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 April 2019

Actual submission date: 30 April 2019

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

Abstract

This is the milestone 2 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.

The milestone 3 version of this document will add: descriptions of use cases for the UNICORE tools; technical requirements for the Unikernel core (the common code-base); technical requirements for the UNICORE toolchain; conclusions.

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
Project Co-ordinator	Joel Nider, IBM / Emil Slusanschi, UPB
Technical Manager	Felipe Huici, NEC
Copyright notice	© 2019 Participants in project UNICORE

Executive Summary

This is the milestone 2 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 to first describe the application domains, how software in those domains is currently deployed and to identify the high level requirements that would need to be met in order to deploy them using unikernels. Then the target deployments that will be used to evaluate the Unicore technology are described, together with an indication as to how the success of each deployment will be measured. In the next phase, these application domains and target deployments will be analysed to identify detailed requirements on the UNICORE Core and Toolstack.

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.

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.

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 BNFs 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.

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 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 and reliability are expected.

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.

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.

In the Serverless Computing domain, CSUC propose several target deployment scenarios. 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.

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.

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.

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 intended that these will also be migrated to unikernels as the project progresses. The dRAX components will run on a relatively low cost Intel i7-based platform, 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.

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.

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.

The milestone 3 version of this document will add a chapter describing how the tools within the UNICORE Toolstack will be used, and what their inputs and outputs will be. It will also add chapters specifying the technical requirements for the UNICORE Core (the common unikernel code-base) the UNICORE Toolstack.

Contents

List of Figures

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.

Chapter 4 describes the target deployment scenarios (trials) that will be used to evaluate the UNICORE Core and the UNICORE Toolstack. There is one section for each application domain and a sub-section for each trial within that application domain. This chapter includes the business requirements and the functional and non-functional requirements for each trial.

The content of chapters 5, 6, 7 and 8 will be delivered at milestone 3 of the project.

Chapter 5 identifies and describes use cases for the tools within the toolstack. There is one section for each tool, and a sub-section for each use case for that tool.

Chapter 6 specifies the technical requirements for the UNICORE Core, organised into several categories.

Chapter 7 specifies the technical requirements for the UNICORE Toolstack. There is one section for each tool.

Chapter 8 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 focus is on the requirements arising from the application domains that have already 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 desirable or nice to have, rather than essential.

The second step is for the partners to describe, in chapter 4, 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.

This is as far as we take it in the milestone 2 version of this document. The next stage will be for the project participants to analyse the application domains and target deployments, and also to define use cases for each tool within the Toolstack. Analysis of the application domains and target deployments will identify requirements on the Unicore Core, and potentially on the Toolstack as well. Analysis of the use cases will define the the inputs that each tool must accept and the outputs that must result. From the discussions and use cases the participants will tease out specific requirements for both the Core and the Toolstack. The requirements will be prioritised using the MoSCoW method. In other words, the partners will need to agree which requirements are essential to be met (Must), which are highly desirable or desirable (Should), and which are nice to have (Could). Requirements that are common to more than one target deployment are likely to be given higher priority. The participants may also decide that some requirements are not feasible, or not worth implementing in the current project (Won't).

3 Application Domains

3.1 Serverless Computing

3.1.1 Digital Content Deployment Scenarios

CSUC has worked for years hosting, developing and implementing different digital repositories focused on digital content for the University Community, in concrete this kind of repositories is called institutional repositories.

Each institutional repository stores their 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 hibrid 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-

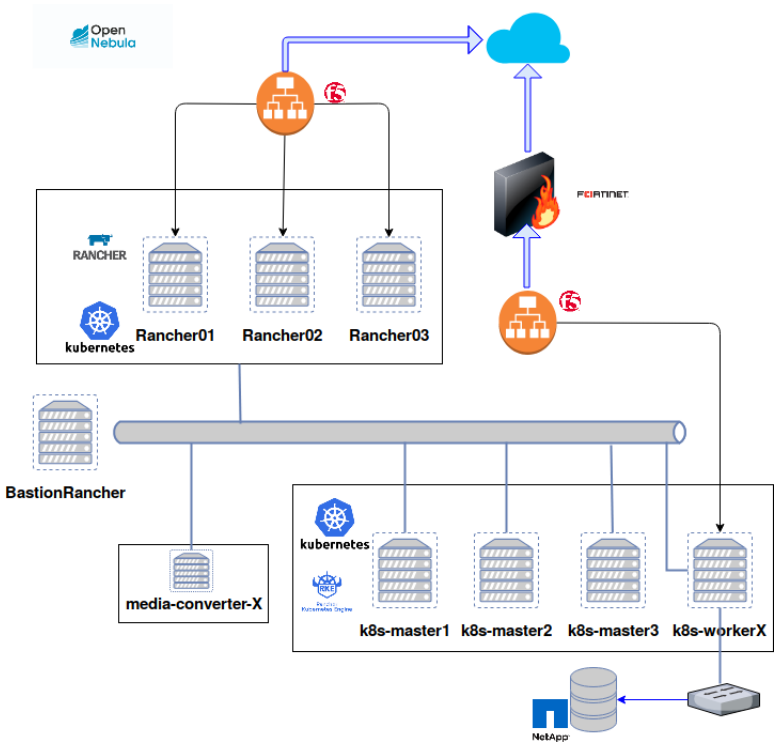

Figure 3.1: CSUC Deployment Scenario

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 How unicore could apply and improve in our ecosystem?

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.

3.1.2 Lambda Packet Processing Deployment Scenarios

3.1.2.1 Lambda function API for packet processing

Serverless services are growing rapidly (28% per year forecasted 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, they pay for the amount of work what was completed, and 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.

3.1.2.2 PacketCloud description and technical requirements

Our target scenario is to offer clients the ability to deploy per-packet network functions running in the ISPs cloud.

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.

3.1.2.3 Current technical landscape overview

To understand better the technical challenges, let us consider the current state of affairs in the field of NFV(Network Function Virtualization).
The de facto technical implementation of this scenario is to:

- run every function within a single VM, to provide security and isolation
- within the VM employ one of the available kernel bypass frameworks such as the industry standard DPDK, or other(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 3.2 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.

