Ref. Ares(2021)1538404 - 28/02/2021

Adaptive edge/cloud compute and network continuum over a heterogeneous sparse edge infrastructure to support nextgen applications

Deliverable D3.1

Edge infrastructure pool framework report (I)

ACCORDION receives funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement No. 871793

DOCUMENT INFORMATION

PROJECT	
PROJECT ACRONYM	ACCORDION
PROJECT FULL NAME	Adaptive edge/cloud compute and network continuum over a heterogeneous sparse edge infrastructure to support nextgen applications
STARTING DATE	01/01/2020 (36 months)
ENDING DATE	31/12/2022
PROJECT WEBSITE	http://www.accordion-project.eu/
TOPIC	ICT-15-2019-2020 Cloud Computing
GRANT AGREEMENT N.	871793
COORDINATOR	CNR
DELIVERABLE INFORMATION	
WORKPACKAGE N. | TITLE	WP 3 | Edge infrastructure pool framework
WORKPACKAGE LEADER	HPE
DELIVERABLE N. | TITLE	D. 3.1 | Edge infrastructure pool framework report (I)
EDITOR	E. Psomakelis (ICCS)
CONTRIBUTOR(S)	E. Psomakelis (ICCS), Blasi (HPE), A. Vailati (HPE), H. Kavalionak (CNR), I. Korontanis (HUA), F. Huici (NEC), P. Dazzi (CNR)
REVIEWER	Saman Zadtootaghaj (TUB)
CONTRACTUAL DELIVERY DATE	02/2021
ACTUAL DELIVERY DATE	28/02/2021
VERSION	1.0
TYPE	Report
DISSEMINATION LEVEL	Public
TOTAL N. PAGES	51
KEYWORDS	Resource management, Resource Monitoring, Resource indexing;

EXECUTIVE SUMMARY

This deliverable provides the first report summarizing the scientific advancements achieved during the project, by the WP3 tasks. The achievements, risks and challenges are presented both at a high level, presenting the outcome of the WP3 tasks (called ACCORDION Minicloud VIM) as a unified component, and at a lower level, presenting the components that comprise the ACCORDION Minicloud VIM. For each component we can clearly identify the progress through the first year of the project, the challenges and problems encountered and the plans for the second year of the project.

DISCLAIMER

ACCORDION (871793) is a H2020 ICT project funded by the European Commission.

ACCORDION establishes an opportunistic approach in bringing together edge resource/infrastructures (public clouds, on-premise infrastructures, telco resources, even end-devices) in pools defined in terms of latency, that can support NextGen application requirements. To mitigate the expectation that these pools will be "sparse", providing low availability guarantees, ACCORDION will intelligently orchestrate the compute & network continuum formed between edge and public clouds, using the latter as a capacitor. Deployment decisions will be taken also based on privacy, security, cost, time and resource type criteria.

This document contains information on ACCORDION core activities. Any reference to content in this document should clearly indicate the authors, source, organisation and publication date.

The document has been produced with the funding of the European Commission. The content of this publication is the sole responsibility of the ACCORDION Consortium and its experts, and it cannot be considered to reflect the views of the European Commission. The authors of this document have taken any available measure in order for its content to be accurate, consistent and lawful. However, neither the project consortium as a whole nor the individual partners that implicitly or explicitly participated the creation and publication of this document hold any sort of responsibility that might occur as a result of using its content.

The European Union (EU) was established in accordance with the Treaty on the European Union (Maastricht). There are currently 27 members states of the European Union. It is based on the European Communities and the member states' cooperation in the fields of Common Foreign and Security Policy and Justice and Home Affairs. The five main institutions of the European Union are the European Parliament, the Council of Ministers, the European Commission, the Court of Justice, and the Court of Auditors (http://europa.eu.int/).

Copyright © The ACCORDION Consortium 2020. See https://www.accordion-project.eu/ for details on the copyright holders.

You are permitted to copy and distribute verbatim copies of this document containing this copyright notice, but modifying this document is not allowed. You are permitted to copy this document in whole or in part into other documents if you attach the following reference to the copied elements: "Copyright © ACCORDION Consortium 2020."

The information contained in this document represents the views of the ACCORDION Consortium as of the date they are published. The ACCORDION Consortium does not guarantee that any information contained herein is error-free, or up to date. THE ACCORDION CONSORTIUM MAKES NO WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, BY PUBLISHING THIS DOCUMENT.

REVISION HISTORY LOG

VERSION No.	DATE	AUTHOR(S)	SUMMARY OF CHANGES
0.1	1/10/2020	E. Psomakelis (ICCS)	T.O.C.
0.2	10/11/2020	E. Psomakelis (ICCS)	Revised T.O.C.
0.3	27/11/2020	E. Psomakelis (ICCS) L. Blasi (HPE) H. Kavalionak (CNR) I. Korontanis (HUA) F. Huici (NEC)	Integrated architecture contributions
0.4	30/12/2020	E. Psomakelis (ICCS) A. Vailati (HPE) H. Kavalionak (CNR) I. Korontanis (HUA) F. Huici (NEC)	Integrated component description contributions
0.5	10/01/2021	E. Psomakelis (ICCS)	Completed Introduction and Conclusion sections
0.6	14/01/2021	E. Psomakelis (ICCS)	Corrected numbering in figures, tables and references
0.7	24/02/2021	E. Psomakelis (ICCS) A. Vailati (HPE) H. Kavalionak (CNR) I. Korontanis (HUA) F. Huici (NEC)	Applied suggestions after internal review
1.0	28/02/2021	P. Dazzi (CNR)	Final version

GLOSSARY

AMD	Advanced Micro Devices, Inc.
API	Application Programming Interface
ARM	Advanced RISC Machines
AWS	Amazon Web Services
AWS S3/Amazon S3	Amazon Simple Storage Service
CLI	Command Line Interface
CNCF	Cloud Native Computing Foundation
CPU	Central Processing Unit
CRD	(Kubernetes) Custom Resource Definition
CRI	Container Runtime Interface
CSI	Container Storage Interface
DoA	Description of Action
EC	European Commission
ETSI	European Telecommunications Standards Institute
EU	European Union
GUI	Graphical User Interface
H2020	Horizon 2020 EU Framework Programme for Research and Innovation
IoT	Internet of Things
K3S	Lightweight Kubernetes
KVM	Kernel-based Virtual Machine
LF	Linux Foundation
Minicloud	The ACCORDION edge resources cloud of IoT and/or other edge devices.

N.A.	Not Available (or No Answer)
NFV	Network Functions Virtualization
OCCI	Open Cloud Computing Interface
OS	Operating System
PromQL	Prometheus Query Language
QoE	Quality of Experience
QoS	Quality of Service
RAM	Random Access Memory
RBAC	Role Based Access Control
REST	Representational state transfer
RISC	Reduced instruction set computer
TBD	To Be Defined
UI	User Interface
UML	Unified Modelling Language
VIM	Virtual Infrastructure Manager
VM	Virtual Machine
VMI	Virtual Machine Instance

TABLE OF CONTENTS

1 Introduction 10
1.1 Scope and objectives of this deliverable 10
1.2 Document Structure 10
2 Architecture 11
2.1 General Architecture 11
2.2 Resource Monitoring & Characterization 13
2.3 Resource Indexing & Discovery 15
2.4 Edge Storage 16
2.5 Lightweight Virtualization 18
2.6 Edge Minicloud VIM 19
3 Resource monitoring & characterization 22
3.1 Description & Objectives 22
3.2 Requirements 22
3.3 Research Challenges & Advancements Achieved 23
3.4 Provided Features and APIs 23
3.5 Technical Challenges and Mitigation 27
3.6 Future Work 28
4 Resource indexing & discovery 29
4.1 Description & Objectives 29
4.2 Requirements 29
4.3 Research Challenges & Advancements Achieved 29
4.4 Provided Features and APIs 31
4.5 Technical Challenges and Mitigation 31
4.6 Future Work 31
5 Edge storage 32
5.1 Description & Objectives 32
5.2 Requirements 32
5.2.1 General Requirements 32
5.2.2 Algorithmic Access Requirements 32
5.2.3 Data Administrator Requirements 33
5.3 Research Challenges & Advancements Achieved 33
5.4 Provided Features and APIs 33
5.5 Technical Challenges and Mitigation 33
5.6 Future Work 34
6 Lightweight virtualization 35
6.1 Description & Objectives 35
6.2 Requirements 36

6.3 Research Challenges & Advancements Achieved 37
6.4 Provided Features and APIs 38
6.5 Technical Challenges and Mitigation 39
6.6 Future Work 40
7 Edge minicloud VIM 41
7.1 Description & Objectives 41
7.2 Requirements 41
7.3 Research Challenges & Advancements Achieved 42
7.4 Provided Features and APIs 47
7.5 Future Work 48
8 Conclusions 49
References 50

1 Introduction

1.1 Scope and objectives of this deliverable

This deliverable aims at delivering the full scope of the research and development tasks conducted in work package 3 of the ACCORDION project during the first year of the project (months 1-12), which is also mentioned as the first "cycle". The document contains both the architecture of the complete Minicloud as well as the architecture of its components. It also contains the objectives, the scientific and technical challenges and the advancements made towards overcoming these challenges. Finally, it presents the plans for the near future as well as the next cycle of the work package tasks. This information is provided in order to present a coherent and complete status of the work package tasks.

1.2 Document Structure

The document presents an architecture overview of the Minicloud platform as well as its components in section 2. In sections 3-7 we can see the details of each component of the Minicloud. In detail, each one of these sections consists of a general description, a requirements, a research challenges, a features and APIs, a technical challenges and a future work sub-sections, painting the complete picture for each component. Finally, there is the conclusions and references sections that conclude this document.

2 Architecture

2.1 General Architecture

The main product of WP3, as indicated by its title, is the "Edge infrastructure pool framework", also named "Edge Minicloud" or simply Minicloud.

From the DoA, an Edge Minicloud is "an orchestration system for pooling and abstracting resources, exposing elasticity properties through a standard-based API", "tailored towards integrating edge and fog nodes (private clouds and infrastructures, telco resources and end-devices) in a single minicloud orchestration system".

The main logical components of the ACCORDION Minicloud are indicated in the UML diagram depicted in Figure 1 and described below. A more detailed description for each component is provided in the following subsections.

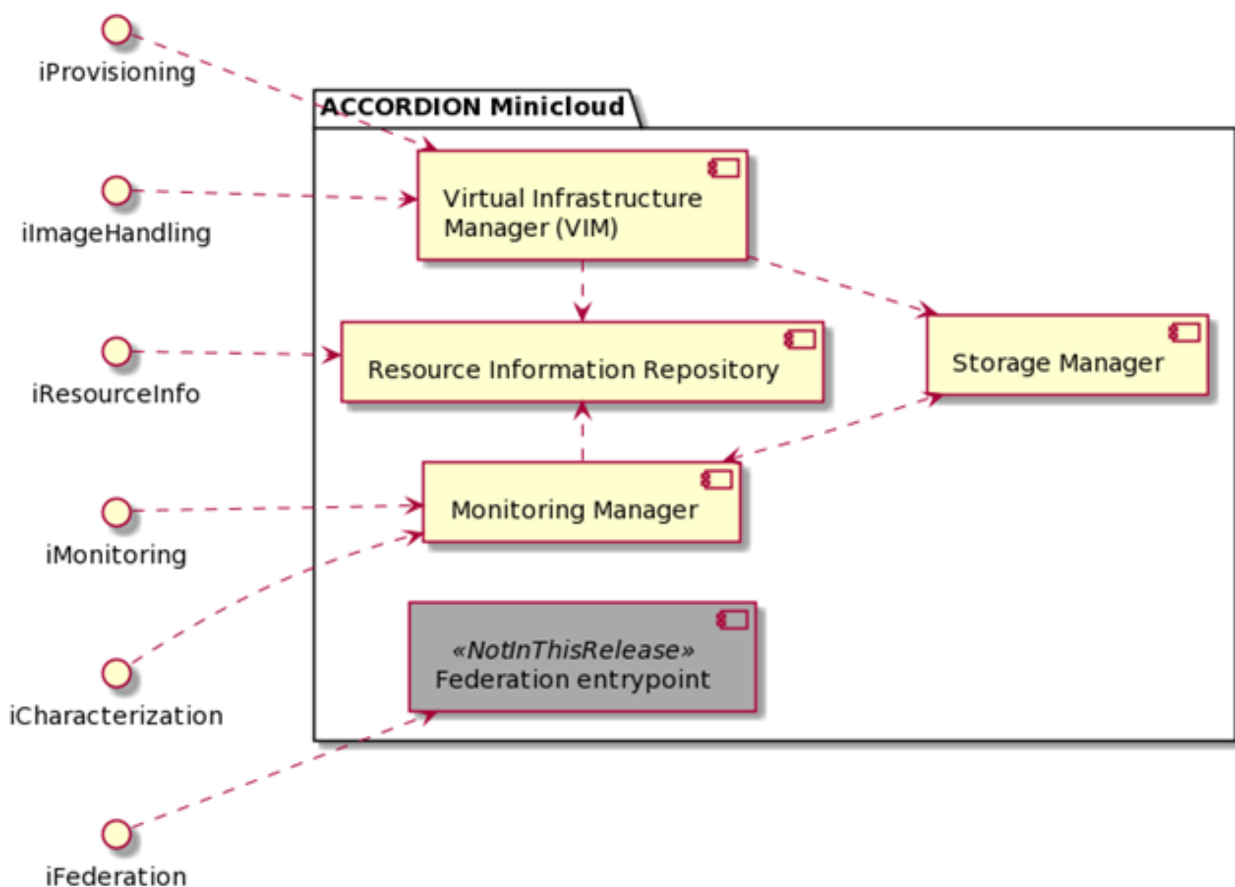

Figure 1: ACCORDION Minicloud components' diagram.

