Future Service Oriented Networks

www.fusion-project.eu

Deliverable D3.2

Updated Node Design, Algorithms and Protocols for Service-Oriented Network Management

Public report, Version 1.0, 20 December 2014

Authors

UCL David Griffin, Miguel Rio, Khoa Phan

ALUB Frederik Vandeputte, Luc Vermoesen

TPSA Dariusz Bursztynowski

SPINOR Michael Franke, Folker Schamel

IMINDS Pieter Simoens

Reviewers Frederik Vandeputte, Luc Vermoesen, Dariusz Bursztynowski

Abstract In this deliverable, we provide an update on the specification, design and implementation of service orchestration and management in FUSION. We first describe various patterns for distributed service orchestration as well as the impact of heterogeneous cloud environments, followed by an update on composite services, introducing the concept of multi-configuration service instances. We revisit and detail on several key FUSION functions, concepts and their corresponding algorithms, including TOSCA-based service manifests for service registration, lifecycle management, service placement and session-slot based scaling. Next, we discuss the design and current implementation status of the key FUSION layers, and finish with a summary of various experiments we performed this year on heterogeneous environments and placement algorithms.

Keywords FUSION, service management, orchestration, design, interfaces, algorithms, heterogeneous execution environments, light-weight virtualization

Revision history

Project funded by the European Union under the
Information and Communication Technologies FP7 Cooperation Programme
Grant Agreement number 318205

Date	Editor	Status	Version	Changes
18/08/2014	Frederik Vandeputte	Initial Version	0.1	Initial ToC
11/09/2014	Frederik Vandeputte	Initial Version	0.2	Revised ToC + partner alloc.
07/10/2014	Frederik Vandeputte	Initial Version	0.3	Revised ToC + text outline
07/11/2014	Frederik Vandeputte	Draft	0.4	Initial full-text draft
07/12/2014	Frederik Vandeputte	Stable Version	0.5	Updated + initial reviews
18/12/2014	Frederik Vandeputte	Stable Version	0.6	Second stable version
20/12/2014	Frederik Vandeputte	Final Version	1.0	Minor edits

GLOSSARY OF ACRONYMS

Acronym	Definition
ALU	Alcatel-Lucent
AMD	Advanced Micro Devices, Inc.
API	Application Programming Interface
ASS	Atomic Service Scenarios
AVX	Advanced Vector Extensions
BDP	Bandwidth Delay Product
CAPEX	Capital Expenditure
CCDF	Complementary Cumulative Distribution Function
CDF	Cumulative Distribution Function
CDN	Content Distribution Network
CEO	Chief Executive Officer
CFQ	Completely Fair Queuing
CFS	Completely Fair Scheduler
CMOS	Complementary Metal-Oxide Semiconductor
CORBA	Common Object Request Broker Architecture
COTS	Common Of The Shelf
CPU	Central Processing Unit
CSAR	Cloud Service Archive
DC	Data Centre
DCA/DCAL	FUSION Data Centre Adaptor (Layer)
DCOM	Distributed Component Object Model
DCTCP	Data Centre TCP
DPDK	Data Plane Development Kit
EC2	Amazon Elastic Compute Cloud
ECN	Explicit Congestion Notification
EPG	Electronic Programming Guide
ES	FUSION Evaluator Service
ETCD	A highly available distributed key value store for shared configuration
ETSI	European Telecommunications Standards Institute
FPGA	Field-Programmable Gate Array
FPS	Frames Per Second
FUSION	Future Service Oriented Networks
GB	Gigabyte
GPU	Graphical Processing Unit
GUI	Graphical User Interface

HD	High Definition
HDD	Hard Disk Drive
HEAT	OpenStack Orchestration Module
HOT	Heat Orchestration Template
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	HTTP Secure
HW	Hardware
IaaS	Infrastructure as a Service
ID	Identifier
IMS	IP Multimedia System
IO	Input/Output
IOPS	I/O Operations Per Second
IP	Internet Protocol
IRQ	Interrupt Request
IT	Information Technology
IVSHMEM	Inter-VM Shared Memory
JSON	JavaScript Object Notation
KQI	Key Quality Indicator
KVM	Kernel-based Virtual Machine
LB	Load Balancer
LP	Linear Programming
MaaS	Machine as a Service
MANO	Management and Orchestration
MB	Megabyte
MMC	Multi-server queuing model (Erlang-C model)
MPI	Message Passing Interface
MTU	Maximum Transmission Unit
NAT	Network Address Translation
NFV	Network Function Virtualization
NIC	Network Interface Controller/Card
NUMA	Non-Uniform Memory Architecture
OASIS	Advanced Open Standards for the Information Society
OFED	OpenFabrics Alliance
OPEX	Operational Expenditure
OS	Operating System
OTT	Over The Top
OVS	OpenvSwitch