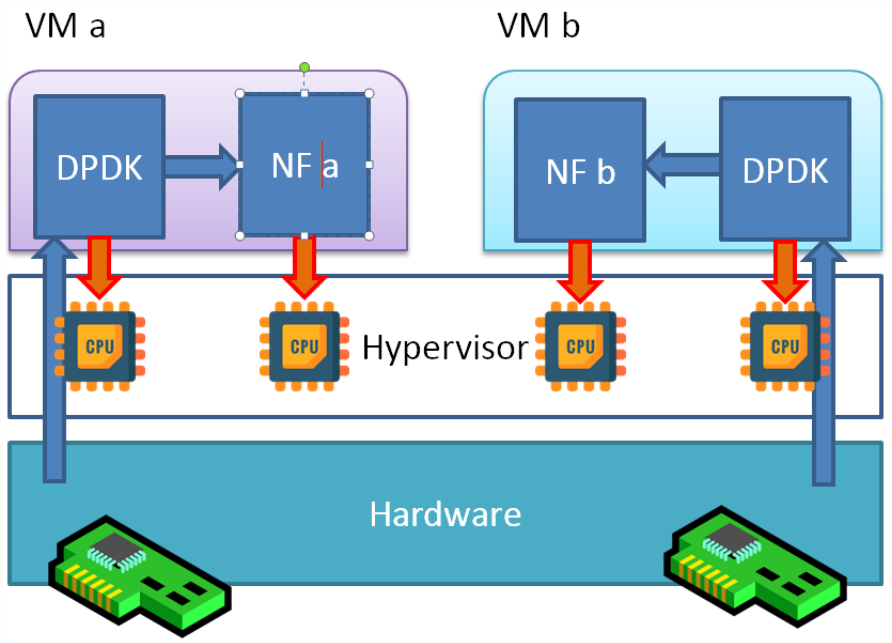

Figure 3.2: DPDK-based lambda service

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

- VM size:
  - (i) Kernel tailored for its NF instance - possibly unikernels.
  - (ii) Minimal runtime environment (tools and libraries) - **Decomposition Tool, Dependency Analysis Tool, Automatic Build Tool**.
- 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.

3.2 Network Function Virtualization

3.2.1 Broadband Network Gateway Scenarios

Orange's target for UNICORE work is determined by the definition of a novel framework for BNG approach, to set metrics of functional perspective for the new implementation model, comparing three different implementations, as an evolution from physical monolithic approach to virtualized monolithic implementation and further to a fully unikernel VM with each session running in a separate unikernel VM.
There are expected to be defined several work streams for the BNG use case:

- **Use case workflow**, including design, preparation, execution, system and use case monitoring, analysis and validation for BNGs as unikernels VMs, KPIs evaluation
- **Implementation workflow**, including activities related to virtualized infrastructure implementation, unicore lightweight VM as network functions for every single (v)CPE

- **Transformation workflow**, technical evaluating of (1) the single physical (PNF) BNG that expose a single network service function for several (v)CPEs; (2) the single virtualized (VNF) BNG that expose a single network service function for several (v)CPEs
- **Outcome workflow**, evaluating the unikernel BNG as VNF/NFV implementation, compared with the others possible scenarios, monolithic NFV and 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 developed technology should enabled seamless creation and deployment of any unikernel Unicore application, performance, scalability, security, isolation, efficiency.

The entire system is based on an efficient network function virtualization, 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. The entire system will run on dedicated lightweight VMs, instantiated in an automatic manner and orchestrated, per customer application.

Today, Orange's BNG implementation is based on several physical BNGs (Nokia 7750 hardware), deployed in different locations in the network, in Orange Data Centers. The 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 3.3.

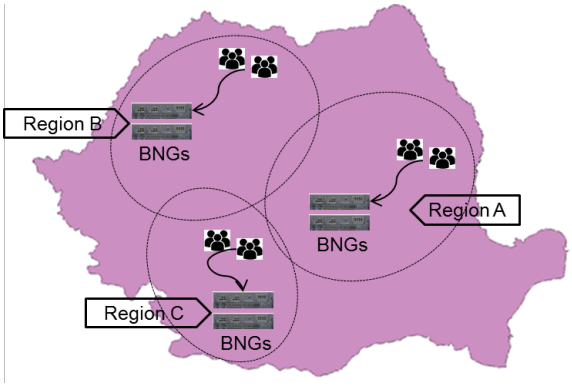

Figure 3.3: BNG Regions

(1) As a first step of today's implementation, in each site location there is deployed a resilient system, two active/standby BNGs being deployed for HA capabilities. The pair of BNGs supports all traffic from the region, for multiple services (Internet, VoIP, etc). The network provides to the customers connectivity for different apps, capacity up to 1Gbps, QoS and users monitoring. The entire system is designed to cope with an estimated traffic and number of customers, with a specific authentication access and service allocation resources. For clarity, the deployment of a region, optimal resource consumption and the time-to-market for the services in this monolithic approach can take up to several months, regardless the number of customers and the service requirements and needs.

(2) The second step of implementation consists in replacing the physical BNGs (PNFs) with virtualized platform in an NFV/VNF scenario, inside the service provider Data Centers, on a virtualized infrastructure. The approach of replacing the PNFs with VNFs is targeting to improve the BNGs deployment time, as a monolith application for services, unfeasible resource deployment for customers connectivity. Excepting the VNFs approach implementation, automation and orchestration capabilities, from the service perspective the approach is the same as (1). There are also several questions regarding the VMs' performance in case of a big number of customers, consuming resources, inefficient and poor security.

3.2.2 5G vRAN NFV Clusters Deployment Scenarios

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 ODM hardware. This is exemplified by the E1000 series of small cell products. The E1000 provides a single, low-powered 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 Network Function Virtualisation (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 3.4.

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

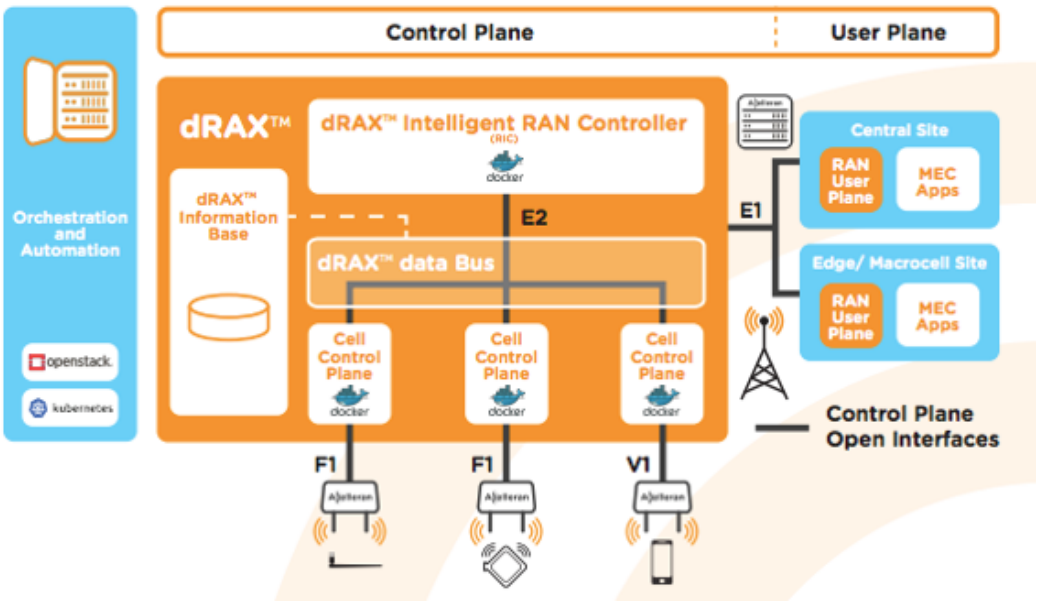

Figure 3.4: Outline dRAX Architecture

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.