ACCORDION Minicloud

This package represents the ACCORDION Minicloud, which is a fundamental component of the ACCORDION system. A Minicloud manages a single resource pool. The managed resources are typically Edge resources, but could also be Cloud resources. The basic criterion that groups resources in a single pool is their locality, which can be realized either in geographical terms or in terms of latency. Geographical locality means that the resources handled by a single Minicloud pool are expected to share the same location or to be geographically close. Network locality is a more difficult concept to define (e.g. [1] defined the locality of two nodes as something proportional to the length of the common prefix of their IP address: the longer the prefix, the shorter the distance between the nodes) and will not be explored here.

In this first version the adopted model is “one minicloud for each provider”. This means that each Minicloud pool includes resources offered by a single provider, and each provider to join the ACCORDION federation has to implement its own Minicloud. In this model the role Infrastructure Owner is the same as Minicloud Owner. In the future we will research the possibility to implement a higher granularity model by defining resource pools in terms of latency and allow a new provider/node to join the federation by joining the closest Minicloud. With this model even providers with few resources will be able to join the federation.

Monitoring Manager

This component, resulting from the work of Task 3.1, collects and provides two types of information, both fundamental for handling resources: monitoring data and resources' characterization. In particular, it monitors the availability of all resources included in the resource pool. More information on this component can be found in section 3.

Resource Information Repository

The purpose of this component, resulting from the work of Task 3.2, is to store and retrieve information about the resources of the Minicloud pool. The source of this information is the monitoring component, for both monitoring data (e.g., availability) and resources' characterization.

Storage Manager

This component, resulting from the work of Task 3.3, provides a unified interface to the storage available in the Minicloud pool.

Virtual Infrastructure Manager (VIM)

The Virtual Infrastructure Manager, resulting from the work of Task 3.5, handles the resource pool and offers the Minicloud API to provision and manage edge deployment units (Containers and VMs) running on top of the resource pool.

Federation entrypoint

This component has not been implemented in the first version of the Minicloud. When available it will expose an API to offer federation-related functionalities such as those involved in the onboarding process. ACCORDION Task 7.5 will greatly help in defining this component’s functionalities.

2.2 Resource Monitoring & Characterization

In ACCORDION, we concluded that for the purpose of monitoring we will use Prometheus[1]. Prometheus is a popular open-source system that can be used for minoring purpose. Prometheus uses a time series database to store metrics in a key-value pair format. It has its own query language named PromQL[2]. Prometheus has a pull model which basically pulls metrics from exporters. Exporters can fetch statistics from non-Prometheus systems and convert them into Prometheus metrics. We can find the metrics of an exporter at /metrics URL. To pull metrics from the exporters Prometheus must know the targets through service discovery or static configuration. In order to create graphs, Grafana performs PromQL queries to Prometheus to get the appropriate results. Also, the Alert manager produces alerts by using PromQL with upper or lower limits.

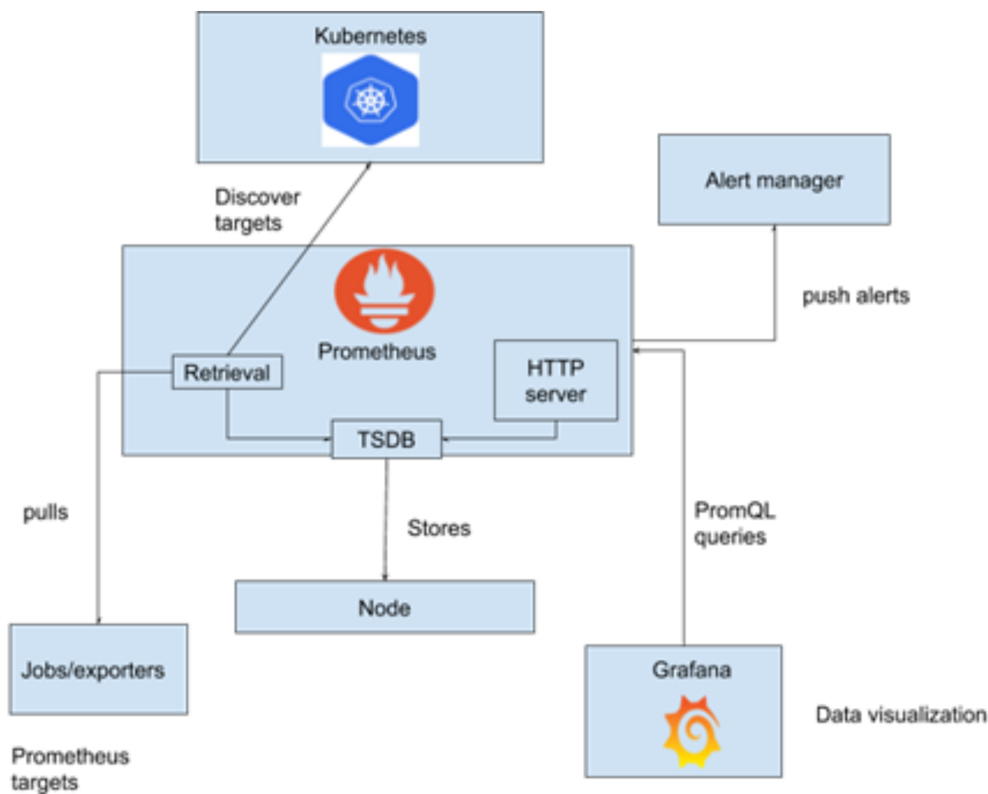

Figure 2: High-level architecture of the monitoring component.

For the deployment of Prometheus on K3s, a github Project was used Cluster Monitoring stack for ARM / X86-64[3]. This project uses Prometheus Operator to manage and configure Prometheus instances on

[1] https://prometheus.io/

[2] https://prometheus.io/docs/prometheus/latest/querying/basics/

[3] https://github.com/carlosedp/cluster-monitoring

Kubernetes / K3s. Prometheus Operator[4] can automatically generate monitoring targets configuration, so each node of the cluster will have exporters to expose their metrics to Prometheus which will be installed to the master node of Kubernetes. For bare metal or VM monitoring the node exporter pods have to be configured on every node of the cluster. The rest of the pods were configured to run only on the K3s nodes with the label monitoring master.

This project uses several other tools:

- Node exporter[5] for hardware and OS metrics.
- Blackbox exporter[6] allows blackbox probing of endpoints over HTTP, HTTPS, DNS, TCP and ICMP.
- Kube-state-metrics[7] expose critical metrics about the condition of a Kubernetes cluster, it generates them from the Kubernetes API server. (health of nodes, pods, deployments, etc.).
- Prometheus adapter[8] which is an implementation of the Kubernetes resource metrics, custom metrics, and external metrics APIs. It can also replace the metrics server on clusters that run Prometheus.
- CoreDNS as a DNS server.
- Grafana[9] to visualize the metrics with graphs.
- SMTP[10] docker container for sending emails when an alert of the Alert Manager is triggered.
- addon-resizer[11] is a container image that watches over another container in a deployment, and vertically scales the dependent container up and down. Currently the only option is to scale it linearly based on the number of nodes, and it only works for a singleton.
- kube-rbac-proxy[12] is a small HTTP proxy for a single upstream, that can perform RBAC authorization against the Kubernetes API.

---

[4] https://github.com/carlosedp/prometheus-operator

[5] https://hub.docker.com/r/carlosedp/node_exporter/

[6] https://hub.docker.com/r/carlosedp/blackbox_exporter/

[7] https://github.com/kubernetes/kube-state-metrics

[8] https://github.com/kubernetes-sigs/prometheus-adapter

[9] https://grafana.com/

[10] https://hub.docker.com/r/carlosedp/snmp_exporter/

[11] https://hub.docker.com/r/carlosedp/addon-resizer

[12] https://hub.docker.com/r/carlosedp/kube-rbac-proxy

2.3 Resource Indexing & Discovery

The aim of the component Resource Indexing & Discovery (RID) is to keep an up-to-date status of computational resources among the various miniclouds and provide a service that allows to run queries on these resources. The main source of data would be the monitoring system of the miniclouds (Task 3.1). Queries can in principle come from any ACCORDION component that aims at finding resources with specific computational features.

Figure 3 describes the generic federation-wide architecture of the RID system. In the figure we show only two miniclouds, while the actual system can be composed by more entities (dots on the figure). As you can see on the figure, we envision the RID as a distributed component. This means that there is an instance of the RID running in every minicloud. For the current design of the components, RID instances communicate with each other using the Internet. The specifics of this communication are strongly tied to the approach that will be used to implement the distributed components.

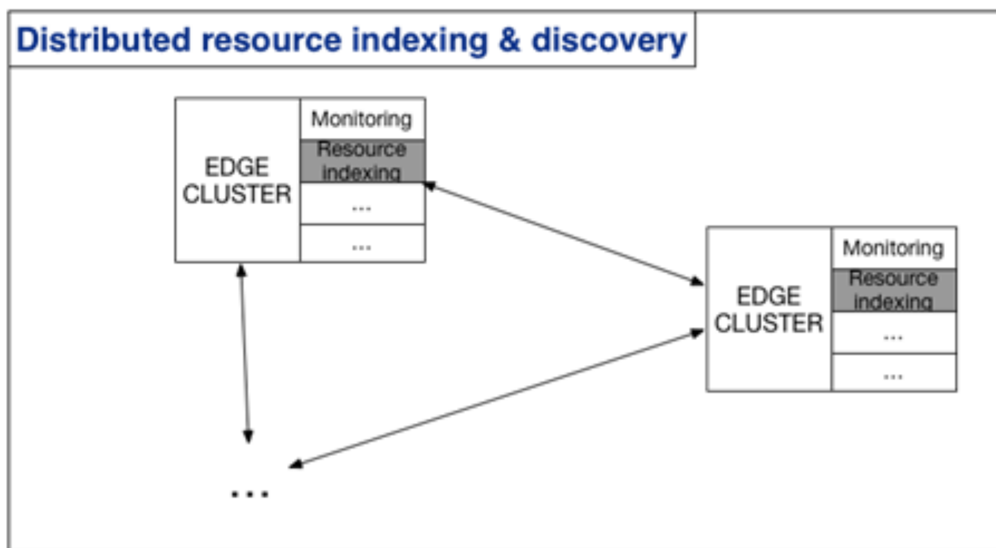

Figure 3: Overview of the overall system architecture.

Figure 4 shows the internal architecture of a given RID instance. The RID contains a REST client that will pull monitoring data from the local (i.e., in the same minicloud) monitoring instance, which will be in turn stored in the local storage component. Note that the local storage does not necessarily keep only local data, or that a minicloud information is not necessarily stored in the local storage. In fact, advanced distributed indexing techniques are employed to distribute data among all their instances, in order to have a fast yet effective information discovery and retrieval.

The RID instance exposes a REST server query interface that allows other ACCORDION components to submit queries. In case, there is the need to transform the query into software artifacts (e.g., network messages, look up command), this task is performed by the **query mapping** component. The processed query is then

forwarded to the **topology manager** component. The topology manager component is responsible for the query execution inside the distributed topology of the system. The topology manager structure depends on the type of queries that the system is optimized to process and is subject to changes during the project progress.

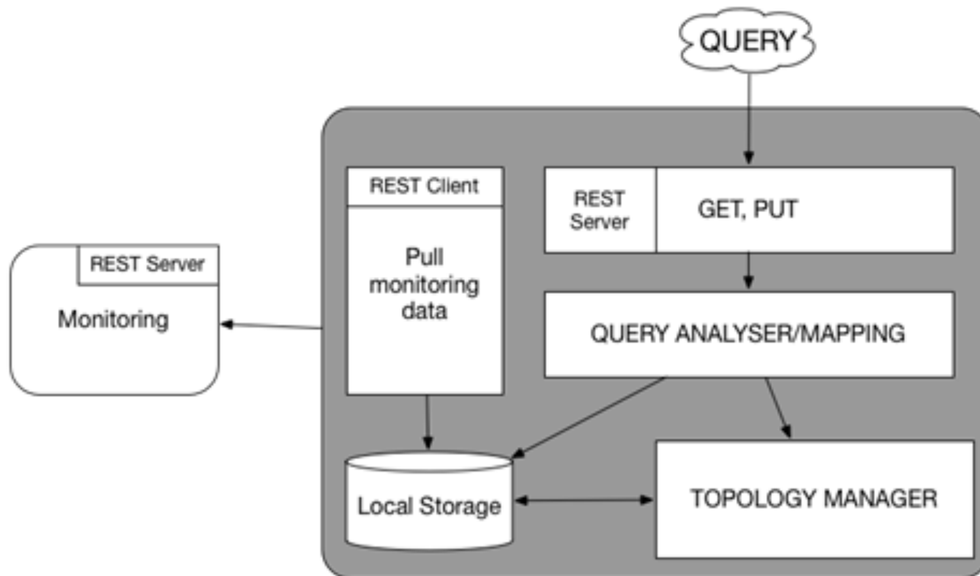

Figure 4: Architecture of the Distributed resource indexing component.

2.4 Edge Storage

The Edge storage component has the goal of providing an edge storage framework that can support the QoE needs of the users, optimizing resource usage in the edge devices and networks. We have two possible base technologies for this; the MinIO [2] and OpenStack [3] platforms that enable us to create highly distributed, lightweight and scalable storage clusters, using Kubernetes as an orchestrator. The final choice of the tool will be made after running a number of experiments, testing their effectiveness and optimizing their configuration for scenarios close to the real use cases that the ACCORDION will be called to handle.

After the choice of the appropriate technology and configuration, a middleware layer will be added between the VIM and storage components in order to expose specific, role-based APIs that ensure security of the data, integrity of the system, optimized QoE for the users, fault proofing, fault tolerance, intelligent caching and other relevant functionalities.