PaaS	Platform as a Service
PCI/PCIe	Peripheral Component Interconnect (Express)
PHP	PHP Hypertext Preprocessor
PoC	Proof of Concept
QoE	Quality of Experience
QoS	Quality of Service
RAM	Random-Access Memory
RDMA	Remote Direct Memory Access
RDS	Reliable Datagram Sockets
RENO	TCP Reno congestion-control strategy
REST	Representational State Transfer
ROCE	RDMA Over Converged Ethernet
RT	Real-Time
RTT	RoundTrip Time
SDP	Session Description Protocol
SIMD	Single Instruction Multiple Data (vector instructions)
SLA	Service Level Agreement
SLT	Senior Leadership Team
SMP	Symmetric MultiProcessing
SR	FUSION Service Router
SR-IOV	Single-Root IO Virtualization
SSD	Solid State Disk
SW	Software
TCO	Total Cost of Ownership
TCP	Transmission Control Protocol
TLB	Translation Lookaside Buffer
TLS	Transport Layer Security
TOSCA	Topology and Orchestration Specification for Cloud Applications
UHD	Ultra-High Definition
UML	Unified Modeling Language
VDesk	Virtual Desktop
VM	Virtual Machine
VNF	Virtualized Network Function
WP	Work Package
XML	Extensible Markup Language
YAML	Yaml Ain’t Markup Language
ZM	FUSION Zone Manager

EXECUTIVE SUMMARY

This document is a public deliverable of the “Future Service-Oriented Networks” (FUSION) FP7 project. It focuses on the algorithms, protocols and functionality of the service management and orchestration layer in FUSION, including the design and implementation of all the FUSION components implementing this service layer.

The key challenges for this work package are twofold. A first challenge is how to build a composable and scalable service layer for efficiently managing large-scale personalized services across distributed heterogeneous execution nodes. A second challenge is developing efficient and scalable service placement and scaling algorithms for optimally deciding on when and where to deploy particular service components, based on service demand, service requirements and platform capabilities.

The service management layer is conceived as a set of orchestration domains. In each orchestration domain, there is a logically centralized domain orchestrator where service providers can register and manage their services, and where services are globally managed within a domain. This includes the high-level placement and scaling of the services in the various execution zones that are managed by the domain. Each execution zone is managed by a FUSION Zone Manager, which handles all local deployment, monitoring, load balancing and scaling management. This decoupling ensures a scalable service layer in which all local management details are handled locally and all global decisions are made at the domain level.

Within an execution zone, we also distinguish between the higher-level FUSION management and orchestration and the lower-level deployment and management of service instances on either physical or virtual resources. In FUSION, we envision that FUSION Zone Managers could be deployed on either bare physical environments or on cloud infrastructures, which may be managed by a third party. The former type allows for a much higher and finer degree of control of the underlying resources but requires direct access and low-level management of FUSION. The latter type allows FUSION to delegate most of the lower-level deployment details to the underlying cloud platform, but obviously allows for less fine-grained control. To be able to abstract this varying degree of control and low-level management, we designed a Data Centre Adaptor layer, to clearly distinguish the higher-level FUSION zone management functions from the lower-level deployment functions.

This deliverable is the second public Deliverable for Work Package 3, and extends and refines the various concepts and initial design and high-level protocols that were described in Deliverable D3.1. The deliverable contains three main parts. The first part covers the overall vision, concepts and algorithms of FUSION orchestration and management, starting with a description of higher-level patterns, challenges regarding heterogeneous environments and a discussion on composite services, presenting a new concept on multi-configuration service instances. We then provide an update and present new algorithms for the key FUSION orchestration functions, including service lifecycle management, placement and scaling.

In the second part of the deliverable, we provide a more detailed design of the three FUSION orchestration layers, and discuss in detail their current implementation status. We also discuss how a heterogeneous cloud platform could be designed onto which FUSION services and FUSION execution zones could be deployed, optimizing both QoS and overall efficiency. At the end of this part, we discuss the FUSION orchestration protocols (which are described in more detail in the Appendix), as well as discuss different inter-service communication protocols to enable efficient inter-service communication within an Execution Zone.

In the last part of this deliverable, we then present specific evaluation results of particular FUSION concepts (such as multi-configuration instances), placement algorithms and enabling technologies (such as lightweight virtualization, NUMA-pinning and real-time guarantees).

In the final year of the project, we will extend the algorithms, functionality and implementation of the FUSION service management layers for dealing with more complex service patterns, specifically with respect to composite services. Secondly, we will continue integrating and extending our work on heterogeneous cloud environments for efficiently managing both services and resources in a distributed heterogeneous environments. This will result in an integrated prototype that will be evaluated as part of WP5.

TABLE OF CONTENTS

1. SCOPE OF THIS DELIVERABLE

This deliverable focuses on the service management layers of the FUSION architecture: the orchestration layer, the execution layer and the service layer. This deliverable extends and refines the various concepts, designs and APIs as described in the previous Deliverable D3.1.