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 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 key performance indicators will depend on which dRAX components are migrated to unikernels.

3.3 Home Automation And Internet of Things

3.3.1 IoT Scenarios based on Symphony by Nextworks

Symphony is the Nextworks Smart Home and Smart Buiding Management platform which integrates home/building control functionalities, devices and heterogeneous sensing and actuation subsystems. The

system allows the creation of scenarios for the control of lighting and curtains, the control of the climate in all the rooms, and the management of TV and phone calls.

Symphony integrates the management of all the monitoring systems of both internal and external areas, through a customizable graphical console.

Symphony can communicate with any automation controls, both standard protocols and proprietary systems. It allows the monitoring and control of diverse building automation systems, by integrating different protocols under a coordinated, unified management level with an open and modular approach.

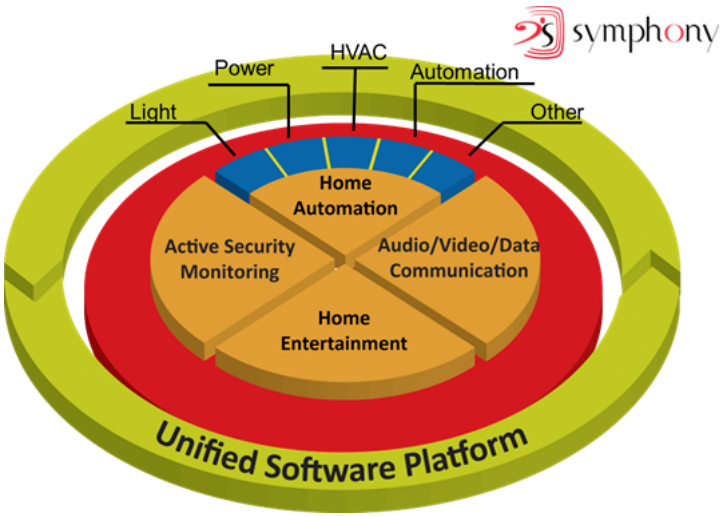

Figure 3.5: Symphony by Nextworks: the integrated Smart IoT platform concept

Symphony can monitor, supervise and control many different building systems, devices, controllers and networks available from third-party suppliers. By intelligently correlating cross-system information, it results in a flexible and highly efficient platform for home and building automation. Moreover, the Symphony Insight management station in the cloud allows operations, administration and management of the Building Management System (BMS) from any authorized remote terminal. The innovative BM-as-a-service paradigm 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.

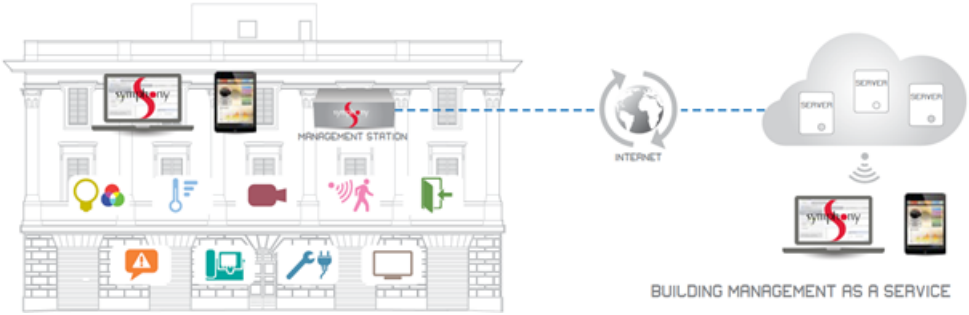

Figure 3.6: Symphony's Building Management as a Service concept

3.3.1.1 Symphony high level architecture and core elements

The Symphony digital living platform consists of a series of modules which implement its IoT platform 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.

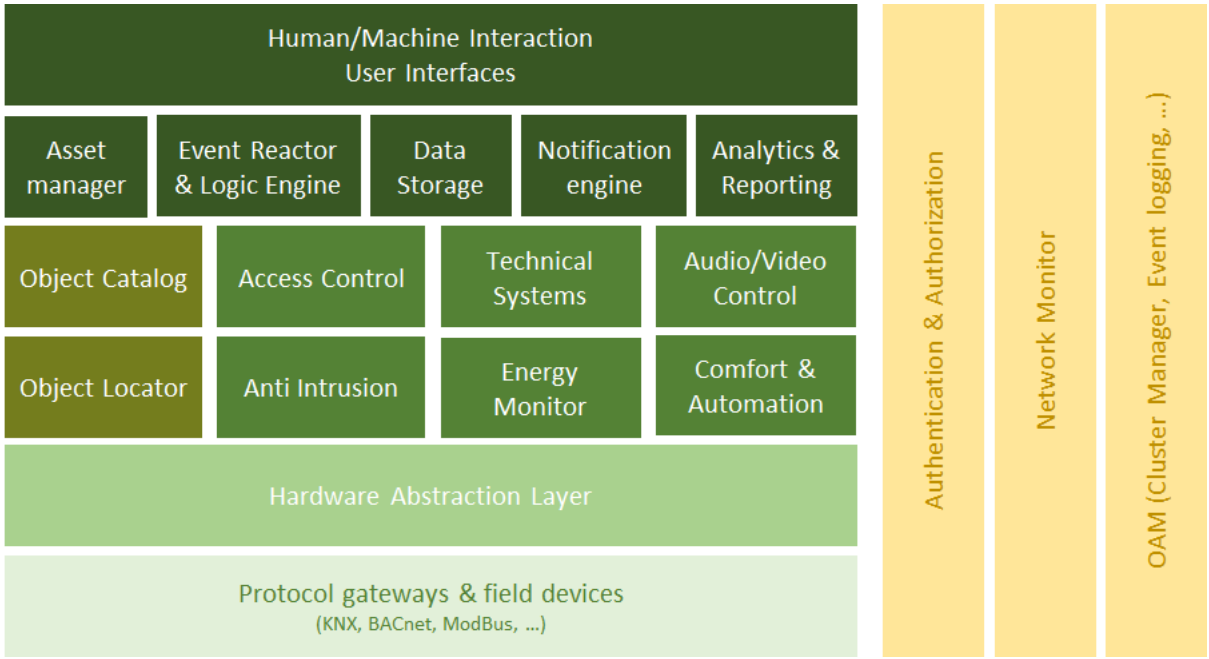

Figure 3.7: Symphony's high level architecture

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.