The component will use the Kubernetes ecosystem by using the Kubernetes master as the storage controller, storage UI access point and Prometheus master for the specific cluster. Each node that is connected to the Kubernetes cluster has the potential of becoming a storage worker for this cluster or/and for the ACCORDION ecosystem. This is enabled by defining a custom label for the node, enrolling it as a storage worker. As a node, we define a Kubernetes node, which can be a PC, laptop, IoT device or any other compatible device. In order to be eligible for the role of storage worker, a node must have sufficient hard disk space available. The

amount of space is highly dependent on the use case, so it is not pre-configured. In the next figure (Figure 5), we can see a high-level architecture of the module with the interconnections between the sub-modules.

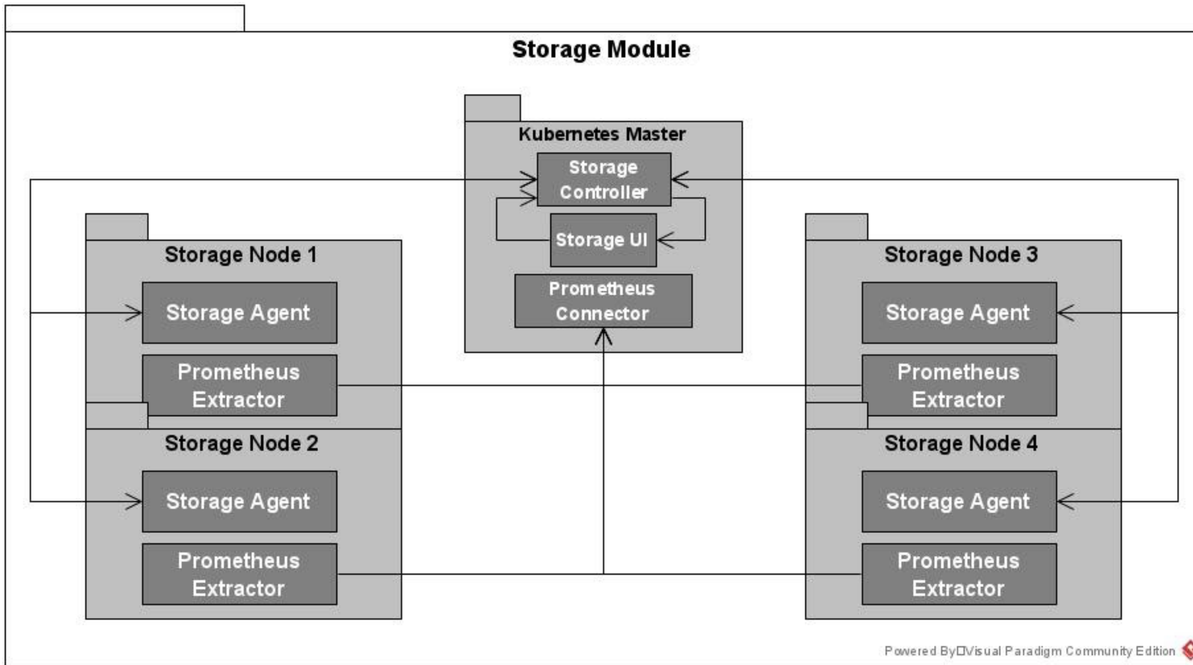

Figure 5: Edge Storage component architecture.

We have isolated four actors that are using the services of the Storage module; the VIM, the Prometheus Aggregator, the Mini-cloud Administrator and the ACCORDION Administrator. VIM will be using the APIs exposed by the component in order to perform automated or semi-automated processes or even expose the functionalities in other interfaces or components. A draft of the APIs that will be exposed by the component is included in Table 1 at the end of this chapter. The Prometheus Aggregator will access the endpoint provided by the cluster Prometheus master in order to scrape the data and collect them in an aggregated database that collects information from all the ACCORDION mini-cloud clusters. The Mini-cloud Administrator will be using the storage UI in order to manage the storage cluster and the data in it for administrative purposes. The ACCORDION Administrator will also be using the storage UI in order to manage and monitor the data and the cluster in accordance with the general ACCORDION needs. In the following UML (Figure 6), we can see a visual representation of these relations.

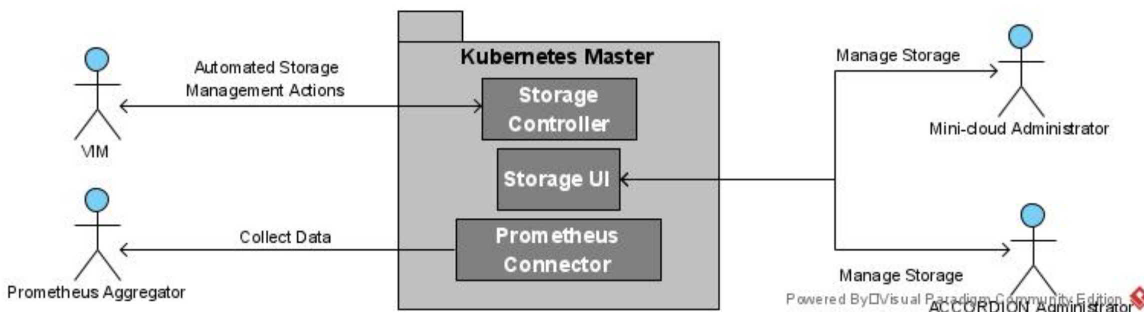

Figure 6: Edge storage use cases and actors.

2.5 Lightweight Virtualization

Traditionally, embedded devices have been centered around specialized, embedded processors and the embedded operating systems running on them (e.g., FreeRTOS and Zephyr). This model has been, and still is, extremely effective in ensuring efficient resource consumption, especially power, but forces developers to port applications to such OSes, since, for the most part, they are not POSIX-compliant.

Increasingly, many embedded devices are being designed around general-purpose processors, especially ARM-based ones. This is a radical shift in the way we think about embedded devices: many so-called “embedded” devices (e.g., in the IoT, edge, gateways and automotive domains) use Linux as default because it is easy to install, is POSIX-compliant and comes with a great array of applications and programming languages, not to mention a friendly, well-known development environment. The great downside is that this monolithic kernel is resource hungry: it is not unusual for a significant amount of a device’s resources to be consumed by Linux itself, leaving less for the actual application. Further, Linux is to a large extent a monolithic kernel, making it often hard or time consuming to customize (e.g., completely removing the scheduler if it is not needed, adding a new memory allocator, or trying to trim it down to reduce boot times). Finally, Linux’s significant code base (in the order of millions of lines of code) results in a large attack surface and exploits, and makes it expensive to certify in domains where safety is critical.

In order to break the dichotomy between (1) difficult-to-use but resource efficient embedded OSes and (2) power- hungry but user and application friendly general-purpose OSes such as Linux, we introduce a novel lightweight virtualization architecture and micro-library operating system called Unikraft which allows for automatically building highly specialized images for embedded devices. Unlike other approaches, Unikraft bridges the gap between resource efficiency and ease of porting with a micro-library approach, allowing for bottom-up specialization and code elimination while retaining POSIX compatibility. In addition, the extremely lean images it produces are ideal candidates for cheaper certification.

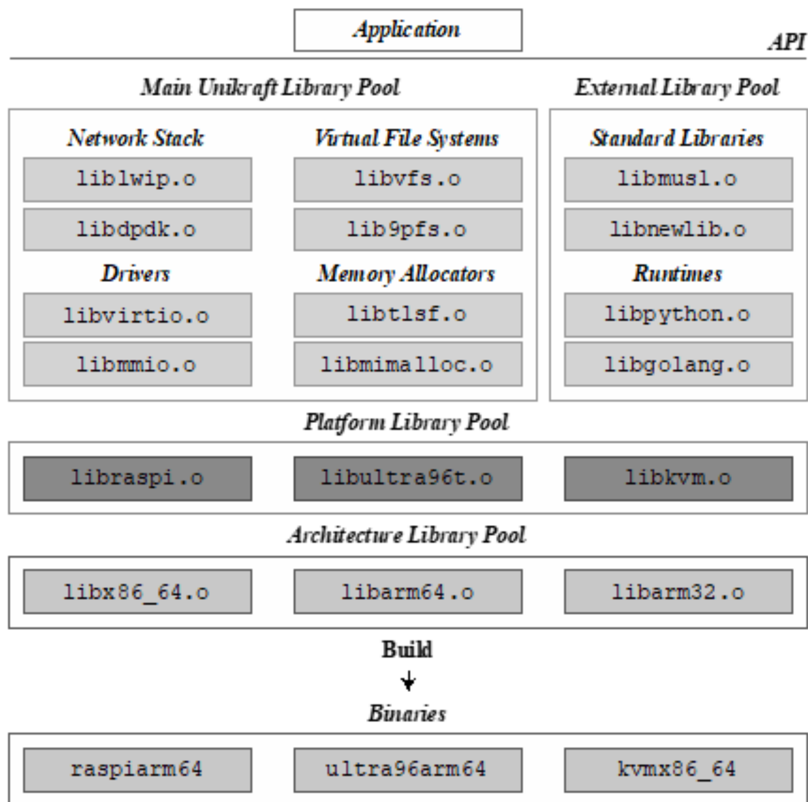

Figure 7: Unikraft's architecture. All components are micro- libraries. Users select an architecture, a platform, the target application and Unikraft creates the image.

Unikraft is fully librarized (see Figure 7): OS primitives such as the scheduler, memory allocator and even boot code are libraries. These can be removed or replaced with equivalents via a Kconfig menu as long as they comply with a set of well-defined APIs. To enable quick boot times, Unikraft can, for example, allow for the use of a simple but quick memory allocator during boot and initialization, whilst still using a different allocator for the application. Alternatively, a user could entirely remove the scheduler if not needed, and run tasks to completion in an event-driven architecture. This and many other scenarios are easy to implement in Unikraft because of its modular design.

Unikraft enables users to easily build extremely specialized, custom OSes without having to develop any actual code as it provides a POSIX-like interface which allows for running standard, off-the-shelf applications such as databases (e.g. SQLite), web servers (e.g. NGINX), key-value stores (e.g. Redis), machine learning frameworks (e.g. PyTorch and TensorFlow), and runtime language environments (e.g. Web Assembly, Python/Micropython, Lua and Ruby). In addition, Unikraft supports a number of compile-time languages including C/C++, Go, Java and Rust, with the potential for allowing different libraries to be written in different languages and combined together into a single, specialized image [4].

2.6 Edge Minicloud VIM

The Edge Minicloud Virtual Infrastructure Manager (VIM) is the core of a Minicloud and one of the main components of the ACCORDION architecture. The Virtual Infrastructure Manager is also one of the major

functional blocks of the ETSI MANO Network Functions Virtualization (NFV) architecture and, like in that architecture, it's responsible for controlling, managing, and monitoring the Infrastructure compute, storage, and network resources.

The tool selection process followed in this first year to find an open source reusable baseline for the VIM, which will be reported in more detail in section 7, resulted in identifying a lightweight Kubernetes (K8s) version named K3s[13] as a starting point.

K3s, like K8s, is good at handling containers, but to use also Virtual Machines (VM) as edge deployment units on top of the resource pool, K3s had to be extended. The tool selection process identified KubeVirt[14] as the best candidate for this extension. KubeVirt uses K8s Custom Resource Definitions (CRD) to offer an API that allows managing VMs in a K8s cluster through the same patterns used to manage containers.

The Edge Minicloud VIM architecture is therefore based on both K3s and KubeVirt, which will take control of the machines in the resource pool by running on them. At least one machine in the resource pool will be the K3s Master Node, and the others will be Worker Nodes. The K3s and KubeVirt components of both a Master and Worker Node are shown in Figure 8.

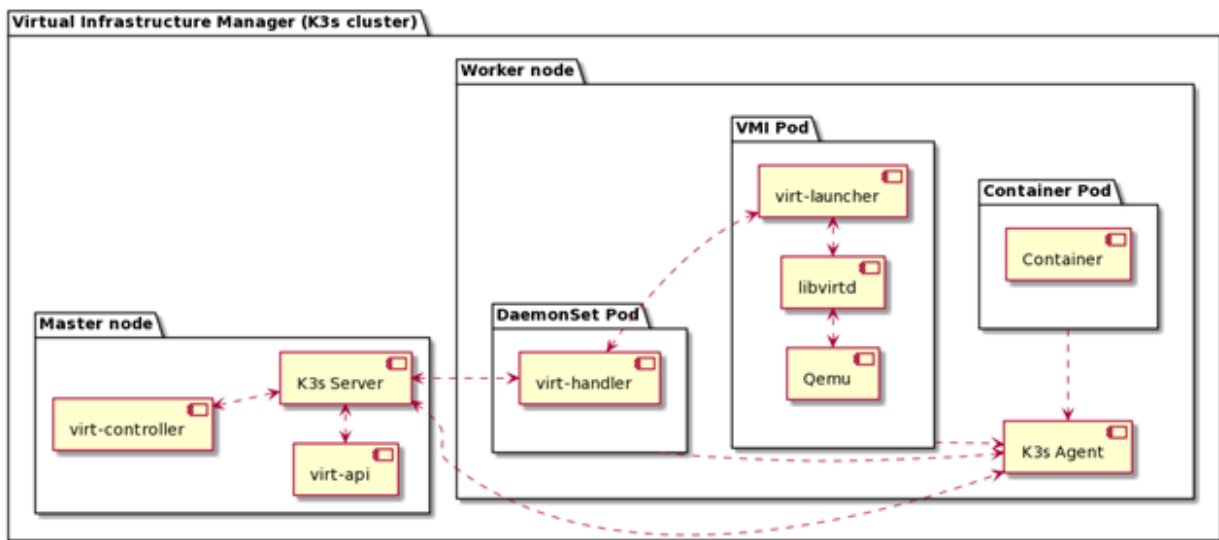

Figure 8: ACCORDION Minicloud VIM logical architecture, based on the K3 cluster architecture

The K3s architecture is quite simple, as it merged all K8s Master components (e.g., API Server, Scheduler, Controller Manager, etc.) into a single executable, the *K3s Server*, and all K8s Worker components in the *K3s Agent*. KubeVirt doesn't merge its components in the same way, so there will be separate components *virt-api* and *virt-controller* on the Master Node. KubeVirt runs virtual machines in special pods (objects of a new type called VMI) as container workloads, therefore in the Worker Node there will be multiple pods controlled

[13] https://k3s.io/

[14] https://kubevirt.io/

by the *K3s Agent*. One pod to run the *virt-handler* component that launches the VMI and configures it until it matches the required state; one VMI pod for each VM including the components *virt-launcher*, *libvirtd* and *Qemu*; and one or more pods to host containers.