An overview of the key contributions w.r.t. D3.1:

- We describe a number of fundamental patterns for distributed service orchestration and discuss their role with respect to composite services;
- We describe key challenges and opportunities with respect to heterogeneous execution environments for efficiently deploying demanding or time-sensitive applications;
- We elaborate on the two main types of composite services we focused our research on during the second year of this project, and we introduce a new concept in FUSION for efficiently overlaying multiple services on top of particular service component instances;
- We discuss how FUSION can leverage and extend TOSCA for automatically deploying and managing demanding services in a distributed heterogeneous environment;
- We provide an update on the key FUSION orchestration functions and their corresponding algorithms, including monitoring, lifecycle management, placement and scaling;
- We provide a second iteration of the design of all FUSION orchestration layers and discuss the current status of their implementation: a FUSION domain orchestrator, zone manager, DC adaptor and underlying heterogeneous cloud platform.
- We provide details on the FUSION orchestration protocol specifications design and implementation, as well on various inter-service communication protocols for efficiently communicating across instance components, describing and comparing various components;
- We provide evaluation results regarding various enabling technologies and algorithms, including experimental results on the potential of a heterogeneous cloud environment and the impact of service placement algorithms.

In the last year of the project, we will focus on the following aspects:

- Work out and extend the key FUSION MANO functional blocks and algorithms for coping with more complex patterns regarding composite services, scaling and placement;
- Extend the prototype implementations of each FUSION layer, breaking them up in the key functional blocks and finalizing their internal protocol specifications;
- Continue work and integration of heterogeneous cloud environments for efficiently deploying demanding services on various hardware infrastructures and platforms.

2. FUSION ORCHESTRATION AND MANAGEMENT

This section provides a detailed update of the various key enabling FUSION concepts, functionalities and algorithms for efficiently orchestrating and managing demanding real-time personalized media services in a distributed heterogeneous cloud environment.

2.1 Patterns for distributed service orchestration

From the FUSION perspective, orchestration of distributed services has several dimensions or aspects, and each dimension introduces a different class of problems to solve. It is assumed that for each such dimension one can provide a set of specific solutions that tend to be repeatedly used in practical situations. We refer to them as patterns.

The main reason to define these dimensions and identify underlying patterns is to expose inherent complexity of service orchestration and enable a pragmatic approach to designing orchestration procedures. For example, we note that of great importance to FUSION is optimal orchestration of services. We believe in particular that designing specific optimisation models for individual, commonly used patterns may be a viable way to solve related problems in practice.

Presented in next subsections is a list of main dimensions and exemplary patterns of FUSION orchestration. It must be stressed that FUSION adopts an incremental approach in defining such patterns. Therefore it is not our goal to provide an exhaustive specification of patterns in this document nor even throughout the duration of the project. However, we note that selected patterns, identified based on the use cases being developed within the project, are elaborated in more detail in other sections of this document. One of the goals of the final year of the project is to identify and work out additional patterns, specifically in the context of composite services.

2.1.1 Service scaling and placement

This dimension deals with the problem of how scaling decisions related to service instances are taken and what the scope is of scaling decisions. We distinguish three different aspects of service scaling as outlined in the remainder of this section.

Service scaling triggers. This aspect relates to the reasons and goals that underlie the decisions to scale services. In this context we identify two canonical patterns as follows:

- Performance degradation due to lack of reserved resources (scaling up needed) or reserved resources available in excess (scaling down needed) **under apparently stable demand**. In this case the goal is to allocate infrastructure resources to service instances while maintaining current capacity of the service in terms of the amount of session slots the service offers to the users (either globally or in certain areas, or based on yet another criteria). Its importance for FUSION may be explained by the fact that due to the demanding nature of services and the scarcity of resources in smaller DCs close to the edge, it is advantageous to minimize the amount of reserved resources. This pattern may apply to both atomic as well as composite services.
- **Increased or decreased demand** and respective performance degradation or over-performance of the service. In this case the goal is to adjust service capacity in terms of the amount of available session slots and keep it within reasonable limits. It has to be noted that the potential importance of this pattern relates to the fact that in a distributed environment such as FUSION, it is not so easy or cost-effective to have enough instances up and running in all necessary locations (compared to central cloud). As a result, FUSION may need to quickly scale up in particular locations based on unpredicted demand. Indeed, due to partitioning of services across DCs, the statistics become harder to use for the long-tail of less-popular services. This pattern may apply to both atomic as well as composite services. We note also that the exact way to achieve the above goal will depend on the multiplexing pattern (see Section 2.3.2) that is implemented by the service.

Scope of scaling action. This aspect relates to the scope of service scaling determined by the type and number of locations across which scaling takes place. Currently we define two scaling patterns as follows:

- *Local scaling* (scaling at the execution zone level). This is a simple pattern that can easily be combined with other patters like the ones defined for reasons and objectives of scaling. In the latter case it can thus generate a compound local pattern based on local performance criteria to maintain session slots or based on demand.
- *Wide-area scaling* across multiple execution zones. It typically involves adjustment of service instances to align the amount of available session slots with demand coming from a particular region, possibly meeting given performance and/or other constraints. This specific variant can also arise when the scaling of particular component(s) of a composite service, done autonomously at the execution zone level (see above), mandates to also scale other component instances of affected composite service instance. From this latter example we observe that scaling patterns can sometimes be chained to generate sequences of related scaling actions. Such complex chains may in fact deserve specific treatment as individual (complex) patterns.