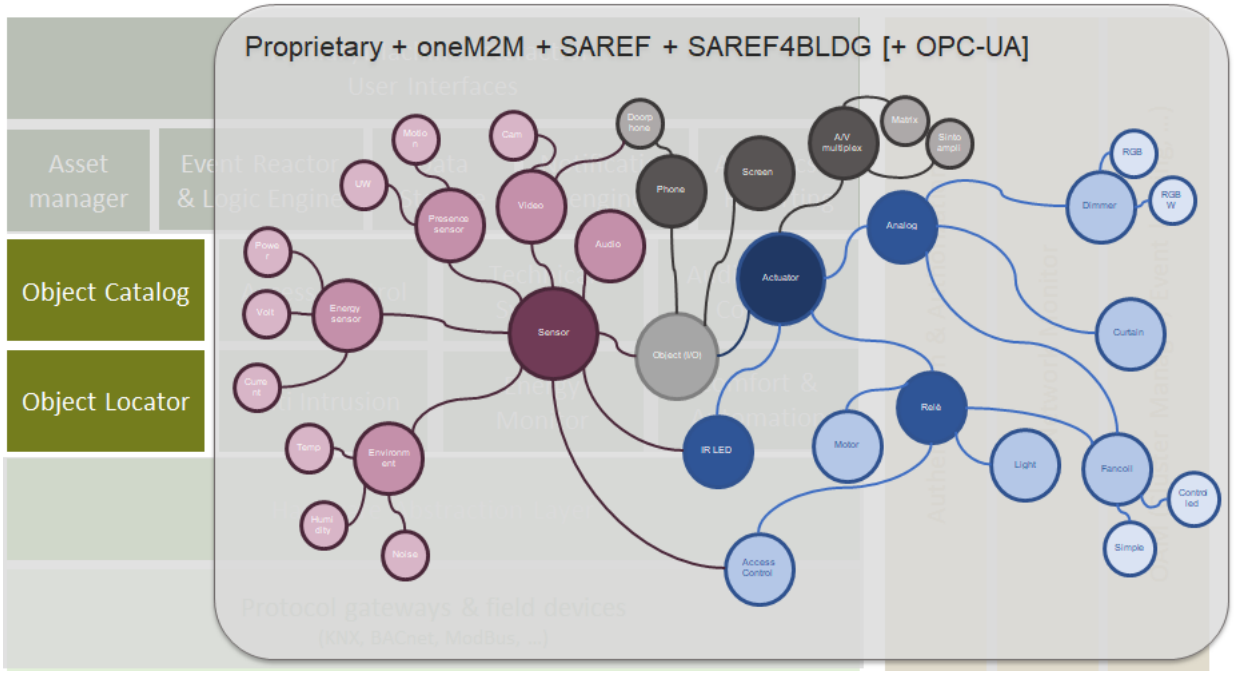

Figure 3.8: Symphony's Information Model

For example, the generic light object interface provides a capability query and proportional controls for the red, green, blue and white channels; the generic camera interface provides control for pan, tilt and zoom motors, as well as an interface to request a video stream or still images, and motion detection event notification. Implementation of the data model is actually spread over all the software modules which make up the middleware, which handles communication and notification mostly via CORBA and REST interfaces.

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, DALI, Z-Wave, several proprietary protocols over ModBus/TCP, ModBus/RTU, serial connection, SNMP. The external interfaces of the module are string-based TCP protocol, REST and protocol buffers. Drivers for Zigbee, M-bus, BACnet, Cisco EnergyWise, 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. The main role of the controller is translating commands to each device's specific protocol and transmitting them over specific channels depending on the device (dry contacts, local RS-232 connections, infrared, Ethernet). The controller is also used to store macros (i.e. sequences of commands) with a local scope. Supported devices include any infrared controlled devices (unidirectional

control), as well as widespread market appliances (e.g. Samsung, Panasonic, LG, Sharp televisions; Denon, Integra, Marantz, Linn amplifiers) that can have a bidirectional control. DLNA support is provided as Media Server and Media Player profiles, while a Media Renderer interface is under development. The external interfaces are based on CORBA IDL and string-based TCP protocol.

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 PTZ control and display multiple cameras by accessing a single, camera-independent interface. The supported devices include ONVIF cameras, proprietary cameras (e.g.: Mobotix, FLIR).

Communications controller. An Asterisk-based IP switchboard for voice and video communications. It supports any 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. The module exposes various interfaces like CORBA IDL, DTMF, ECMA CSTA.

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). Policies are modular and pluggable, and can exploit any kind of actuators and information available to the middleware (including environment data, external weather sources, etc.). Polices can range from simple priority based load control strategies, to more sophisticated self-learning algorithms to schedule loads depending on smart grid optimization. The module exposes CORBA IDL and string-based TCP protocol interfaces.

Notification engine. The notification engine provides the middleware modules with an opaque asynchronous message bus, operating according to the publish/subscribe paradigm. It is based on CORBA notification facilities.

Data storage. The data storage module allows each application to save configuration, state parameters and historical data. It currently supports an SQL back-end (to store static data) and an 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, AMQP and 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