Figure 8 is a logical diagram showing a single instance for each type of component, but a real deployment may have a more complex configuration. For example, multiple Master Nodes (usually three) are needed to implement High Availability (HA) configurations. Multiple Worker Nodes are also the norm in a K8s/K3s cluster, as they are those configured to run the real application workload. Some Workers may run only containers, some others only VMs and others may run both VMs and containers. Container Pods will be present only in Workers running containers, and DaemonSet plus VMIs pods will be present only in Workers running VMs.

In Figure 8 there is no ACCORDION-specific component, as the initial idea is to implement all the needed logic, such as deciding which node to use to run a given container or VM, through K8s-native configuration.

3 Resource monitoring & characterization

3.1 Description & Objectives

The resource monitoring & characterization component has to monitor and characterize the resources as other components will query for information to perform actions based on metrics. Monitoring and characterization won't have a central component, instead it will be done on the Edge. Monitoring is going to be dynamic on contrary characterization is static, as it will have information for the hardware of the Edge devices.

The goal is to have a component that can monitor both physical (devices) and virtual layer (VMs, containers, pods). In addition, this component is cloud agnostic as it does not need a specific provider to perform monitoring or characterization. Between Edge networks, we may have heterogenous devices with different computational power, as Prometheus server is actually a pod and char-agents are containers that they can run on any device that has K3s and Docker.

3.2 Requirements

Monitoring component runs in a K3s cluster, each node of the cluster must have Docker installed. For the monitoring part Cluster Monitoring stack for ARM / X86-64 was used as it produces the configuration for Prometheus, exporters and Grafana on K3s. To automate the deployment of the whole monitoring component some Python and shell scripts were developed. A device to run those scripts needs to have Python 3.6[15] installed. A characterization agent is a custom Docker container that is developed in Python. To retrieve the GPU information container uses mesa-utils[16] and to be more specific from the glxinfo. As Mesa is a 3D computer graphics library it has to get the model of the GPU from the host, some configurations must be done before running the container:

1. non-network local connections have to be added to access control list by running xhost local:root command
2. set display with export DISPLAY=':0.0'
3. the final step is to run the container with the Display parameter and network set to host to get the IP of its host docker run --rm -it --privileged -e DISPLAY=$DISPLAY --network=host char-agent

In the experiments, two Raspberry Pi 4B and on PC were used as K3s supports ARM and AMD architectures. In the case of Raspberry Pi 4B, there was an extra configuration that needed to be made for the case of char-agent. If a screen was not connected to the Raspberry Pi 4B it did not initiate the GUI, so the container failed

[15] https://www.python.org/downloads/release/python-360/

[16] https://wiki.debian.org/Mesa.

to get the GPU details to avoid this error a static screen resolution must be configured so when device boots GUI also boots.

3.3 Research Challenges & Advancements Achieved

As it was mentioned in D2.2, a challenge for monitoring is to be able to connect the information that is provided from the physical and virtual layer, so it would be able to describe in a comprehensive way how a problem in the physical layer can impact the virtual layer. In Prometheus, node-exporter, Prometheus Adapter and kube-state-metrics are configured which expose metrics for physical and virtual monitoring so we could develop PromQL queries that give us the CPU usage of the device but also the CPU usage that a specific K3s pod has.

By using Prometheus for monitoring purposes, we can also monitor Cloud VMs without using cloud providers' monitoring tools. The monitoring stack was deployed also on a K3s cluster with a Okeanos Cloud VM and 2 other RaspberryPi for Edge storage experiments and Prometheus could successfully monitor the VM without changing the configuration or using a Cloud provider monitoring tool. Heterogeneity was another challenge, as we said between edge networks, we may have a variety of devices. With experimentations on the K3s clusters we concluded that we can perform monitoring and characterization VMs, PCs and Raspberry Pis without having a huge impact on them.

For the characterization we tried to find an open standard or a model to describe bare metal devices. TOSCA was chosen as with it we could write YAML files that are commonly used. TOSCA was extended to represent the clusters and its devices.

3.4 Provided Features and APIs

The monitoring API will provide other ACCORDION components monitoring and characterization information. As it is a REST API, the format of the results is JSON. For monitoring information, the path for the calls is */monitoring* and the name of the supported HTTP parameter is metric. An ACCORDION component that needs to consume monitoring information about the physical layer will have to declare in the metrics parameter the value physical_metrics as the example below and the result is shown in Figure 9.

http://0.0.0.0:3000/monitoring?metric=physical_metrics

An endpoint that provides the monitoring metrics for bare metal and VMs of the cluster (CPU usage, RAM usage, disk write and read latencies, filesystem usage, disk size, disk free space disk IO).


```
{"Results": [{"timestamp": 1613918256.884719, "Cpu Usage Results": [{"node": "giannis", "cpu_usage(percentage)": " 11.39"}, {"node": "raspberrypi", "cpu_usage(percentage)": " 6.08"}, {"node": "raspberrypi1", "cpu_usage(percentage)": " 6.80"}]}, {"timestamp": 1613918256.948491, "Memory Usage Results": [{"node": "giannis", "mem_usage(percentage)": " 43.58"}, {"node": "raspberrypi", "mem_usage(percentage)": " 87.23"}, {"node":
```

```
"raspberrypi1", "mem_usage(percentage)": " 75.87"}]}, {"timestamp": 1613918257.013779, "Disk Write Latency Results": [{"node": "giannis", "device": "sdc", "disk_write_latency(percentage)": " 0.80"}, {"node": "giannis", "device": "sdd", "disk_write_latency(percentage)": " 1.22"}, {"node": "raspberrypi", "device": "mmcblk0", "disk_write_latency(percentage)": " 1.20"}, {"node": "raspberrypi", "device": "mmcblk0p2", "disk_write_latency(percentage)": " 1.20"}, {"node": "raspberrypi1", "device": "mmcblk0", "disk_write_latency(percentage)": " 18.77"}, {"node": "raspberrypi1", "device": "mmcblk0p2", "disk_write_latency(percentage)": " 18.77"}]}, {"timestamp": 1613918257.075608, "Disk Read Latency Results": [{"node": "giannis", "device": "sdc", "disk_read_latency(percentage)": " 0.02"}]}, {"timestamp": 1613918257.137085, "Filesystem Usage Results": [{"node": "giannis", "mountpoint": null, "fstype": null, "filesystem_usage(percentage)": " 19.49"}, {"node": "raspberrypi", "mountpoint": null, "fstype": null, "filesystem_usage(percentage)": " 74.97"}, {"node": "raspberrypi1", "mountpoint": null, "fstype": null, "filesystem_usage(percentage)": " 74.55"}]}, {"timestamp": 1613918257.198618, "Disk Size Results": [{"node": "giannis", "disk_total_size(bytes)": "2204823101440"}, {"node": "raspberrypi", "disk_total_size(bytes)": "17658145792"}, {"node": "raspberrypi1", "disk_total_size(bytes)": "17658145792"}]}, {"timestamp": 1613918257.258936, "Disk Free Space Results": [{"node": "giannis", "disk_free_space(bytes)": "2170876522496"}, {"node": "raspberrypi", "disk_free_space(bytes)": "6878654464"}, {"node": "raspberrypi1", "disk_free_space(bytes)": "6956704256"}]}, {"timestamp": 1613918257.318943, "Disk IO Usage Results": [{"node": "giannis", "disk_io_time_spent(seconds)": "0.007369490285714287"}, {"node": "raspberrypi", "disk_io_time_spent(seconds)": "0.00036869494949545245"}, {"node": "raspberrypi1", "disk_io_time_spent(seconds)": "0.08465387434343274"}]}]}
```


Figure 9. Node Monitoring

In the case of virtual layer monitoring the parameters that are currently supported are pods CPU usage, pods memory usage, pods status phase and pods info. To get the results of the pod (virtual layer) metrics of a specific application one has to declare the HTTP parameter metric and the HTTP parameter namespace. The values of the HTTP parameter namespace are the available namespaces of the K3s cluster. The result is shown in Figure 10.

http://0.0.0.0:3000/monitoring?metric=virtual_metrics&namespace=application


```
{"Results": [{"timestamp": 1613918536.169836, "Pod Info Results": [{"pod": "mysql-bfc5c9f44-477rv", "pod_ip": "192.168.1.2", "namespace": "application", "created_by_kind": "ReplicaSet", "replica": "1", "node": "giannis", "node_ip": "192.168.1.2"}, {"pod": "wordpress-86885f548-44m8z", "pod_ip": "10.42.0.46", "namespace": "application", "created_by_kind": "ReplicaSet", "replica": "1", "node": "giannis", "node_ip": "192.168.1.2"}]}, {"timestamp": 1613918536.238763, "Kube Pod Status Phase Results": [{"kube_pod_status_phase": "Running", "pod": "mysql-bfc5c9f44-477rv", "instance": "10.42.0.42:8443",
```

```
"namespace": "application"}, {"kube_pod_status_phase": "Running", "pod": "wordpress-86885f548-44m8z", "instance": "10.42.0.42:8443", "namespace": "application"}]}, {"timestamp": 1613918536.300128, "Pod CPU Usage Results": [{"node": "giannis", "pod": "mysql-bfc5c9f44-477rv", "pod_cpu_usage(seconds)": "0.018315272811091667"}, {"node": "giannis", "pod": "wordpress-86885f548-44m8z", "pod_cpu_usage(seconds)": "0.004569118139741808"}]}, {"timestamp": 1613918536.368414, "Pod Memory Usage Results": [{"node": "giannis", "pod": "mysql-bfc5c9f44-477rv", "pod_memory_usage(bytes)": "20627632.01304764"}, {"node": "giannis", "pod": "wordpress-86885f548-44m8z", "pod_memory_usage(bytes)": "2882807.349989445"}]}]}
```


Figure 10. Pod Monitoring

Monitoring can also provide the K3s namespaces of the cluster and their cpu and memory usage. In this case the HTTP namespace parameter has to be declared as all, the result is shown in the Figure 11.

http://0.0.0.0:3000/monitoring?namespace=all


```
{"Results": [{"timestamp": 1613918683.353328, "Namespace CPU Results": [{"namespace": "monitoring", "cpu_usage(seconds)": "0.1492824091256434"}, {"namespace": "kube-system", "cpu_usage(seconds)": "0.012014744984433889"}, {"namespace": "application", "cpu_usage(seconds)": "0.0007409287284330483"}]}, {"timestamp": 1613918683.416821, "Namespace Memory Results": [{"namespace": "kube-system", "memory_usage(bytes)": "232685568"}, {"namespace": "monitoring", "memory_usage(bytes)": "1953517568"}, {"namespace": "application", "memory_usage(bytes)": "1104633856"}]}]}
```


Figure 11. Namespace Monitoring

The path for the characterization results in API is */characterization* and the supported HTTP parameter is the format. The result can be returned in JSON with REST calls or as a TOSCA YAML downloadable file.

http://0.0.0.0:3000/characterization?format=json/tosca


```
[{"device": {"_id": {"$oid": "60326f8f528bf960d6b36ce3"}, "device_name": "raspberrypi", "ip": "192.168.1.205", "UUID": "e7cd9caa-7451-11eb-85aa-dca632298c4f", "RAM(bytes)": 4095737856, "Battery": "None", "CPU": {"Arch": "armv7l", "bits": "32", "cores": 4}, "GPU": {"GPU_name": "llvmpipe (LLVM 7.0, 128 bits) (0xffffffff)", "GPU_type": "Intergated graphics processing", "GPU_video_memory(bytes)": 4095737856, "unified_memory": "no"}, "OS": {"OS_name": "Linux", "OS_version": "4.19.118-v7l+"}, "DISK": [{"device": "/dev/root", "fstype": "ext4", "mountpoint": "/dev/termination-log"}, {"device": "/dev/root", "fstype": "ext4", "mountpoint": "/etc/resolv.conf"}, {"device": "/dev/root", "fstype": "ext4", "mountpoint": "/etc/hostname"}, {"device": "/dev/root", "fstype": "ext4", "mountpoint": "/etc/hosts"}], "K3s": {"node_role": ""}, "Region": {"continent": "Europe", "country": "Greece", "city": "Athens"}}}, {"device": {"_id": {"$oid": "60326fa4528bf960d6b36ce4"}, "device_name": "raspberrypi1", "ip": "192.168.1.203", "UUID": "e7344550-7451-11eb-9473-dca632299078", "RAM(bytes)": 4095737856, "Battery": "None", "CPU": {"Arch": "armv7l", "bits": "32", "cores": 4}, "GPU": {"GPU_name": "llvmpipe
```