Interaction patterns. This aspect is related to different functional elements of the overall FUSION architecture that can undertake scaling actions. As scaling may require cooperation between those functions, interaction patterns between them can be defined. So far, we have identified three following interaction patterns:

- *Orchestrator-initiated scaling* - scaling decision is taken by the orchestrator based on either (objective) criteria for scaling based on native DC-level measurements or (subjective) criteria provided by the service or service components based on internal (proprietary) measurements.
- *Service-initiated scaling* – scaling decision is explicitly requested by the service based on (possibly specific) measurements done by atomic service components.
- *Agent-initiated scaling* – from the orchestrator perspective similar to service-initiated scaling but, compared to the service itself, potentially providing additional degree of flexibility in terms of sophistication of interactions with other platforms components (orchestrator, zone manager, DC manager).

In case of orchestrator-initiated scaling, the role of the orchestrator is both to decide whether the performance criteria for a scaling action are met, evaluate the desired new configuration of the service including such aspects as placement of new components, as well as authorize the scaling action to assure that it complies with policies and other functional requirements that may apply. In a service-initiated scaling pattern, the service instance triggers the FUSION platform by issuing a scaling request and provides information describing the desired new configuration of the service.

Placement. In general, placement of service instances involves the use of algorithms and from this point of view placement is a result of certain optimizations rather than of applying a repeatable rule that defines a pattern. However, we note that FUSION introduces the concept of an evaluator service as a means for the orchestrator to collect information describing the capabilities of DC environment to execute service of a given type. We believe the use of evaluator services for orchestration purposes opens many possibilities for research and may give rise to the creation of patterns that will define best orchestration practices with the use of evaluators. This topic will be studied in more detail during the next period of the project.

2.1.2 Plans (work-flows or choreographies)

This dimension is intended to capture the dynamic aspects of a composite service structure (or topology) during execution. The goal is to provide the orchestrator with knowledge about dynamic behaviour of services that can be used in taking optimal orchestration decisions related, e.g., to the placement of atomic instances of composite service components.

From a FUSION perspective, an instance of a composite service is a collection of atomic instances, which evolves in time in two dimensions: as atomic instances join and depart the composite instance, and as they set up and release communication sessions with other atomic instances during the service instance lifetime. In general, such evolution is a choreography or execution plan that can be expressed using appropriate notation to capture the desired features of service evolution.

A simplest example of plan pattern is when all atomic instances of a composite service and all required communication sessions are instantiated (terminated) when the composite service instance is created (respectively, terminated); in this case the orchestrator has full knowledge of the service to figure out its optimal instantiation even if the graph of the service is arbitrary (star, tree, mesh, etc.). Another pattern where close-to-optimal orchestration is possible is a star topology where a root atomic instance created at service instantiation time communicates with leaf instances and the leaf instances (and corresponding communication sessions) are created in predefined time instants and last for a predefined time. Note that randomized variants may be derived from the latter deterministic pattern by randomizing its selected parameters, e.g., the creation time of the leaf instances and/or the duration of corresponding communications session. In either case, the orchestrator has less information on the actual execution of the composite service instance but still can try to figure out suboptimal decisions. Note also that in a worst case it is always possible to project a given pattern onto the simplest (static) variant. This approach can be treated as default, available to all orchestrators.

It must be stressed that at current stage, plans are assumed to represent only the external view of composite services. That means in particular that plans do not describe the internal use of communication channels represented in the service graph. In other words, plans do not define different end-to-end flows that may traverse such channels and the (logical) paths such flows may take among the service components of composite service. The plans do not deal with the functionality, configuration and state of service components to map such flows onto an instantiated service graph. There is a strong similarity of plans to the concept of VNF Forwarding Graphs discussed within the framework of NFV [ETSI13].

We conclude pointing out that FUSION plans may correspond to TOSCA-defined plans. It is believed that a set of basic plans (e.g., the two mentioned above in this section) can already be supported by the many notations allowed by TOSCA. However, FUSION may introduce specific behaviours and thus require enhanced notation and respective platform mechanisms that enable the description and realization of related patterns. We leave the latter for further analysis when explicit needs arise for such extensions.

2.1.3 Service overlaying

A basic and simple scenario for the realization of a composite service is to assume that each atomic instance of such a service responsible for a subset of functionality is realized in the form of a stand-alone virtual machine. In case of such one-to-one mapping the orchestrator has a clear view of service instances both on the level of atomic instances as well as composite service instances. Despite its simplicity, this option may however lead to a poor use of infrastructure resources, especially in case the services are scattered across many small data centres.

A possible workaround to the above problem is extending the capabilities of atomic service components so that a single component instance can support multiple instances of the target service (multiple service sessions each corresponding to a distinct user). From the functional point of view this is equivalent to overlaying atomic component instances of the target service onto the instances of hosting service components. Of course it will be up to the particular implementation of the FUSION platform (orchestrator and zone manager in particular) what its actual awareness is of such overlaying capability and what the allowable patterns are for orchestrating respective composite services. Nevertheless, it becomes clear from the discussion that, from the theoretical point of view, two canonical orchestration patterns can be defined as follows:

- single-service component pattern when one atomic service component can realize only one instance of atomic service (the latter being a component of simple or composite service);
- overlaid multi-service patterns when one atomic service component can realize multiple (distinct) instances of atomic service and the orchestrator may play a double role to orchestrate both the hosting.