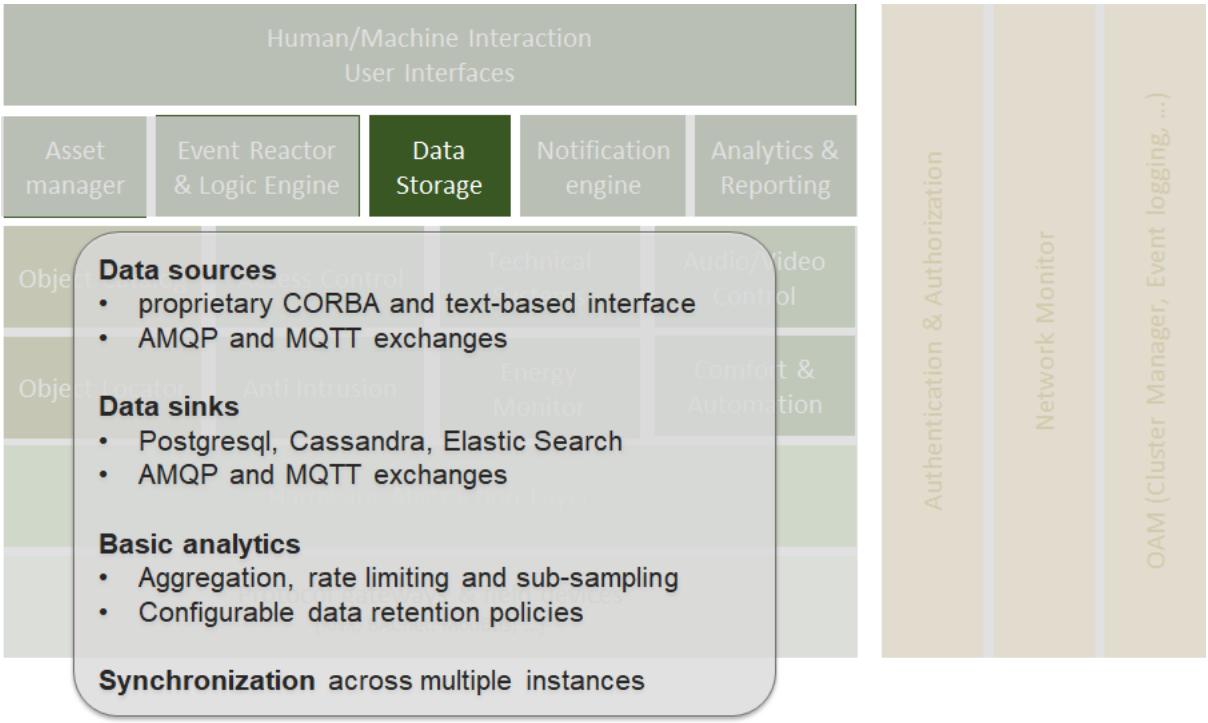

Figure 3.9: Symphony's data storage architecture

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 the end user, but are usually decoupled from the actual service implementation.

Figure 3.10: 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)

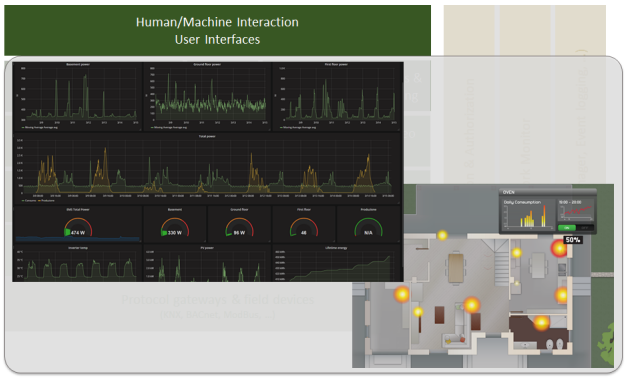

Figure 3.11: Symphony's Dashboards

The high level architecture of the Symphony cloud system is depicted in Figure 3.12.

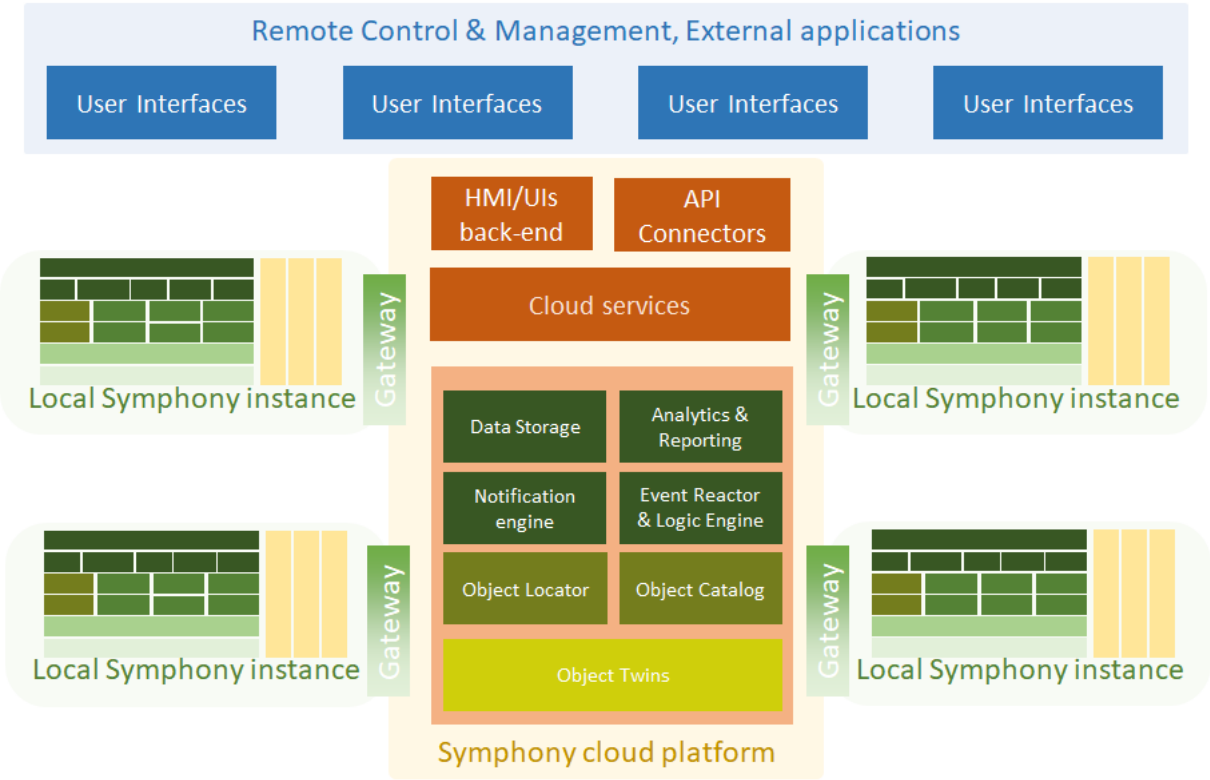

Figure 3.12: Symphony's high level architecture

3.3.1.2 Options for implementing Symphony functions as 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

Other specific requirements are under study and will be specified in future releases of this document.

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.

Other specific performance requirements are under study and will be specified in future releases of this document.

3.3.2 Home Automation And Internet of Things Deployment Scenarios

3.4 Smart Contracts

Smart contracts are executed on the blockchain which is often seen as a public ledger. The blockchain is a recent technology that stores data inside a chain of blocks cryptographically linked together so that one can't modify the integrity of the data without others detecting it. Usually smart contracts take as input a state stored inside a block, execute their code and update the state accordingly. The output of the smart contract will then be stored in a new transaction in the next block.

3.4.1 Smart Contracts Deployment Scenarios

Nowadays smart contracts can be written using a fairly limited language because there are many restrictions that need to be taken in account. The way a blockchain creates blocks is always based on the assumption that the different peers acting will reach a consensus. For instance, the well-known Bitcoin or Ethereum miners need to compute a hard problem to prove that they spent time on it so that they can't create blocks randomly. To reach this consensus, the miners need to execute the transactions to get the output that will define the new states but those executions need to be limited in some way. If not, anyone could write a piece of code with

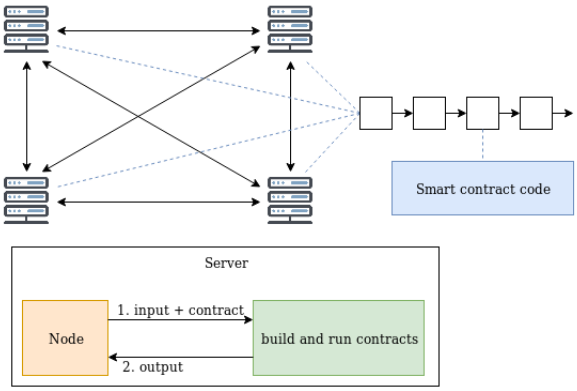

Figure 3.13: Smart Contracts Deployment Scenario

a infinite loop and then stall the entire system. Ethereum miners, for instance, are paid depending on the computational power required by a smart contract and that defines the transaction fee.