(LLVM 7.0, 128 bits) (0xffffffff)", "GPU_type": "Intergated graphics processing", "GPU_video_memory(bytes)": 4095737856, "unified_memory": "no"}, "OS": {"OS_name": "Linux", "OS_version": "4.19.118-v7l+"}, "DISK": [{"device": "/dev/root", "fstype": "ext4", "mountpoint": "/dev/termination-log"}, {"device": "/dev/root", "fstype": "ext4", "mountpoint": "/etc/resolv.conf"}, {"device": "/dev/root", "fstype": "ext4", "mountpoint": "/etc/hostname"}, {"device": "/dev/root", "fstype": "ext4", "mountpoint": "/etc/hosts"}, {"device": "/dev/root", "fstype": "ext4", "mountpoint": "/dev/termination-log"}, {"device": "/dev/root", "fstype": "ext4", "mountpoint": "/etc/resolv.conf"}, {"device": "/dev/root", "fstype": "ext4", "mountpoint": "/etc/hostname"}, {"device": "/dev/root", "fstype": "ext4", "mountpoint": "/etc/hosts"}], "K3s": {"node_role": ""}, "Region": {"continent": "Europe", "country": "Greece", "city": "Athens"}}}, {"device": {"_id": {"$oid": "60326fa4528bf960d6b36ce5"}, "device_name": "giannis", "ip": "192.168.1.2", "UUID": "ea831f38-7451-11eb-8bc7-fcaa149d94de", "RAM(bytes)": 8396820480, "Battery": "None", "CPU": {"Arch": "x86_64", "bits": "64", "cores": 6}, "GPU": {"GPU_name": "AMD BONAIRE (DRM 2.50.0, 4.15.0-135-generic, LLVM 7.0.1) (0x665f)", "GPU_type": "Dedicated graphics processing", "GPU_video_memory(bytes)": 2147483648, "GPU_total_available_memory(bytes)": 4289724416, "unified_memory": "no"}, "OS": {"OS_name": "Linux", "OS_version": "4.15.0-135-generic"}, "DISK": [{"device": "/dev/root", "fstype": "ext4", "mountpoint": "/dev/termination-log"}, {"device": "/dev/root", "fstype": "ext4", "mountpoint": "/etc/resolv.conf"}, {"device": "/dev/root", "fstype": "ext4", "mountpoint": "/etc/hostname"}, {"device": "/dev/root", "fstype": "ext4", "mountpoint": "/etc/hosts"}, {"device": "/dev/root", "fstype": "ext4", "mountpoint": "/dev/termination-log"}, {"device": "/dev/root", "fstype": "ext4", "mountpoint": "/etc/resolv.conf"}, {"device": "/dev/root", "fstype": "ext4", "mountpoint": "/etc/hostname"}, {"device": "/dev/root", "fstype": "ext4", "mountpoint": "/etc/hosts"}, {"device": "/dev/sdc1", "fstype": "ext4", "mountpoint": "/dev/termination-log"}, {"device": "/dev/sdc1", "fstype": "ext4", "mountpoint": "/etc/resolv.conf"}, {"device": "/dev/sdc1", "fstype": "ext4", "mountpoint": "/etc/hostname"}, {"device": "/dev/sdc1", "fstype": "ext4", "mountpoint": "/etc/hosts"}], "K3s": {"node_role": "control-plane,master"}, "Region": {"continent": "Europe", "country": "Greece", "city": "Athens"}}}]

Figure 12. Characterization Results

Beside the monitoring, this Task has to characterize resources based on their hardware. To be able to characterize resources every node of a cluster has to host a char-agent container which identifies the characteristics of the device and exposes them via an API in JSON format. The master node of a K3s cluster is the one who collects the information from the characterization-agents and stores them in a MongoDB database.

The components of the Tasks that are shown in Table 1 will get monitoring and characterization information from the monitoring API. Monitoring API will support HTTP parameters and based on the path it will return the related information. Examples of usage are shown in the above Figures.

Table 1: Monitoring data needed per ACCORDION Task.

Tasks	Required Information from Monitoring
Task 3.2 Resource indexing & discovery	Per-node information: • Static Hardware: amount of CPU,ram,disk, etc. • Static Software: installed software (from a list of selected) and relative versions • Dynamic hardware: load, ram consumptions, etc.. (average on 5-10-15 minutes?)
Task 3.3 Edge Storage	For evaluation: • System statistics such as disk usage, ram usage and network usage • Node health statistics about failing or disconnecting nodes and drives. During runtime: • Node health statistics about failing or disconnecting nodes and drives.
Task 5.5. NextGen application development toolkit	All (or as many as possible) APIs will be used to display information to the user, maybe a Grafana or similar program will be hosted in an I-frame in order to display graphical information directly from the Prometheus distributions.

3.5 Technical Challenges and Mitigation

A technical challenge that we encountered was that after the deployment of the Prometheus server pod, we found that if someone knows the IP of the K3s master or/and the ingress IP developed by the deployment files (prometheus.master-ip.nip.io) he/she can bypass the monitoring API and perform PromQL queries on the GUI or API of Prometheus.

To avoid this problem, we had to add an authentication layer to Prometheus. Prometheus on its own does not have an authentication mechanism, a reverse proxy needs to be used to have a security layer. The reverse proxy that we use is Traefik[17] as it is downloaded with K3s by default. To secure the endpoints of Prometheus,

[17] https://doc.traefik.io/traefik/

we performed a random password generation for the monitoring the user, both credentials stored in a htpasswd[18] file which was used later on a shell script to create a secret in the monitoring namespace. By adding in the Prometheus ingress configuration file, the reverse proxy, the authentication type and the secret in the annotations segment the authentication was added successfully to all Prometheus endpoints. The credentials were also stored in an encrypted YAML file and in another file the key to decrypting the credential file was stored. This is achieved with the usage of Python's library cryptoyaml[19], which uses the Fernet symmetric encryption. The only component that needs to know the decryption key and have access to credentials YAML file is the monitoring API, every other component won't have access to Prometheus endpoints.

3.6 Future Work

For future work, we have to rename some JSON objects and arrays in the JSON response of the monitoring API. In addition, the names of the HTTP parameters have to change into simpler ones, also we have to interact with the other WPs to see if there is a need for new endpoints in the monitoring API that provide different metrics.

We have already developed some installation scripts for monitoring to automate the deployment and installation of Prometheus and Grafana to the K3s master. The next would be to find a way to deploy characterization agents as a set of pods and containerize the monitoring API to finalize the automation of deployment the monitoring component. Another part of installation scripts would be the installation of the required libraries to perform the actions that are described in the Technical Challenges and Mitigation Section.

[18] https://httpd.apache.org/docs/2.4/programs/htpasswd.html

[19] https://pypi.org/project/cryptoyaml3/

4 Resource indexing & discovery

4.1 Description & Objectives

The aim of the Resource indexing & discovery (RID) component is to support and assist the other system components. The computational resources available to the ACCORDION nodes inside the Minicloud are subject to continuous change due to application activities or the migration of applications between Miniclouds. As a matter of fact, in order to effectively orchestrate the available resources, it is highly important to rely on the current load state of the components.

In this context, the role of the RID component is twofold: on one side to keep updated the state of computational resources distributed between various Miniclouds in the system and on the other side the RID component should provide the functionality for the effective retrieval of information about available resources. The main sources of the data for the RID is the Resource monitoring & characterization component, described in Section 3. Nevertheless, the queries can in principle come from any ACCORDION component that needs to find resources with specific computational features, in particular the intelligent Orchestrator (See deliverables D4.1 and D4.2).

4.2 Requirements

The role of the RID component inside the system is twofold. On the one side, the RID component has to keep an updated state of the available computational resources in the system. On the other side, the RID component should provide an effective functionality for the system state information retrieval. Hence in order to provide the high-quality service the RID should address the following requirements:

- Provide functionality for storing and retrieving information about the dynamically changing resources in the system. The functionality should be done in a scalable manner not only in terms of amount of data but also in terms of the variety of the information to retrieve.
- RID component should be able to address the user provided requirements for the resource discovery. For example, the requirements in performance and precision of resource discovery.
- The component should be integrated with resource and queries ontologies presented in the Resource monitoring & characterization component description (see Section 3.4).

4.3 Research Challenges & Advancements Achieved

Recent years witness the development and use of the technologies for large-scale Heterogeneous infrastructures such as Grids [5], Clusters [6], Clouds [5]. These large-scale distributed computing environments (LDCE) technologies allow us to rely not only on a single available resource but on the use of different heterogeneous physically spread resources in the network. Indeed, one of the most challenging issues for these technologies is an effective way to perform resource discovery. Resource discovery is an essential part of effective resource utilization in LDCE. It includes locating, retrieving and advertising of the

available resources in the system. However, considering the distributed and large-scale nature of LDCE, resource discovery technologies in such environments have to address well-known challenges for distributed systems, like efficiency, scalability and reliability [7].

The efficiency of a resource discovery solution can be mostly described as a combination of the conceptual metrics latency and load balancing [7]. The latency of resource discovery is built up by different delay sources: network transfer time, query processing algorithm, the complexity and type of the query, network size and the level of the computing parallelism in the system. The other metric that characterises the efficiency of resource discovery is the load balancing for the discovery procedure. In other words, how the query load is distributed among resource information providers.

Among the various proposals, MatchTree [8] is a P2P-based approach that reduces query response times with redundant query topologies, dynamic timeout policies and sub-region queries. It also balances processing overheads between resources. OntoSum [9] relies on a semantic-aware topology construction method for resource discovery in Grids. The method significantly improves the discovery efficiency by propagating the discovery requests only between semantically related nodes. SWORD [10] is a P2P DHT-based approach that relies on multiple overlays. The load-balancing mechanism of SWORD is less efficient in the case of non-uniform distribution of the nodes in the environment. Node-Wiz [11] is a hybrid approach. It is load-balanced and supports clustering and self-organization for overlays. It relies on a single distributed indexing mechanism.

Scalability of resource discovery characterises the ability of the system to react and address the variations in the system in terms of the amount of resource involved. Scalability is one of the key factors for evaluation of the distributed systems. The highest impact on the scalability of the system is made by the size of the network. The increase in the system size increases the communications delays and overheard and leads to an increase in the response time for the discovery queries. On one side in order to create an effective scalable system the one should avoid relying on centralized architecture with the risks of resource bottleneck and single point of failure. On the other side the fully distributed architecture has its own drawbacks in terms of bootstrapping and the increasing response time for the querying. Zarrin et al. [7] demonstrate that Grid and P2P based approaches like OntoSum and MatchTree provide better scalability than the hybrid ones like NodeWiz. These conclusions also come from the fact that both OntoSum and MatchTree approaches are decentralized and hence have more potential to provide scalable solutions. However, the distributed solutions require a higher level of communication that can impact on efficiency.

In order to achieve the desired level of scalability we plan to apply hybrid distribution strategies for resource discovery. We particularly concentrate on the investigation of the approaches that best mix structured and unstructured data structures techniques in order to fulfil the issue of scalability in the most effective way.

Reliability of resource discovery can be described by the following metrics: its accuracy, validity, dynamicity and faults tolerance. The accuracy addresses the correctness of the resource description and querying in the system. For the high accuracy of resource discovery, the applied resources description and query models should correspond to the available resources in the system. At the same time the description model as well as the resource querying should satisfy more than just a single match or single resource querying. For example, MatchTree algorithm can guarantee query completeness [8]. One more factor of reliability is

dynamicity and fault tolerance of the system. The desired reliability should tackle the leaving and joining nodes in the system. Resource discovery should provide solutions for dealing with possible failures of the resource holders. For example, SWORD and Node-Wiz protocols support dynamic topologies and attributes. MatchTree supports fault tolerance in terms of churn rate of nodes and internal node failure. It also addresses dynamicity in terms of adding a new attribute. OntoSum also supports dynamicity in terms of probability to issue a query, probability to leave the system and the probability of new nodes/resources to join the system [7]. The validity of the resource discovery system is of particular interest to ACCORDION. In order to build an effective distributed architecture, it is highly important that the resource discovered information is valid and updated.

4.4 Provided Features and APIs

Each RID component proposes a http interface. This interface allows other components of the ACCORDION system to request information about computational resources via queries. The current implemented version of the interface supports the queries in JSON format. The queries are sent via 'POST' requests. The results of the queries are returned in JSON format.

The current version of the RID component accepts the following types of hte queries:

1. Multi attribute range queries on numerical values, e.g., available RAM.
2. Exact queries for string values, e.g., specific GPU chipset names.
3. Boolean query for specific feature availability, e.g., availability of a GPU.

The supported types of queries are subject to changes with the development of the ACCORDION project. New types of queries can be added or the existing ones can be changed based on the future requirements of the ACCORDION system components.

4.5 Technical Challenges and Mitigation

Due to the limited progress of the project in the current state, it is not possible to test the distributed solutions for RID components. Most of the planned components of the ACCORDION system are in the development phase and, for now, cannot be involved in the emulation and evaluation of the possible distributed solutions for the RID. Due to these temporal limitations and in order to proceed with the project plans for the RID component we plan to start the evaluation of the available solutions for the RID based on the simulation techniques.

4.6 Future Work

In the short-term future plans, we are considering simulating the distributed solutions studies for RID components. At the same time, with the progress of the development of other system components we plan to continue the integration between the RID and other system entities.

5 Edge storage

5.1 Description & Objectives

The Edge storage component aims to provide an edge storage framework, supporting the QoE needs of the users, be it stakeholders, developers, applications or any other interesting person or software. This is achieved by optimizing resource usage, allocation and data management plans on edge devices per Minicloud. A clearer description of objectives can be derived from the use case requirements.