Concluding the above discussion on service overlaying we note that the overlaid multi-service pattern is important for services considered in FUSION such as the EPG service. As such, the different aspects of service overlaying and multiplexing will be discussed in greater detail in Section 2.3.2.

2.1.4 Service forwarding graphs

This dimension deals with the problem of configuring the atomic component instances of a running instance of a composite service so that the internal flow of information (so to say, routing "inside" a service instance) complies with predefined requirements of the supported application.

In fact, this functionality complements plans and guarantees that end-to-end (application-level) flows handled within service instance take appropriate paths (traverse appropriate atomic instances in order). This functionality thus deals with the configuration and state of service components to map such flows onto instantiated service graph. Such a service graph can be treated as a virtual network with nodes in the form of atomic instances and links in the form of communication sessions between such instances.

We note that such requirements may be very diverse, e.g., they may relate to different protocols (layers) including application protocols, and even to whole protocol stacks in a general case. Therefore it is hard to provide a generic mechanism that would cover all cases as they are inherently related to the patterns of internal communication at the application level.

2.2 Challenges and opportunities of a heterogeneous environment

This section covers the key challenges and opportunities of a heterogeneous cloud environment for managing demanding workloads, services and network functions.

2.2.1 Motivation

One of the key drivers for (public and private) cloud is the promise of providing potentially large operational gains for service and infrastructure providers, by being able to automatically deploy and manage services (using a single common orchestration platform) on COTS general purpose hardware (allowing consolidating of available hardware resources across all services). As depicted in Figure 1, this works well for standard IT services such as Web services, NoSQL data bases, etc. Apart from standard IT services, operators are currently also investigating the potential for cloud for network functions (e.g., IMS, etc.) in the NFV consortium [OPNFV], where the initial focus was on defining a common architecture for deploying (control plane) network functions.

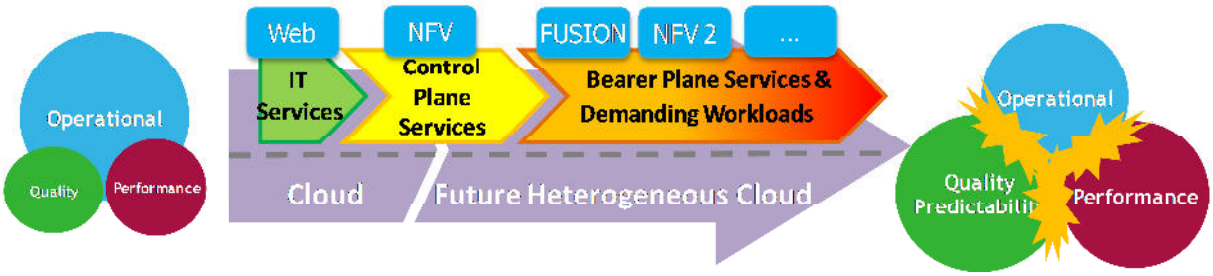


Figure 1 – In future cloud, not only operational benefits will be important

However, for more demanding workloads or bearer plane functions as FUSION is also targeting, not only the operational aspects are important, but also the overall efficiency and QoS of those applications and services. For such applications, deploying them agnostically on a general purpose cloud, not taking into account the service requirements and infrastructure capabilities and limitations, may not result in a overall TCO reduction due to reduced overall quality and performance. This is shown in Figure 2, where the impact on the operational benefits and overall efficiency is depicted of cloudifying demanding services on an *agnostic* general purpose cloud. Note that the horizontal axis in this graph could be loosely regarded as OPEX reduction, whereas the vertical axis loosely represents CAPEX reduction. Consequently, the diagonal represents TCO reduction.

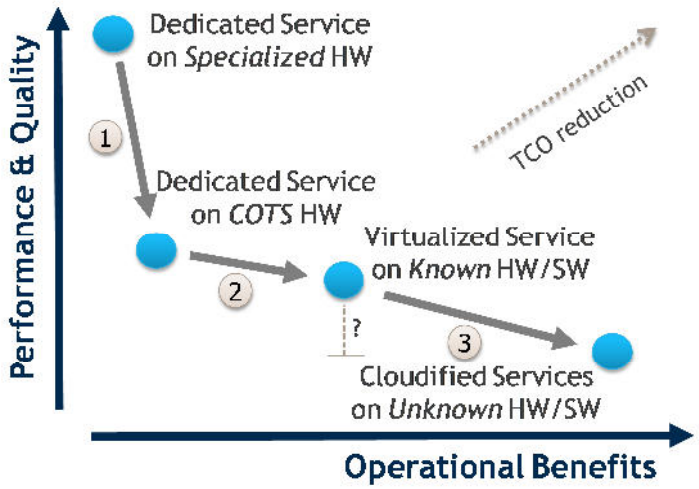


Figure 2 – Trade-off between operational benefits & efficiency when virtualizing and cloudifying demanding services on unknown COTS hardware and software infrastructures

The process of cloudifying (existing) demanding workloads typically consists of three key steps:

1) First, as demanding services are currently often deployed on special-purpose hardware, a first step is to modify these services to run onto COTS hardware. Although this may already provide some operational benefits (i.e., consolidation of less expensive hardware), a potentially large impact in relative performance and/or quality may be noticed, as the COTS hardware is by far not as efficient as the specialized hardware.