Another important aspect is the determinism of the execution because each miner must be able to reproduce the exact same output. Ethereum is using EVMs (Ethereum Virtual Machines) to execute the byte code of a smart contract written in Solidity, and then it doesn't suffer from the pain of floating-point precision or multi-threading scheduling because the EVM/Solidity is designed to avoid those aspects.

In summary, integrating a virtual machine directly into a distributed ledger solves some problems but introduces costs and limitations on the way a smart contract can be implemented.

3.4.1.1 Unikernels

Ideally we want to replace the usage of a virtual machine by unikernels that would let anyone write smart contracts using a supported language. Each peer acting for the blockchain can then execute the unikernel of a given contract previously built according to the platform the peer is running.

The direct benefit of doing so is to increase the number of potential distributed applications that can be written because one problem that developers are currently facing with Solidity is the limited computional power/available libraries that prevents 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.

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-based smart contract should result in the same changes to the distributed ledger.

In summary, the important aspects required are:

- Fast boot as each execution is independent

- A budget in terms of resources for a given execution
- Deterministic execution over any supported platform

3.4.1.2 Deployment and Security

As mentioned previously, the blockchain is a distributed system and one aspect and benefit of this should be the versatility of the deployment. This is required because you want as many actors as possible in the system to increase the security. The more heterogenous the system is, the more difficult and expensive it is to build an attack against it. Concretely, imagine that a failure in one hypervisor's 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 will be detected and the integrity of the ledger will be protected.

4 Target Deployment Scenarios

4.1 Serverless Computing

4.1.1 Digital Content Target Deployment Scenarios

Following the previous domain application introduction the CSUC use cases will focus on the media converter sevice which is in charge of convert images to more lightweight formats. The goal consists on 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.

4.1.1.1 Architecture

The figure 4.1 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 it finishes converting will put the 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.

4.1.1.2 Use cases

As orchestrators CSUC uses OpenNebula to manage Virtual Machines over KVM and Rancher to manage kubernetes with Pods over OpenNebula. The main use case CSUC is planning to develop is an OpenNebula driver to make OpenNebula allowed to manage the different kind of Unikernels used to convert the different types of files. OpenNebula will store in its datastores the different unikernels ”images” previously prepared with UNICORE and deploy it over KVM like if it were a Virtual Machine and run the task as explained before 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 and used to gather any kind of

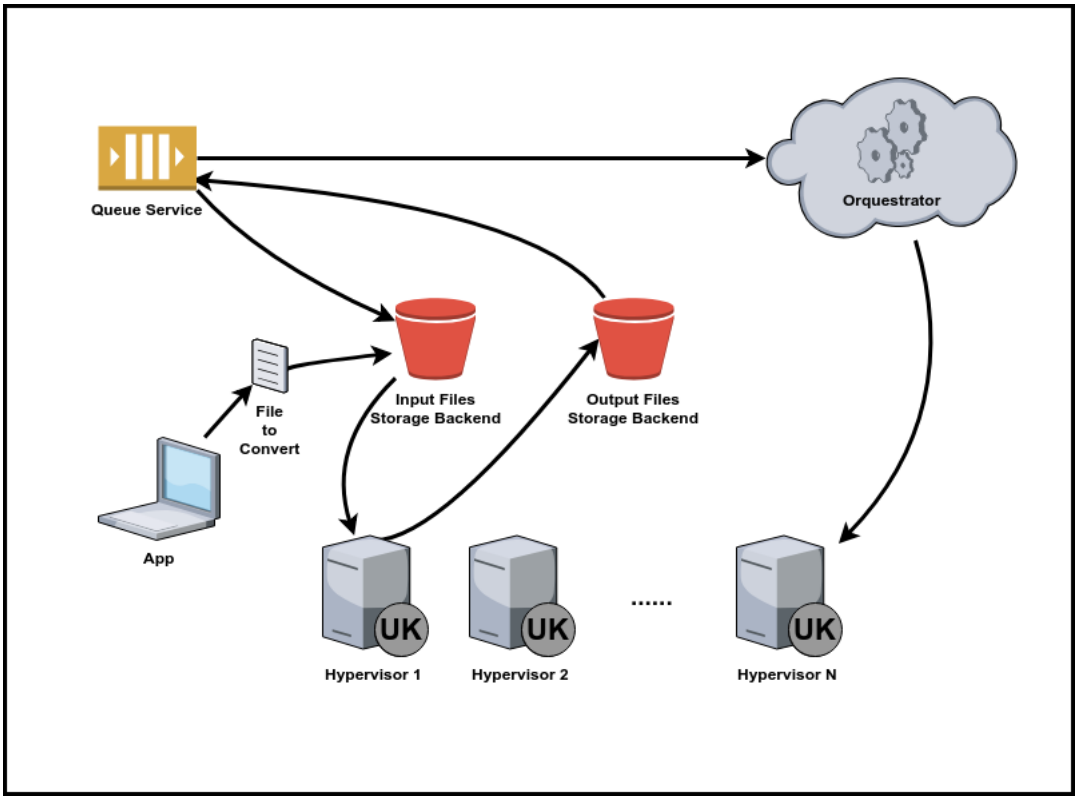

Figure 4.1: Serverless architecture of media converter service

information that can use later to decide if instantiate or not more Unikernels process over KVM hipervisor.

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

Both use cases will use all the components developed by 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.

4.1.1.3 What is expected about the trials

The trials will consist on instantiate a specific number of unikernel, virtual machines and kubernetes if it's affordable on same characteristics nodes 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

4.1.2 Lambda Packet Processing Target Deployment Scenarios

4.1.2.1 PacketCloud lambda service for packet processing deployment overview

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

- Efficiency
- Security
- Flexibility

Next we offer a deployment overview for PacketCloud (as depicted by 4.2):

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

  (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.

4.2 Network Function Virtualization

4.2.1 Broadband Network Gateway Target Deployment Scenarios

Unicore unikernels implementation assumes the decomposition of the 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 clients' application domains, providing the capability and flexibility to provide customer specific implementation and resource allocation, optimization and fast deployment of the services.