In detail, this component needs to provide a reliable, fast, stable and secure shared storage engine, accessible by all devices and users of a Minicloud using a role-based security schema. This engine needs to be extremely lightweight since it is created for edge devices with extremely limited resources, like Raspberry Pies or other micro-computer devices. It also needs multiple access points depending on the role and category of the client that needs to access or modify the data. For example, a human administrator would like to have a web-based GUI in order to manage the stored data or the storage engine itself. A software client on the other hand would like to have either an API or a mounted file path in order to manage the data. The edge storage component needs to be provided both in a uniform and easy-to-understand manner.

5.2 Requirements

The requirements for the Edge Storage component can be separated into three categories; the general requirements, which include general purpose requirements that all data storage systems have, the algorithmic requirements, which concern the usage of the storage system by the ACCORDION and its hosted applications, and the data administrator requirements, which concern the usage, management and configuration of the system by an administrator. All these requirements are analysed in detail in the following sub-sections.

5.2.1 General Requirements

- Small enough data retrieval latency to cover the needs of all ACCORDION use cases.
- High availability by employing fault tolerance and mitigation methods.
- Hight integrity by employing integrity validation methods.
- Role based security allowing access to all authorized users.

5.2.2 Algorithmic Access Requirements

- Singular purpose endpoints with clearly defined APIs.
- Emulated filesystem support for applications that require it.
- Automated authorization methods enabling role-based access to authorized systems.
- Fault proofing for large batch processes.
- Atomicity to some degree according to the needs of the application.

5.2.3 Data Administrator Requirements

- Easy to learn and use web-based GUI.
- Strict authorization and security mechanisms.
- Easy to understand and explore monitoring data about the data storage platform and the hosted data.

5.3 Research Challenges & Advancements Achieved

The open research issues in the area of edge storage platforms revolves around three main pillars; the minimization of overhead when transferring large quantities of small data packets, the intelligent admission mechanisms that allow an optimized pre-fetching of useful data and intelligent caching that minimizes the network traffic while optimizing the bound resources of edge devices. More details on these research issues can be found in the more extended description present in D2.2.

The first year of the project was aimed at creating a reliable platform, supporting the requirements of the ACCORDION use cases and enabling us to run experiments and gather data about the functionality of the platform, the bottlenecks and the points that need optimization, which has been achieved using the K3S distribution of kubernetes and customized deployments based on the Prometheus monitoring system as well as the MinIO and OpenStack datastores. This platform will serve as a basis for advancing the scientific goals set by the challenges mentioned.

5.4 Provided Features and APIs

The current version of the edge storage component is based on MinIO which provides both a web-based GUI and an AWS S3 compatible API library. Also, Prometheus comes bundled with Graphana which provides an excellent web-based GUI for displaying the highly customized datasets that Prometheus gathers when monitoring the edge storage component and the machines involved in it. A number of wrapping and higher-level customized endpoints are planned for the next version of the component, covering more specific needs of the ACCORDION use cases.

5.5 Technical Challenges and Mitigation

Technical challenges concern machine faults, common to edge devices, such as overheating or unstable network access. These challenges are one of the motivations of creating a customized edge storage component instead of using a generic solution so their mitigation is covered under the fault tolerance and mitigation requirement of the component.

5.6 Future Work

For the second year of the ACCORDION project, we aim at tackling at least one of the open research issues, creating an innovative prototype, customized and optimized for the ACCORDION use case needs. There will also be more endpoints that serve more specialized needs of the use cases, wrapping and grouping some of the existing endpoints or creating entirely new ones.

6 Lightweight virtualization

6.1 Description & Objectives

Specialization is arguably the most effective way to achieve outstanding performance, whether it is for achieving high throughput in network-bound applications, making language runtime environments more efficient, or providing efficient container environments, to give some examples. Even in the hardware domain, and especially with the demise of Moore's law, manufacturers are increasingly leaning towards hardware specialization to achieve better performance; the machine learning field is a primary exponent of this.

In the virtualization domain, unikernels are the golden standard for specialization, showing impressive results in terms of throughput, memory consumption, and boot times, among others. Some of those benefits come from having a single memory address space, and thus eliminating costly syscall overheads, but many of those are the result of being able to hook the application at the right level of abstraction to extract best performance: for example, a web server aiming to service millions of requests per second can access a low-level, batch-based network API rather than going with the standard but slow socket API. Such an approach has been taken in several unikernel projects but often in an ad hoc, build-and-discard manner, and despite their clear benefits, unikernels suffer from two major drawbacks:

- They require significant expert work to build and to extract high performance; such work has to for the most part be redone for each target application.
- They are often non-POSIX compliant, requiring porting of applications and language environments.

We argue that these drawbacks are not fundamental, and propose a unikernel architecture built specifically to address them. Existing unikernel projects, even those based on library architectures, tend to consist of small but mono- lithic kernels that have complex, intertwined and sometimes opaque APIs for their components. This means that developers not only have to often port applications to such systems, but that optimizing their performance requires digging into the code and the specifics of the (uni)kernel in order to understand how to best obtain performance gains.

Furthermore, such systems typically rely on size-based specializations: removing all unnecessary components to achieve minimal images. While this strategy already offers significant benefits, we argue that unikernels based on library architectures should ease access to true specialization, allowing users to choose the best system component for a given application, environmental constraints, and key performance indicators

We propose Unikraft, a novel micro-library operating system targeted at painlessly and seamlessly generating specialized, high performance unikernels. To do so, Unikraft relies on two key principles:

- The kernel should be fully modular in order to allow for the unikernel to be fully and easily customizable. In Unikraft, OS primitives such as memory allocators, schedulers, network stacks and early boot code are stand-alone micro-libraries.

- The kernel should provide a number of performance- minded, well-defined APIs that can be easily selected and composed in order to meet an application's performance needs. In Unikraft, such APIs are micro-libraries themselves, meaning that they can be easily added or removed to a unikernel build, and that their functionality can be extended by providing additional such micro-libraries.

In brief, the key conceptual innovation of Unikraft is defining a small set of APIs for core OS components that makes it easy to replace-out a component when it is not needed, and to pick-and-choose from multiple implementations of the same component when performance dictates. The APIs have been built to enable performance (e.g., by supporting batching by design) and minimality in mind (no unneeded features).

To support a wide range of applications, we port the musl libc library, and provide a syscall shim layer micro-library. As a result, running an application on Unikraft can be as simple as building it with its native build system, and linking the resulting object files back into Unikraft.

6.2 Requirements

Before deriving what the key design principles for Unikraft are, it is worth analyzing the features and (heavyweight) mechanisms of traditional OSes that are unnecessary or ill- suited to single application use cases:

- Protection-domain switches between the application and the kernel might be redundant in a virtualization context because isolation is ensured by the hypervisor, and result in measurable performance degradation.
- Multiple address spaces may be useless in a single application domain, but removing such support in standard OSes requires a massive reimplementation effort.
- For RPC-style server applications, threading is not needed, with a single, run-to-completion event loop sufficing for high performance. This would remove the need for a scheduler within the VM and its associated overheads, as well as the mismatch between guest and hypervisor schedulers.
- For performance-oriented UDP-based apps, much of the OS networking stack is useless: the app could simply use the driver API, much like DPDK-style applications already do. There is currently no way to easily remove the network stack from standard OSes.
- Direct access to NVMe storage removes the need for a VFS layer and any actual filesystem, but removing filesystem support from existing OSes is very difficult.
- Memory allocators have a large impact on application performance, and general purpose allocators have been shown to be suboptimal for many apps. It would therefore be ideal if each app could choose its own allocator; this is however very difficult to do in today's operating systems because the allocators that kernels use are baked in.

This admittedly non-exhaustive list of application-specific optimizations implies that for each core functionality that a standard OS provides, there exists at least one or a few applications that do not need it.

Removing such functionality would reduce code size and resource usage but would often require an important re-engineering effort.

The problem we want to solve is to enable developers to create a specialized OS for every single application to ensure the best performance possible, while at the same time bounding OS-related development effort and enabling easy porting of existing applications. This analysis points to a number of key requirements:

- Single address space: The focus is on single application scenarios, with possibly different applications communicating with each other through networked communications.
- Fully modular system: All components, including operating system primitives, drivers, platform code and libraries should be easy to add and remove as needed; even APIs should be modular.
- Single processor mode: No user-/kernel-space separation to avoid costly processor mode switches. This does not preclude compartmentalization (e.g., of micro- libraries), which can be achieved at reasonable cost.
- Static linking: enables compiler features such as Dead Code Elimination (DCE) and Link-Time Optimization (LTO) to automatically get rid of unneeded code.
- POSIX support: In order to support existing or legacy applications and programming languages while still allowing for specialization under that ABI.
- Platform abstraction: Seamless generation of images for a range of different hypervisors/VMMs.

6.3 Research Challenges & Advancements Achieved

Given the requirements above, the question is how to implement such a system: by minimizing existing general-purpose operating systems, by starting from existing unikernel projects, or from scratch.

Existing work has taken three directions in tackling this problem. The first direction takes existing OSes and adds or removes functionality. Key examples add support for a single address space and remove protection domain crossings: OSv [12] and Rump [13] adopt parts of the BSD kernel and re-engineer it to work in a unikernel context; Lupine Linux [14] relies on a minimal, specialized configuration of the Linux kernel with Kernel Mode Linux patches. These approaches make application porting easy because they provide binary compatibility or POSIX compatibility, but the resulting kernel is monolithic.

Existing monolithic OSes do have APIs for each component, but most APIs are quite rich as they have evolved organically, and component separation is often blurred to achieve performance (e.g., sendfile short circuits the networking and storage stacks).

In summary, starting from an existing project is suboptimal since none of the options were designed to support the key principles we have outlined. We opted for a clean-slate API design approach, though we did reuse components from existing works where it is relevant.

As achievements, Unikraft supports a number of already-ported applications (e.g., SQLite, Nginx, Redis), programming languages and runtime environments such as C/C++, Go, Python, Ruby, Web Assembly and Lua, and a number of different hypervisors (KVM, Xen, Amazon Firecracker and Solo5 as of this writing).

Our performance evaluation (e.g., requests per second a nginx web server, or insertions per second for a SQLite database) using such applications on Unikraft results in a 30%-50% performance improvement compared to Linux guests. In addition, Unikraft images for these apps are around 1MB, require less than 10MB of RAM to run, and boot in around 1ms on top of the VMM time (total boot time 2ms-70ms).

6.4 Provided Features and APIs

Unikraft can improve the performance of applications in two ways:

1. Unmodified applications, by eliminating syscall overheads, reducing image size and memory consumption, and by choosing efficient memory allocators.
2. Specialization, by adapting applications to take advantage of lower-level APIs wherever performance is critical (e.g., a database application seeking high disk I/O throughput).

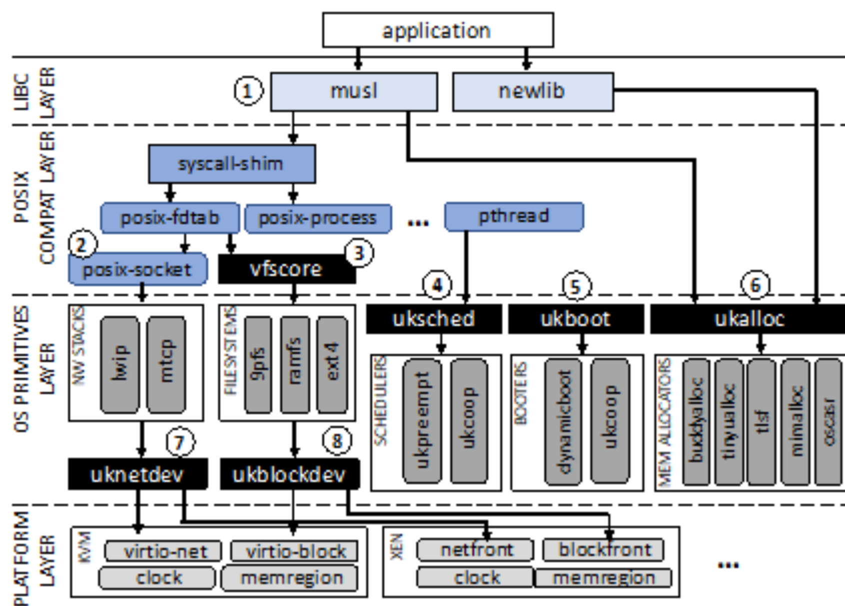

Figure 9: Unikraft architecture (APIs in black boxes) enables specialization by allowing apps to plug into APIs at different levels and to choose from multiple API implementations.

Figure 9 shows Unikraft's architecture. All components are micro-libraries that have their own Makefile and Kconfig configuration files, and so can be added to the unikernel build independently of each other. APIs are also micro-libraries that can be easily enabled or disabled via a Kconfig menu; unikernels can thus compose which APIs to choose to best cater to an application's needs (e.g., an RCP-style application might turn off the uksched API in order to implement a high performance, run-to-completion event loop). The ability to easily swap components in and out, and to plug applications in at different levels presents application developers with a wide range of optimizations possibilities. To begin with, unmodified applications can plug in to musl (① in Figure 9), transparently getting low boot times, lower memory consumption and improved throughput because of the lack of syscall overheads (note Unikraft provides a syscall shim layer to support musl, but the

syscalls result in simple function calls rather than context switches). Likewise, the application developer can easily select an appropriate memory allocator (⑥) to obtain maximum performance, or to use multiple different ones within the same unikernel (e.g., a simple, fast memory allocator for the boot code, and a standard one for the application itself).

Developers interested in fast boot times could further optimize the unikernel by providing their own boot code (⑤) to comply with the ukboot API. For network-bound applications, the developers can use standard socket interface (②) or the lower level, higher performance API (⑦) in order to significantly improve throughput. Similarly, disk-bound applications such as databases can follow a standard path through the vfscore micro-library (③), or optimize throughput by coding against the ukblock API (⑧). Schedulers are also pluggable (④), and each CPU core can run a different scheduler.