2) The second step typically involves adding a virtualization layer and deploying the applications on known COTS hardware. Knowing the environment and infrastructure allows the service provider to keep the reduction in efficiency to a minimum, while gaining additional operational benefits by allowing multiple services to be deployed on the same physical infrastructure (i.e., oversubscription of physical resources), as well as having a common packaging and deployment platform (i.e., virtualization layer).

3) The last step is to integrate the virtualized service in a fully automated cloud environment, adding proper management services, manifests etc. to enable auto-deployment, auto-scaling and auto-healing capabilities. This will again provide significant operational benefits, whereas the impact on efficiency heavily depends on the underlying cloud infrastructure and policies.

In summary, these three steps may induce significant relative operational benefits, however at the expense of potentially large reductions in relative efficiency for demanding and/or sensitive applications or network functions. In the end, this may even result in a higher TCO when moving to a cloud solution instead of a dedicated solution.

Consequently, for such applications (and network functions) to be cost-efficient in a automated cloud environment, we envision a heterogeneous cloud environment, taking into account the

application requirements and the HW/SW platform characteritis, and automatically tuning the software platform and hardware infrastructure based on these requirements and the underlying capabilities and limitations.

We also envision in FUSION that these (heterogeneous) cloud environments, apart from being distributed, in the future also will consist of novel hardware infrastructures such as micro-servers that are designed for energy efficiency, as well as hardware accelerators for regaining some of the performance efficiency losses from using general purpose hardware only. In summary, we foresee several key opportunities and challenges regarding a cost-efficient deployment of such services in a heterogeneous environment, on which we elaborate in the next section.

2.2.2 Challenges

Due to the characteristics and requirements of demanding and/or time-sensitive applications, deploying them on general purpose hardware in modern cloud environments introduces a number of nontrivial challenges and opportunities, both from a software as well as a hardware point of view.

- **Understanding the underlying hardware infrastructure**

  Both at the application layer as well as the cloud management layer, it is important to understand the properties and complexities of the underlying hardware infrastructure. This includes understanding how to cope with particular cache hierarchies and their respective sizes, how to deal with NUMA configurations and other aspects that may significantly impact efficiency and how to handle hardware acceleration from a resource planning and application perspective.

  A key challenge is how to enable this in a cloud-friendly manner. For example, manually pinning applications onto NUMA nodes may result in a fragmentation of the resources with corresponding fragmentation overheads. Also, oversubscription can negatively impact application performance, resulting in reduced QoE and reliability.

- **Understanding the software platform**

  A second major challenge is how to cope with multiple abstraction layers, like virtualization, platform APIs, etc., as well as understanding an application's behaviour and its runtime requirements. These layers abstract the hardware infrastructure and simplify deployment and management of applications and increase the resource consolidation across more applications. Unfortunately, performance and performance predictability are very hardware dependent, and for some application classes, ensuring proper operation and QoE requires guaranteeing particular (minimal) levels of performance. Consequently, understanding the impact of a particular abstraction layer on a particular hardware infrastructure is a first step for ensuring proper QoS levels.

- **Guaranteeing QoS**

  A third key challenge related to deploying a real-time application on general purpose hardware in a cloud environment is how to guarantee proper QoS levels [BONI10], or vice versa, what QoS levels an application can expect from such an environment, and what the platform needs to provide for guaranteeing these QoS levels. A concrete example is how a multi-tenant cloud environment could guarantee a particular minimum amount of memory bandwidth to a particular application. Current multi-tenant virtualization and isolation mechanisms fail to guarantee such fine-grained QoS levels. However, for particular types of applications, such fine-grained control is essential for guaranteeing proper execution and QoS.

As mentioned earlier, we envision future (distributed) data centres to be heterogeneous in nature, consisting of different physical hardware architectures, including novel DC infrastructures and models such as micro-servers, server disaggregation as well as the incorporation of hardware

accelerators such as GPUs and FPGAs. Such heterogeneity however introduces a number of additional challenges:

- **Management and configurability**

  As we will show in Section 4.1, efficient usage of hardware acceleration in a virtualized environment greatly complicates the overall configurability of a system: many specific configuration parameters need to be applied for the entire system to collaborate in an optimized manner. Since the benefits can be huge, a key challenge is how to control and reduce the management and configuration complexity of such complex systems.

- **Programmability**

  Introducing specialized hardware appears to contradict with one of the core philosophies of cloud, namely that applications should be able to run everywhere without caring about the underlying platform. *Hey, if one VM does not seem capable of handling all the load (due to additional virtualization overhead), then just simply scale out and launch another one, right?* Although this may be sufficient for particular classes of applications, we claim that for more demanding or time-sensitive applications, simply scaling out is not always an option, and hardware acceleration is inevitable for cost-efficiency or performance reasons. For several reasons, these hardware accelerators should preferably be either fixed functions that can be leveraged for a wide range of applications (e.g., video encoding function), or general purpose accelerators (e.g., GPUs, FPGAs, etc.) that should be programmable and configurable via standard APIs (cfr. OpenCL), preventing cloud environments from becoming dedicated hardware platforms. Vice versa, this promotes developing applications that can be deployed efficiently on different types of cloud infrastructures with different types of hardware acceleration (if any).