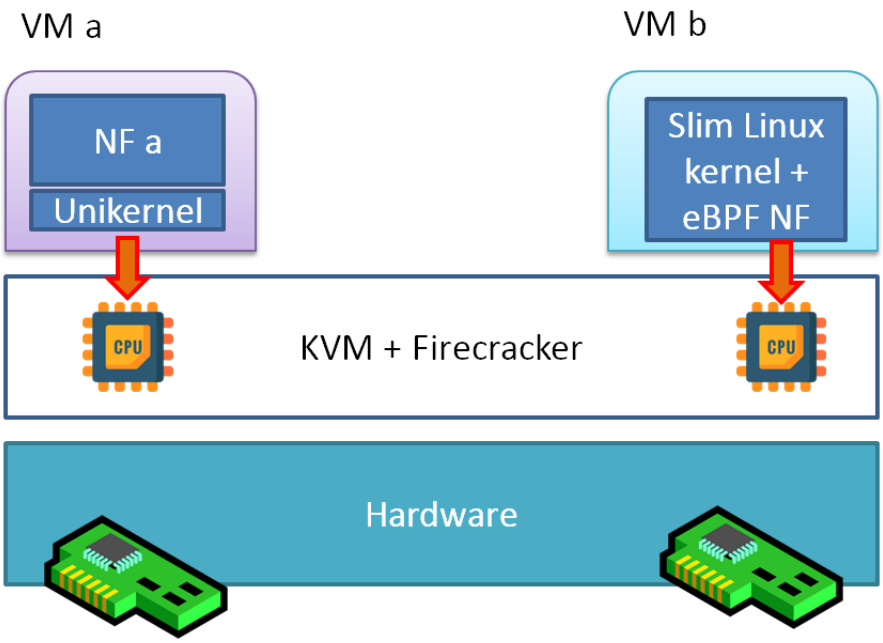

Figure 4.2: PacketCloud deployment overview

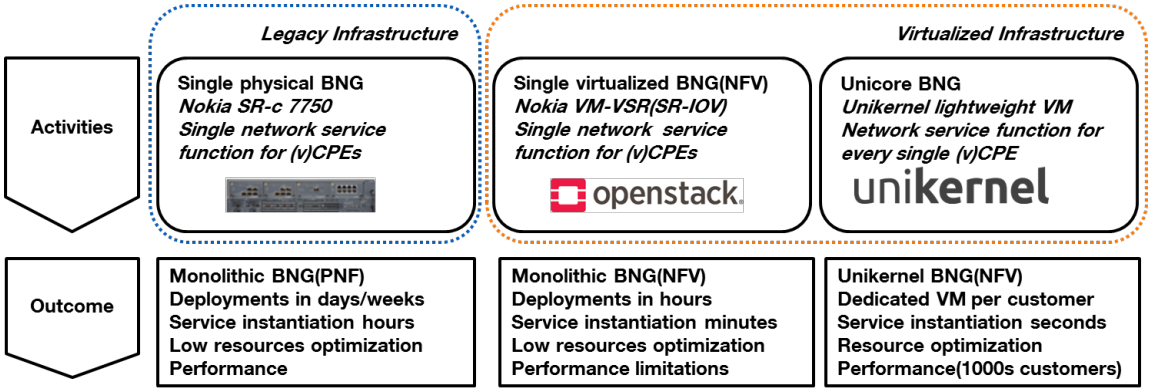

Figure 4.3: Comparison of legacy and virtualised BNG infrastructure

Unicore implementation for unikernel VMs per customer is expected to change the implementation model, the monitoring and resource control per customer and for the network, to provide improved capabilities and capacity for customers, depending on several subscription models.

From a high level perspective, the Unicore unikernel implementations should provide the possibility of light VMs deployments dedicated for one customer or per CPE, automated service instantiation, scalability capabilities, per region service provisioning, monitoring tools for resource allocation and service KPIs, metrics performance. The entire Unicore system is expected to be supported on a virtualized environment, VNFs with VMs Openstack (KVM hypervisor) based or containerized infrastructure, with the possibility of introducing several open tools for orchestration and service instantiation. The infrastructure platform deployments will support specific integration of lightweight BNGs VMs. The scenario should be extended for 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 the system proposed should respect customer service KPIs and network KPIs, in terms of bandwidth, delay, provisioning time, capacity, resource consumption and

efficiency.

A successful overall implemented scenario for unikernel apps 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, as OSM for resources, in a virtualized (Openstack based) or containerized (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 (Heat, ONAP). 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 in unikernel implementation requires advanced security features, at BNG apps level, Unicore embedded, to minimize the exposure to different security attacks (unicore by design), hardware and platform security level and more relevant for a telco operator, by the decomposing process of monolithic application into smaller building blocks running in an isolated environment (one of each other BNGs).

4.2.2 5G vRAN NFV Clusters Target Deployment Scenarios

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 DU simulator is available, performance evaluation of the dRIC could be based on the time taken to spin up many Cell Control Plane instances.

4.3 Home Automation And Internet of Things

4.3.1 IoT Deployment Scenarios with Symphony by Nextworks

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 following pictures depict possible installations inside the building:

Figure 4.4: Nextworks trial for home automation use case: domotic meeting room

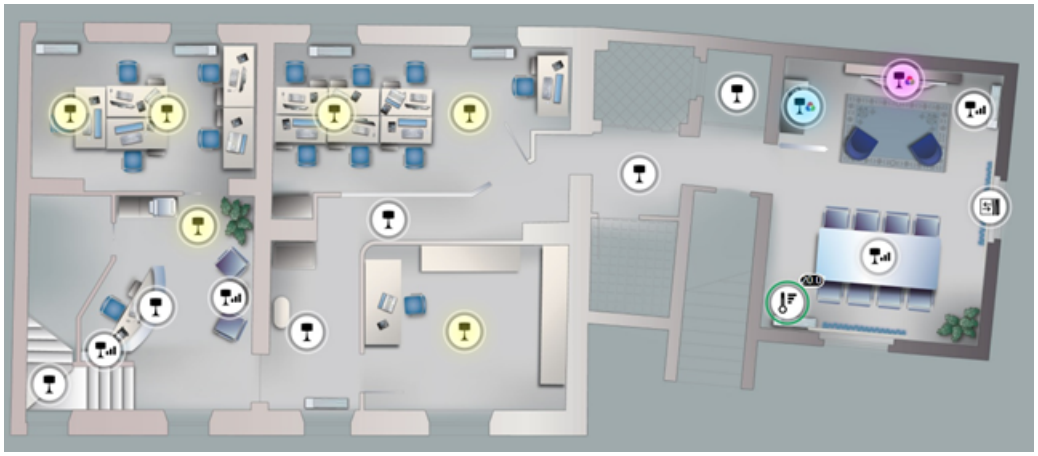

Figure 4.5: Nextworks trial for home automation use case: IoT devices at ground floor.