6.5 Technical Challenges and Mitigation

Perhaps the main challenges is supporting a wide range of existing applications and those needed by ACCORDION use cases. Arguably, an OS is only as good as the applications it can actually run; this has been a thorn on unikernels' side since their inception, since they often require manual porting of applications. More recent work has looked into using binary compatibility (e.g., HermiTux [15]), where unmodified binaries are taken and syscalls translated, at run-time, into a unikernel's underlying functionality. This approach has the advantage of requiring no porting work, but the translation comes with important performance penalties.

In order to avoid these penalties costs but still minimize porting effort, we take a different approach: we rely on the target application's native build system, and use the statically-compiled object files to link them into Unikraft's final linking step. For this to work, we ported the musl C standard library, since it is largely glibc-compatible but more resource efficient, and newlib, since it is commonly used to build unikernels

To quantify the challenge of supporting additional applications (especially to support ACCORDION use case) we conduct a short analysis using the Debian popularity contest data to select a set of 30 popular server applications not yet supported by Unikraft (e.g., apache, mongodb, postgres, avahi, bind9). To derive an accurate set of syscalls these applications require to actually run, and to extend the static analysis work to include dynamic analysis, we created a small framework consisting of various configurations (e.g., different port numbers for web servers, background mode, etc.) and unit tests (e.g., SQL queries for database servers, DNS queries for DNS servers, etc.). These configurations and unit tests are then given as input to the analyzer which monitors the application's behavior by relying on the strace utility. Once the dynamic analysis is done, the results are compared and added to the ones from the static analysis.

We plot the results against the syscalls currently supported by our system in the heatmap on Figure 10. Each square represents an individual syscall, numbered from 0 (read) to 313 (finit_module). Lightly colored squares are required by none of the applications (0 on the scale) or few of them (20% of them); black squares (e.g., square 1, write) are required by all. A number on a square means that syscall is supported by Unikraft, and an empty square is a syscall not supported yet.

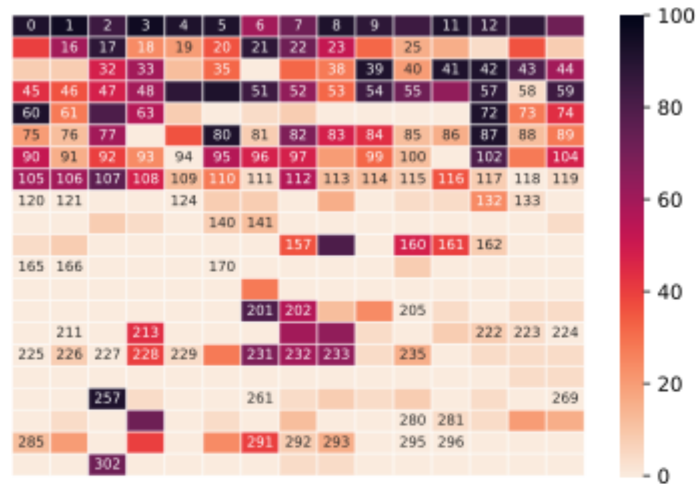

Figure 10: Syscalls required by a set of 30 popular server applications versus syscalls currently supported by Unikraft.

As can be seen from the map, more than half the syscalls are not even needed in order to support popular applications, and most of the needed syscalls we already support. Of those that are not supported (in the order of about 40):

- several can be quickly stubbed in a unikernel context (e.g., kill, since we do not have processes);
- many are relatively trivial to implement since the necessary functionality is already supported by Unikraft (e.g., semget/semopt/semctl).
- the remaining syscalls are work in progress (e.g., epoll, eventfd).

In all, we estimate that a moderate level of additional engineering work to support these missing syscalls would result in even wider support for applications. Finally, for cases where the source code is not available, Unikraft also supports binary compatibility and binary rewriting.

6.6 Future Work

As future work, we are continuing the effort to provide better syscall compatibility in order to transparently support even more mainstream applications. We also aim to leverage Unikraft's modularity for security purposes, coding micro-libraries in memory-safe or even statically-verifiable languages and using compartmentalization techniques to maintain safety properties as the image is linked together.

7 Edge minicloud VIM

7.1 Description & Objectives

An adequate management process is needed to put together and handle hardware and software resources so that they can be placed strategically near the data source or maybe near the end user, in order to lower latency and improve the user experience. The component dedicated to manage resource allocation and workload execution in a virtualized environment is known as Virtual Infrastructure Manager (VIM). Interaction with a VIM is usually done manually via Command Line Interface (CLI) or programmatically with an Application Programming Interface (API). Several different VIM implementations with different focuses are already available: there are for example OpenStack and VMware vCloud Director to manage Cloud Infrastructure, but also OpenShift (actually Kubernetes) to manage containers. There is also OpenVIM, which is a component of the OpenMANO platform for Network Functions Virtualization (NFV). Beside traditional cloud frameworks, to manage resources hosted in datacenters, there are also solutions with their own VIM developed for different environments, where resources can be scarce or not accountable like in IoT. In our research we started reviewing the state of the art, as reported in deliverable D2.2 (sect. 2.4.3.2), and we found several projects working on VIMs that could address the identified issues. Therefore, in this first year we screened in more detail some of the most promising solutions identified in D2.2, to choose the one that better matches our requisites. The evaluated solutions are Eclipse Fog05, LF Eve, Rancher K3S and MicroK8S. Eclipse Fog05 [16] is a project sponsored by the Eclipse Foundation, with a very simple architecture and based on a communication bus (Zenoh) based on pub/sub, focused on keeping a low-overhead profile even using unreliable networks. LF Eve [17] is hosted by the Linux Foundation and focuses on managing IoT resources, developing a secure-by-design system to run containers that can be managed remotely. MicroK8S[20] is a Kubernetes distribution for IoT hosted and developed by Canonical. Rancher K3S [21] is a Kubernetes distribution for IoT. Our objective is to choose one of the solutions listed above as a baseline for our VIM component. This VIM should be easy to be extended, able to work well both on cheap hardware such as Raspberry PI and on datacenter servers, ensure enough security, and be able to support different kind of workloads, i.e. both containers and Virtual Machines.

7.2 Requirements

The baseline VIM for ACCORDION must satisfy a set of requirements in different aspects, to have a good open-source basis to enable further development. It should be easy to extend with provided interfaces and should have an active community working on its evolution. It should be able to manage the kind of limited

[20] https://microk8s.io/

[21] https://k3s.io/

hardware usually available on the edge as well as standard data centre resources. As a way to guide our comparison we identified the following list of detailed requirements:

- VIM should be able to manage non-homogeneous resources, either large or scarce in terms of CPU and RAM.
- It must be able to run on small devices such as Raspberry PI, so it should support both ARM as well as x86 CPU-architecture.
- VIM should use the least possible number of resources for cluster maintenance, keeping more available for workloads.
- A security layer should be available to enforce confidentiality in components' communication, and to support authentication and access control.
- It must support, either natively or through extensions, the execution of three different kinds of workload: it should be able to run Docker containers, which are the mainstream technology for containerization, and also Unikernels for high start-up performance and maximum efficiency in resources' utilization. It should also be able to run virtual machines, giving developers the possibility to have an environment that manages both containers and virtual machines can simplify the migration from legacy to cloud-native applications.
- To simplify adoption and diffusion of ACCORDION, it's important that our VIM is accessible using standard or industry recognized interfaces.

The use of open-source components is also important, as it enables developers to modify and extend functionality independently, without the need to request permission from software owners or pay license fees.

7.3 Research Challenges & Advancements Achieved

Due to the number of candidates derived from the first screening, to tackle the selection we adopted an ad-hoc methodology. The first step was to define a set of criteria, listed below, to be evaluated for each candidate. Then the analysis of each solution followed a sequence of three phases, each phase incrementally increasing our understanding. First, each project's website was visited to collect all information available, then we reviewed project and software documentation looking at the characteristics of the solution, seeking information on the current activity of the project, its developers' community as well as its diffusion to have an idea of the long-term viability and support available for the solution. After the documentation analysis, where possible, we experimented by installing the software in our lab, to better understand capabilities, compatibility, strengths and weaknesses of each candidate solution.

In each project's webpages, we looked for a summary to understand if the available features were matching our requirements. Then we extracted the knowledge of each solution by studying its documentation, even if sometimes it wasn't enough and a direct dialogue with the related developer community was needed. Some projects are still work in progress and it was necessary to understand their real state of work by filtering all the incomplete or misleading information found in websites, source code and presentations. In most cases, it wasn't easy neither to understand if some important feature was present or missing, nor to obtain a roadmap of upcoming features. In some cases, we were able to install the solution locally and verify its

functionality by directly testing its features. In all these analysis phases, we collected information following a set of criteria based on the requirements defined in the previous section. The criteria were:

- **Documentation**: document availability and completeness, from both a developer and administrator stand point, is important to simplify the solution adoption in ACCORDION and let all partners get acquainted with it.
- **Installability**: the complexity of installation on supported systems is a hint about the product's production readiness; moreover, a simple installation is a plus in itself for its smaller operations costs.
- **Licensing**: type of open-source license, used to understand if is appropriate and matches the requisite of free reuse and modification of source code, like Apache 2.0
- **Community**: understanding if there is an active community working on the product and supporting it, gives assurances about the software's future evolution and its present support level.
- **Portability**: is VIM usable on different platforms? In ACCORDION, it's important that the selected VIM supports both ARM as well as AMD/x86 architectures.
- **Supportability**: linked with Community, indicates if support for the software is promptly available.
- **Architecture**: the simpler the solution architecture, the simpler it will be to maintain it. We looked In particular at the deployment architecture.
- **Security Features**: this criterion is linked to the need of securing the ACCORDION solution from any unauthorized and malicious access.
- **Virtualization Supported**: what kind of hypervisors are supported.
- **Compatibility**: criterion that evaluates how easy it is to integrate this solution with mainstream technology.
- **Maintenance**: complexity of the day by day operation.
- **Extensibility**: if and how the solution supports extensions.
- **Hardware Requirements**: how many resources should be dedicated to run VIM processes and are thus stolen from users' workloads
- **Project Maturity**: criterion indicating the current state of the project's development work, answer questions such as: has it reached its first stable version? Is it production ready?

The criteria listed above guided any phase of our analysis, and any phase gave different contributions for each of the solutions under analysis. During the process also the relevance of the criteria become clearer: as for community criteria, especially during the laboratory tests, the presence of an active community to support when tackling issues, was crucial. Security features must be a native part in the VIM solution, at least for low level operations, like node management and configuration management. The laboratory test was crucial to better understand potential problems in a candidate or even to better understand features poorly described in the documentation. For some projects, it has been difficult to find all the information needed and even to distinguish some promised features from actually implemented ones.

As a result of this selection work there is a much clearer picture of the current solutions for virtualization and clusterization in environments with few resources. What we learnt drove us to select K3S as VIM baseline. K3S shows good evaluations in all criteria, especially for lightweight resource consumption and for documentation completeness for both developers and administrators. From Kubernetes, K3S inherits also its REST API: an industry recognized interface, a de facto standard. This facilitates the interaction for any third

party application and especially enables ACCORDION to leverage on the whole Kubernetes ecosystems: a rich and continuously growing set of extensions and integrations for the most disparate tasks. Moreover, it is interesting the K3s project governance evolution: recently it joined CNCF (Cloud Native Computing Foundation) as a Sandbox project, which increases its chances to be kept available as an open source community version. Another interesting feature is its Software conformance[22] obtained by Rancher, meaning that K3S will keep API compatibility and can be a Kubernetes drop-in. Finally, also other ACCORDION tasks are using K3S for their labs, so this choice simplifies their work and the whole ACCORDION integration. Table 2 summarizes the comparison analysis: each candidate has a column and is evaluated on all criteria represented as rows.

Table 2: Comparison of VIM baseline candidates.