  A second challenge is developing proper programming models for efficiently programming heterogeneous devices in a cloud environment. Although standards like OpenCL exist and provide a common hardware abstraction layer and unified programming model, there are some important limitations. First, OpenCL has a limited scope by focusing on massively parallel high-performance applications. Second, OpenCL was mainly designed for HPC or grid environments rather than cloud environments, assuming a single-tenant system with full visibility and control over all available hardware accelerators. Third, OpenCL does not provide any performance abstractions. This means the application developer has to provide specific implementations for specific devices (and even device models), which can be very costly or impractical when these applications are to be deployed in a cloud environment, where the set of accelerators may not be known in advance.

- **Automation and optimization**

  One of the core aspects of cloud computing is the ability to automate the deployment and management of the hardware infrastructure as well as the applications that are deployed on that hardware. Managing accelerated applications on a heterogeneous set of hardware significantly complicates this automation process, having to take into account the different hardware ratios of the available systems (e.g., a low-power high-storage versus a high-performance low-IO system).

  On the one hand, the cloud management layer needs awareness of the heterogeneity of resources, as well as awareness of resource utilization factor, etc. of new and specialized resource types. On the other hand, it needs to optimally map the various applications on the hardware infrastructure as well as assign and configure the selected accelerators to the respective applications or their VMs. To automate this, the cloud resource manager needs to know which accelerators are required or preferred by a particular application. The application needs to provide this information statically or dynamically in a preferably standardized manner to allow interoperability between clouds. (e.g., as part of their TOSCA description or through FUSION evaluator services).

This raises the question on how applications will describe their affinity or dependency towards particular types of hardware accelerators. A strict dependency (e.g., a specific GPU model) may severely limit the deployment flexibility. In case of weak dependency, the cloud orchestration may assign a hardware accelerator to an application yielding suboptimal service quality or efficiency.
A third key challenge is how to automate the dependency formulation and how the interface towards the applications would look like. In FUSION, we propose the concept of evaluator services for solving this issue in a very flexible dynamic manner.

- **Economics of scale**

  One of the key drivers of cloud computing concerns the cost advantages that arise from deploying services in large data centres with mass-production general purpose hardware. A major potential pitfall when introducing hardware accelerators in such environments is that the performance benefits from the accelerator may be counterfeited by the cost of owning and managing that accelerator. Consequently, the economics of scale should also apply here: these accelerators should be mostly COTS hardware that can be deployed in relatively large volumes across multiple data centres and that can be programmed via specific APIs or programming models and that also can be consolidated across many application domains. Examples types of accelerators include GPUs, FPGAs, etc.

- **Migration**

  Introducing heterogeneous hardware significantly complicates and constrains VM or application migration from one hardware system to another [PITT07]. At the coarse-grain level, it is not so straight-forward to efficiently migrate a VM running on a x86 system onto an ARM system. At the fine-grain level, applications may rely on specific hardware acceleration (e.g., AVX instructions, or a specific GPU) and may either fail hard or perform very badly when being migrated to another system at any point in time. Moreover, these accelerators often contain active application state that needs to be migrated and that is not visible to the hypervisor when that device is passed through to the VM. Non-application assisted migration of applications or VMs using specialized hardware in pass-through mode thus is currently hard to accomplish. As such, migration should be handled differently in such heterogeneous environments.

2.2.3 Opportunities

In this section we list a number of key opportunities towards more cloud-friendly optimization strategies.

- **Reducing friction between applications and their environment**

  Applications often have specific requirements, not all of which are often understood or identified in advance, and the environment is typically not optimized for these requirements. Even on dedicated environments, this results in time-consuming optimization cycles to try to better match the application to its environment, or vice versa. Deploying these applications in a particular environment causes friction. In cloud environments, techniques like elastic scaling are typically used to partially circumvent the impact of this friction. In heterogeneous cloud environments, a key question is how to optimally map an application onto particular hardware. This may require proper knowledge of key application requirements, the hardware topology and capabilities as well as the interference of one application onto the other in case of multi-tenancy, all of which needs to be taken into account by an advanced placement algorithm. Alternatively, another interesting research question is how applications, hardware infrastructures or even execution models could be improved to reduce the amount of friction a priori.

For a particular infrastructure, the amount of friction can be reduced even further by carefully tuning how an application is deployed onto the available infrastructure. This may require improving the operating system, hypervisor as well as cloud management platform in several ways. In Section 4.1, we illustrate the benefits of reducing this friction manually. Obviously, in a cloud environment, this process should be fully automated, and should have self-monitoring and self-healing capabilities for timely error-recovery.

We envision a solution where a cloud management platform like OpenStack could take in the application requirements on the one hand and the infrastructure capabilities and limitations on the other hand, and performs an intelligent mapping, either based on a number of rules, policies and heuristics, or through a series of incremental feedback driven optimizations.

- **Reconciling heterogeneity with cloud**

  Introducing true heterogeneity in the cloud may significantly complicate management of applications and hardware resources. An important research question is what the practical limits in terms of heterogeneity are with the current cloud model: how much heterogeneity can be tolerated without radically changing the core principles of cloud computing. A key driver here is the consolidation of these accelerators of heterogeneous hardware across as many applications as possible. Vice versa, there may also be other cloud hardware or software models that do not suffer from these limitations and for which heterogeneity is a necessity rather than a burden.