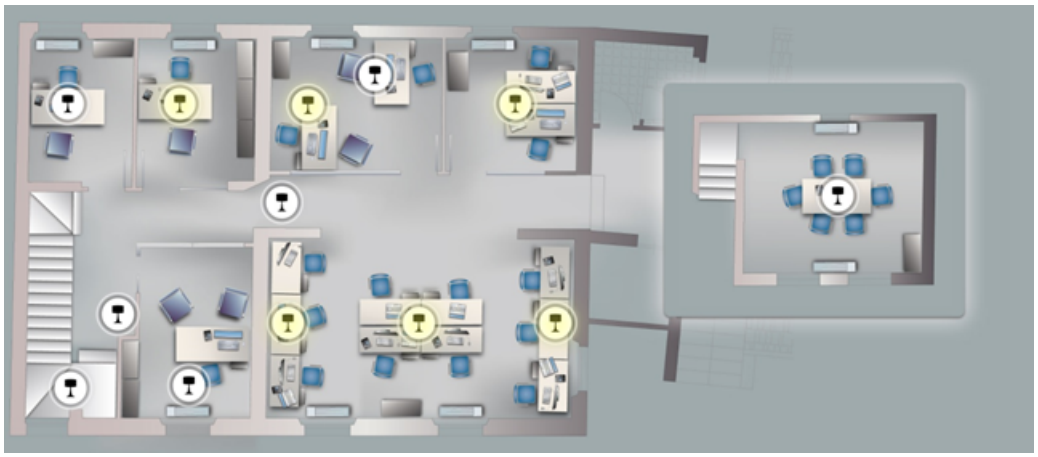

Figure 4.6: Nextworks trial for home automation use case: IoT devices at first floor.

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 in Chapter 3 with the primary goals of

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

4.4 Smart Contracts

4.4.1 Smart Contracts Target Deployment Scenarios

Chapter 3 introduced the notion of virtual machines in the Ethereum blockchain to run smart contracts written by developers using the Solidity language, so that the execution can be limited and controlled. EPFL use cases

Figure 4.7: Nextworks trial for home automation use case: external IoT devices (CCTV cameras)

will focus on the way smart contract are written to give a more flexible solution. The goal of the experiments will be to show how a generic language can be used to develop distributed applications that couldn't be deployed on existing blockchains. UPB will support EPFL with designing and implementing the unikernel features for deterministic execution in a generic language such as Rust or C or C++.

4.4.1.1 Build Tools

When writing a smart contract, developers must be aware of specific limitations like accessing external resources. A smart contract can't read or write external resources at runtime because of the non-deterministic nature of it. For instance, a request to a distant server to get some data might end up with different results for each node or a request for a random value from the host machine's entropy source should be different from one node to another.

Then the first step of the experiment will be to use the build tools and insure that smart contracts with dangerous behaviour get warnings. We will also run the resulting unikernels to show that they produce the correct result on different architectures that have known differences.

4.4.1.2 Usage

The next step of the experiment will be to use unikernel-based smart contracts with a live deployment of the Cothority (the set of nodes involved in the DEDIS blockchain). As mentioned in chapter 3, the goal here is to insure that a consensus can be reached independently from the architecture or the platform of the nodes but we will focus first on x86- and ARM-based machines.

The experiment itself will focus on deploying the smart contracts in the blockchain to finally execute a set of transactions. First an intermediate representation of the smart contract needs to be stored so that each node can use the unikernel build kit and make a unikernel compatible with the host machine. When a transaction

needs to be executed, each node will follow those steps:

(i) Extract the code from the block

(ii) Build a compatible unikernel

(iii) Execute the unikernel with the arguments and the previous state

(iv) Read the output to store the new state

Note that the first step can be avoided because the image can be cached and reused later on. The second step will involve different machines with different available services. The goal is to be able to run unikernel-based smart contracts over multiple environments, with or without hypervisor available for instance.

4.4.1.3 Assessment

The goal is for the unikernel-based deployment to ensure consensus for smart contracts. If improperly implemented, intentional or unintentational misuse of the provided features will break consensus. We will assess the deterministic behavior in two ways:

(i) Generate smart contracts using consensus-breaking features such as multi threading and show that they are rejected by the build tools

(ii) Use guided generation of random viable smart contracts (i.e. not using consensus-breaking features) and used techniques inspired from fuzz testing to show they work properly and run in a deterministic manner and that they are accepted by build tools

The first approach is used to ensure the are no false negatives and the second approach will ensure there are both no false negatives and no false positives. Invalid inputs (the first approach) must be rejected at built time, while valid inputs (the second approach) must be accepted at build time and must not break consensus.

4.4.1.4 What Is Expected About the Trials

The trials will consist on a live Cothority deployed with a variety of environment, in the sense that we want the nodes to use different platforms and hypervisors like KVM or Xen. The goal is to insure the deterministic behaviour of multiple executions of smart contracts written with a generic purpose language, as opposed to a language designed specifically for smart contracts (e.g. Rust versus Solidity). This is mandatory for nodes to reach a consensus when executing the transactions. In summary, we want the following:

- Dynamic deployment of smart contracts on the blockchain
- Equivalent performance compared to pre-compiled contracts
- Enforce limited resources per contract execution (e.g. cycles, RAM)
- Deterministic execution over different architectures

5 Tooling Use Cases

The data presented now is an outline of the content to be provided in the final version of D2.1 in September.

5.1 Decomposition Tool Use Cases

5.1.1 Decomposition Tool Use Case 1

5.1.2 Decomposition Tool Use Case 2

5.1.3 Decomposition Tool Use Case n

5.2 Dependency Analysis Tool Use Cases

5.2.1 Dependency Analysis Tool use case 1

5.2.2 Dependency Analysis Tool use case 2

5.2.3 Dependency Analysis Tool use case n

5.3 Automatic Build Tool Use Cases

5.3.1 Automatic Build Tool use case 1

5.3.2 Automatic Build Tool use case 2

5.3.3 Automatic Build Tool use case n

5.4 Verification Tool Use Cases

5.4.1 Verification Tool use case 1

5.4.2 Verification Tool use case 2

5.4.3 Verification Tool use case n

5.5 Performance Optimisation Tool Use Cases

5.5.1 Performance Optimisation Tool use case 1

5.5.2 Performance Optimisation Tool use case 2

5.5.3 Performance Optimisation Tool use case n

6 Unikernel Core Technical Requirements

The data presented now is an outline of the content to be provided in the final version of D2.1 in September.

6.1 UNICORE Unikernel Requirements

6.1.1 General Requirements

6.1.2 API Requirements

6.1.3 Orchestration Environment Integration Requirements

6.1.4 Security and Isolation Requirements

6.1.5 Deterministic Execution Requirements

7 Unicore Toolchain Technical Requirements

The data presented now is an outline of the content to be provided in the final version of D2.1 in September.

7.1 Overall Toolchain Requirements

7.1.1 Host Platform Requirements

7.1.2 Target Platform Requirements

7.2 Decomposition Tool Requirements

7.3 Dependency Analysis Tool Requirements

7.4 Automatic Build Tool Requirements

7.5 Verification Tool Requirements

7.6 Performance Optimisation Tool Requirements