Criteria	Criteria Explanation	Fog05	EVE LF-EDGE	K3S	MicroK8s
Documentation	Comprehensive, appropriate, well-structured user documentation?	Minimum instruction install and general architecture	Wiki and documentation available is high-level. Not available interfaces and implementations specs.	Available for both users and developers. Interface is inherited from K8S	For user, not for developer to contribute. Interface in inherited from K8S
Installability	Straightforward to install on a supported system?	Pkg distribution only for Ubuntu. Compilation needed for windows	Only on bare metal with package made via make. It uses type1 hypervisor Xen or KVM, open to others type	Straightforward, with an installation script a basic cluster can be setup	Straightforward, snap script.
Licensing	Adoption of appropriate license?	Apache License, Version 2.0 Eclipse Public License 2.0	Apache License 2.0	Apache License 2.0	Apache License 2.0
Community	Evidence of current/future community?	Sponsored by Eclipse Foundation. Not big community of developer at the moment (not documented but seem less than 4 persons)	Part of LF Edge program, open to external to project contributors. Approx. less than 20 contributors in the last year	Joined CNCF (Cloud native computing foundation), SUSE announced that it is acquiring Rancher Labs	Canonical (Ubuntu)
Portability	Usable on multiple platforms?	Ubuntu Linux and Windows	tested on some IOT devices https://wiki.lfedge.org/display/EVE/EVE+in+the+Market	Linux. X86 64bit, ARM64 and ARMv7 support	Linux, Windows, macOS. x86, Raspberry/ARM
Supportability	Evidence of current/future developer support?	Is available a Slack channel open to anybody to communicate bugs or receive help. A roadmap is not	Mailing list and chat available. A 2021 roadmap is under development	2020 Roadmap available, missing 2021. Project present on GitHub for eventual defect.	Roadmap not available. Project Active with community forum

[22] https://www.cncf.io/certification/software-conformance/

		available at the moment			
Architecture	How complex is it?	Really simple, made of agents and distributed infrastructure manager.	Simple: controller-agent	Simplified wrt k8s, one or more single process masters nodes and single process workers nodes	Simplified respect to K8S
Security Feature	Access Control, TLS.	non present	secure by design: access control and authentication available	Available from k8s	Available from k8s
Project Maturity	Is the project consolidated? How many project use such vim? Is product completed or under development?	Under development, few project using it in non-production env.	no evidence of use of community version in production, is also a company product no public data about deployments	Already used in production by a Kubernetes cloud provider	Stable version 1.19, available
Virtualization Supported	Like Docker container, Lxd...	Containerd Lxd ROS2 Kvm Native application	Virtual Machines, Unikernels or Docker/OCI	As K8S distro: Docker, Containerd, CRI-O, and any implementation of the Kubernetes CRI (Container Runtime Interface)	As K8S distro: Docker, Containerd, CRI-O, and any implementation of the Kubernetes CRI (Container Runtime Interface)
Compatibility	with OpenStack and OCCI	Compatible with ETSI OSM Orchestrator, but only for few features	No compatibility with main stream technology	As K8S distro there are several de facto standard REST API interfaces	As K8S distro there are several de facto standard REST API interfaces
Maintenance	how is complex day by day maintenance	TBD	Enable remote maintenance of bare metal OS, with rollback feature	available a system-upgrade-controller that automate system update	Automatic security update. single command to Kubernetes update
Extensibility	Is possible to add features? Is it complex?	High: software architecture plugin based, for OS, FDU (fog deployment unit, kind of workload), network	Community support encouraged, but no evidence of standard interface or external contribution like for K8S CSI	Extension can be done on K3S. Also K8S enable extension with different interfaces widely known like: CSI, CRD, and CRI…	Extension done on MicroK8s, to include new feature but also on K8S that enable extension with different interfaces widely known like: CSI, CRD, CRI…
Hardware Req.		Seams light: 57mb, no details for CPU and memory, no benchmark available	CPU: Arm or Intel/AMD support for hypervisor. 1G RAM or better 1G storage 1 network interface 1 serial port for development	OS: Linux RAM: 512MB Minimum (recommend at least 1GB) CPU: 1 Minimum Storage: 100Mb	OS:Lin,Win,mac Ram:4GB Storage: 20GB
Tested in our lab		Tested looking for: - high availability of message bus zenoh - try running simple Container	It doesn't work with nested virtualization, found problem executing, nobody ever tried on top of VMware or VirtualBox	Test demonstrates simple installation and good integration with VirtualBox for VM execution	N.A.

K3S natively can run only Containers, not virtual machines or Unikernels, so it was necessary to find an extension to support such other types of workloads. A further selection was needed to find the right plugin to extend K3s for supporting at least Virtual Machines. We found several candidates: Kata Containers[23], RancherVM[24], KubeVirt[25], KubeEdge[26] and Virtlet[27]. This time the selection was simple, as most candidates have some critical issue: Kata Containers implements VMs as a way to insulate containers, so the main workload in this framework is still a Container and not a Virtual Machine. RancherVM is a project from the same company providing K3S but it is probably in its early stages or abandoned as the documentation is completely missing and there are no signs of a community working on it. Virtlet is quite complete from the point of view of functionality and documentation, but it seems to be an abandoned project. KubeEdge architecture is not edge-only but needs a cloud component to run. KubeVirt instead matches all the requisites of virtual machine management, has a big community around it, it is also hosted by CFCN and is being used as the base for the Red Hat OpenShift Virtualization product, therefore KubeVirt is our selection as the K3S extension to support Virtual Machines. Table 3 contains candidates' evaluation for each criterion:

Table 3: Comparison analysis for candidate baseline extensions to support Virtual Machines.

Criteria	Kata Containers	Rancher VM	KubeVirt	KubeEdge
Documentation	Available for users and developers	Really Poor / Not present	For User and developer	For User and developer
Installability	VM with nested virtualization: Kata is packaged for major Linux Distro. Bare metal	N.A.	Few cmd line instruction to install	Simple installation of agent and controller
Licensing	Apache License 2.0	N.A.	Apache License 2.0	Apache License 2.0
Community	Supported by the Open Infrastructure Foundation	Rancher	CNCF compliant and GitHub community with more than 30 significant contributor in the last year	CNCF compliant and GitHub community with more than 30 significant contributor in the last year
Portability	Linux or system with: Intel VT-x technology. ARM Hyp mode (virtualization extension). IBM Power Systems. IBM Z mainframes	N.A.	Linux	Linux

[23] https://katacontainers.io/

[24] https://github.com/rancher/vm

[25] https://kubevirt.io/

[26] https://kubeedge.io/

[27] https://github.com/Mirantis/virtlet

Supportability	Project is active in development and roadmap is under development.	N.A.	Roadmap under development.	High level roadmap available
Architecture	Duplicate some k8s components	N.A.	Duplicate some k8s components	Quite complex, as it needs k8s cluster, cloud component controller and agent, so 2 separate system to maintain
Security Feature	Focused on issue of isolation of processes. Using Hw Virtualization	N.A.	Like k8s.	Like k8s
Project Maturity	Stable version has been released	N.A.	Near to 1.0, most feature are available, guarantee to keep same API. Already used as base for commercial product	
Virtualization Supported	Docker/OCI inside VM	Support for: qcow2, raw, and ISO images	KVM via libvirt	-Docker -Containerd -Cri-o -Virtlet: dead project
Compatibility	OpenStack Zun compatibility	N.A.	K8S and derivate. Tested and packaged for Katacoda MiniKube Kind Cloud k8s provider	N.A.
Maintenance	N.A.	N.A.	Zero downtime rolling updates available	automatic update
Note	Not to run generic VM			

7.4 Provided Features and APIs

As stated in the previous section, K3S is a lightweight distribution of Kubernetes (K8S). Its main value is its small resource footprint: this flavour of K8S reduced memory usage by consolidating all K8S services in two processes: one running on the master node and one running on agent nodes, both packaged as a single binary of less than 50MB. The K3S implementation includes support for SQLite3 as default storage mechanism and has cut out all unnecessary K8S API code, either alpha or deprecated. Another important feature is its optimization for ARMv7 and ARM64 processors that enable working on several small devices like Raspberry Pi, but binaries are available for AMD64 processor architecture too.

Being a K8S distribution with software conformance certified by CFCN, K3S is fully compliant with K8S inheriting most of its features. Features such as the extensibility with Custom Resource Definition (CRD), which enable users to define their own custom resources and manage them with a custom API. The CRD mechanism has been used by KubeVirt to add two new K8S resource types: Virtual Machine (VM) and Virtual Machine Instance (VMI). There are other ways to extend Kubernetes, and thus K3S, like the Container

Runtime Interface (CRI). CRI enables us to plugin different kinds of container runtimes, without the need to recompile. The current container runtime implementation for K3S is Containerd[28], which enables more efficient execution of Docker containers. For user interaction, K3S exposes two kind of interfaces:

- HTTP REST API
- Command Line Interface (CLI)

With these interfaces, users are able to configure any aspect of a K3S cluster.

Documentation for the K8S API and CLI is available online at https://kubernetes.io/docs/reference/

The latest up-to-date documentation about K3S implementation and customization is available online at https://rancher.com/docs/k3s/latest/en/

KubeVirt is a Kubernetes add-on: its main features are Virtual Machine management enablement. In doing so it has several features like: framework update silently with no downtime, migration of VMIs to a different node, and role based access control (RBAC) for authorization. All these features are available to the users via a CLI named virtctl, or via a dedicated HTTP REST API. More details about virtctl are available online at https://kubevirt.io/user-guide/#/installation/virtctl. Details on KubeVirt API are also available online, at https://kubevirt.io/api-reference/.

7.5 Future Work

In the second year we will integrate other ACCORDION edge components: edge storage, resource indexing, and resources monitoring, with the aim of obtaining a unique package for the Minicloud. We will start integration testing using a sample application, but then the plan is to run experiments using the ACCORDION WP6 use cases, to verify if their requirements are satisfied.

[28] https://containerd.io/

8 Conclusions

In conclusion, we can see that the Minicloud platform has achieved most of its goals and objectives for the first year of the project. The basic objectives of the Minicloud were to provide a platform which is able to improve the QoE for users of ACCORDION supported applications near the edge of the network. This is accomplished by having the first version of the Minicloud VIM that is highly scalable, distributed and lightweight. A prototype platform has been created, supporting basic functionality while most of the research goals have been clearly defined and they are ready to be explored and overcome. In the near future, the Minicloud VIM component will integrate its subcomponents and will be able to deploy a sample application on edge devices.

In detail, a monitoring component has been created, tackling the challenge of gathering all the real time monitoring metrics and the static characteristics of the virtual and physical machines that are part of the ACCORDION framework or even lend resources to the framework. In addition, this monitoring component is creating a connection between the physical and virtual layers, unifying them under a common context. The RID component is providing the framework with an efficient, reliable and scalable means of discovering and identifying resources dynamically as they change, move, enter or exit the resource pool. The Edge Storage component is providing the capability of utilizing the edge resources in order to store, retrieve and migrate data in a fast, secure and durable way, ensuring the QoS and QoE requirements of the applications using the ACCORDION framework. The lightweight virtualization and Unikernels component allow us to deploy images on edge nodes using minimal resources, allowing us to use mini-clouds of cheaper devices while keeping the performance at the necessary level to preserve the QoE requirements. All these components are tied under the VIM component, which integrates them together in a mini-cloud architecture.

Finally, the targets for the second year of the project have been clearly defined, both on component level and on platform level, based on the challenges identified during the first year. This includes additional functionality for the Minicloud VIM as well as optimizations to already provided functionality by expanding the current status of the art in the domain targeted by each of the WP3 task outcomes.

References

[1] C. Cramer, K. Kutzner, and T. Fuhrmann, “Bootstrapping locality-aware P2P networks,” in *Proceedings. 2004 12th IEEE International Conference on Networks (ICON 2004) (IEEE Cat. No.04EX955)*, Nov. 2004, vol. 1, pp. 357–361 vol.1, doi: 10.1109/ICON.2004.1409169.

[2] MinIO Inc, “MinIO | Enterprise Grade, High Performance Object Storage,” *MinIO*, 2020. https://min.io (accessed Apr. 26, 2020).

[3] OpenStack, “Build the future of Open Infrastructure.,” *OpenStack*. https://www.openstack.org/ (accessed Apr. 26, 2020).

[4] S. Santhanam *et al.*, “Towards Highly Specialized, POSIX -compliant Software Stacks with Unikraft: Work-in-Progress,” in *2020 International Conference on Embedded Software (EMSOFT)*, Shanghai, China, Sep. 2020, pp. 31–33, doi: 10.1109/EMSOFT51651.2020.9244044.

[5] I. Foster, Y. Zhao, I. Raicu, and S. Lu, “Cloud Computing and Grid Computing 360-Degree Compared,” in *2008 Grid Computing Environments Workshop*, Nov. 2008, pp. 1–10, doi: 10.1109/GCE.2008.4738445.

[6] G. Mateescu, W. Gentzsch, and C. J. Ribbens, “Hybrid Computing—Where HPC meets grid and Cloud Computing,” *Future Gener. Comput. Syst.*, vol. 27, no. 5, pp. 440–453, May 2011, doi: 10.1016/j.future.2010.11.003.

[7] J. Zarrin, R. L. Aguiar, and J. P. Barraca, “Resource discovery for distributed computing systems: A comprehensive survey,” *J. Parallel Distrib. Comput.*, vol. 113, pp. 127–166, Mar. 2018, doi: 10.1016/j.jpdc.2017.11.010.

[8] K. Lee, T. Choi, P. O. Boykin, and R. J. Figueiredo, “MatchTree: Flexible, scalable, and fault-tolerant wide-area resource discovery with distributed matchmaking and aggregation,” *Future Gener. Comput. Syst.*, vol. 29, no. 6, pp. 1596–1610, Aug. 2013, doi: 10.1016/j.future.2012.08.009.

[9] J. Li, “Grid resource discovery based on semantically linked virtual organizations,” *Future Gener. Comput. Syst.*, vol. 26, no. 3, pp. 361–373, Mar. 2010, doi: 10.1016/j.future.2009.07.011.

[10] J. Albrecht, D. Oppenheimer, A. Vahdat, and D. A. Patterson, “Design and Implementation Trade-Offs for Wide-Area Resource Discovery,” *ACM Trans Internet Technol*, vol. 8, no. 4, Oct. 2008, doi: 10.1145/1391949.1391952.

[11] Sujoy Basu, Sujata Banerjee, Puneet Sharma, and Sung-Ju Lee, “NodeWiz: peer-to-peer resource discovery for grids,” in *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.*, May 2005, vol. 1, pp. 213-220 Vol. 1, doi: 10.1109/CCGRID.2005.1558557.

[12] A. Kivity *et al.*, “OSv: Optimizing the Operating System for Virtual Machines,” in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USA, 2014, pp. 61–72.

[13] A. Kantee and others, “Flexible operating system internals: the design and implementation of the anykernel and rump kernels,” 2012.

[14] H.-C. Kuo, D. Williams, R. Koller, and S. Mohan, “A linux in unikernel clothing,” in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–15.

[15] P. Olivier, D. Chiba, S. Lankes, C. Min, and B. Ravindran, “A binary-compatible unikernel,” in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2019, pp. 59–73.

[16] A. Corsaro, “Eclipse fog05,” *projects.eclipse.org*, Apr. 08, 2018. https://projects.eclipse.org/proposals/eclipse-fog05 (accessed Jan. 14, 2021).

[17] “EVE - LF Edge.” https://www.lfedge.org/projects/eve/ (accessed Jan. 14, 2021).