integration with existing orchestration frameworks and ecosystems such as Docker and Kubernetes so that adopters can seamlessly profit from UNICORE's technologies without having to make major changes to their infrastructure; and the set of automated tools (for dependency analysis, for automated performance optimization, etc.) that will make the creation of the unikernels and the solutions around them (and thus the user experience) more transparent.