- **Guaranteeing QoS**

  As discussed earlier, particular applications have more stringent requirements than the best-effort approach offered by modern cloud environments. Efficiently dealing with such applications requires advances at all layers of the stack, from the hardware layer all the way up to the application layer. The hardware layer could provide more fine-grained performance isolation, for example by adding hardware support, or by introducing new hardware models that enable finer partitioning of the hardware resources amongst applications, reducing the need for multi-tenancy and resource sharing. The OS and hypervisor layers could better map applications onto specific hardware infrastructures and have better support to monitor and handle performance issues. For example, when a VM exceeds its requested memory bandwidth quota, the hypervisor or OS could temporarily suspend that VM, preventing the VM from starving other memory-intensive VMs on the same system.

  These QoS requirements could either be specified in a more detailed manner by the applications, or the platform should be able to extract these lower-level requirements automatically, based on a number of higher-level application requirements specified by the application developer. Finally, the cloud management layer should take into account the QoS requirements of the applications and the specifications of the hardware to efficiently map and monitor these QoS-sensitive applications running in the cloud.

- **Relaxing the fixed hardware ratio**

  A fundamental problem with modern hardware infrastructures is that each server blade basically has a fixed hardware configuration with an associated fixed hardware ratio: a particular amount of CPU power with a particular amount of memory, networking, storage, etc., which inevitably results in a partitioned system. In a multi-tenant (heterogeneous) cloud environment, applications are bin-packed on that hardware, eventually resulting in unused and therefore wasted resources and possibly bad application QoS. Moreover, application requirements often change over time, so the optimal hardware ratio may also change over time.

  One interesting solution would be to relax this fixed ratio and allow the cloud management system to define the hardware environment for a particular application dynamically according to the application requirements. This may result in higher application efficiency, stability as well as

better consolidation of available resources, thereby also reducing the amount of friction between the applications and their environment.

- **Creating pro-active tools and quality metrics**

  More advanced tools as well as key quality indicators (KQIs) are required to assess application and system behaviour dynamically. This may involve automatically identifying the necessary performance metrics that are relevant for a particular workload set, correlate all metrics as well as predict future behaviour (e.g., based on past application behaviour [SHER02]), identify the culprit (e.g., memory interference between applications) and feed this information to the cloud management system.

- **Embracing emerging hardware technologies**

  New emerging hardware technologies like graphene transistors [DURA13, LIN08, MALK12], 3D-stacking and memristors may drastically change the landscape of how compute nodes should be designed and applications should be developed, which in its turn may also radically change the current cloud models as we know them today.

2.2.4 FUSION perspective

In FUSION, we address some of these issues in a number of ways. First, we introduced already the concept of evaluator services in combination with matching placement algorithms for efficiently and flexibly taking into account service-specific and hardware-specific requirements, by allowing services to make a thorough fine-grained assessment of the capabilities of all environments within and across execution zones.

However, these evaluator services can only detect and query what capabilities a particular environment has to offer. Additionally we also envision a heterogeneous cloud platform (underneath a zone manager) that can provide these more optimal environments for particular applications so that they can make effective use of these low-level optimizations. These tunings may include better resource isolation, pinning, providing particular accelerated functions, etc. In other words, both the service layer as well as the infrastructure/platform layer can both optimize according to their respective criteria.

Service quality in FUSION is handled through a combination of various mechanisms. From a service layer perspective, the session slots provide an excellent metric for expressing how many sessions a particular service instance can deliver at any moment in time on a particular environment. For example, if a service instance detects problems with respect to the underlying resources (e.g. due to resource contention), it can reduce the number of available session slots, compensating for these issues. In worst case, it can reduce the available session slots to zero, forcing the FUSION load balancer to use other instances within the zone or even coming from another zone. On a longer time scale, the evaluator service can easily take into account historical data from previous runs of a service in a particular environment to assess the number of slots (if any) that such environment can deliver for that service with proper QoS.

From an infrastructure and platform perspective, an infrastructure provider may have incentives for providing proper QoS towards particular services to avoid revenue loss to other infrastructure providers. A heterogeneous cloud platform that can balance the service provider and infrastructure provider expectations through appropriate optimization and tuning can result in a significant differentiating advantage.

Lastly, we also envision that FUSION could provide particular FUSION service APIs for improved integration and acceleration of specific functions. For example, as we will discuss in Section 3.7, different inter-service communication channels can have a significant impact on bandwidth and latency characteristics. Some FUSION execution zones implementing some of such capabilities could provide them via a standard API, shielding the services from the lower-level details. Different zones

could have different offerings, which could be detected by the service, for example through the evaluator services.

2.3 Composite services

2.3.1 Overview

In Deliverable D3.1, we described a wide range of possible variants of service graphs, related services and service instance graphs, ranging from the description and instantiation of static service graphs over template service (instance) graphs, to fully dynamic service instance graphs.

As a reminder, an overview of the various terminology is depicted in Figure 3.

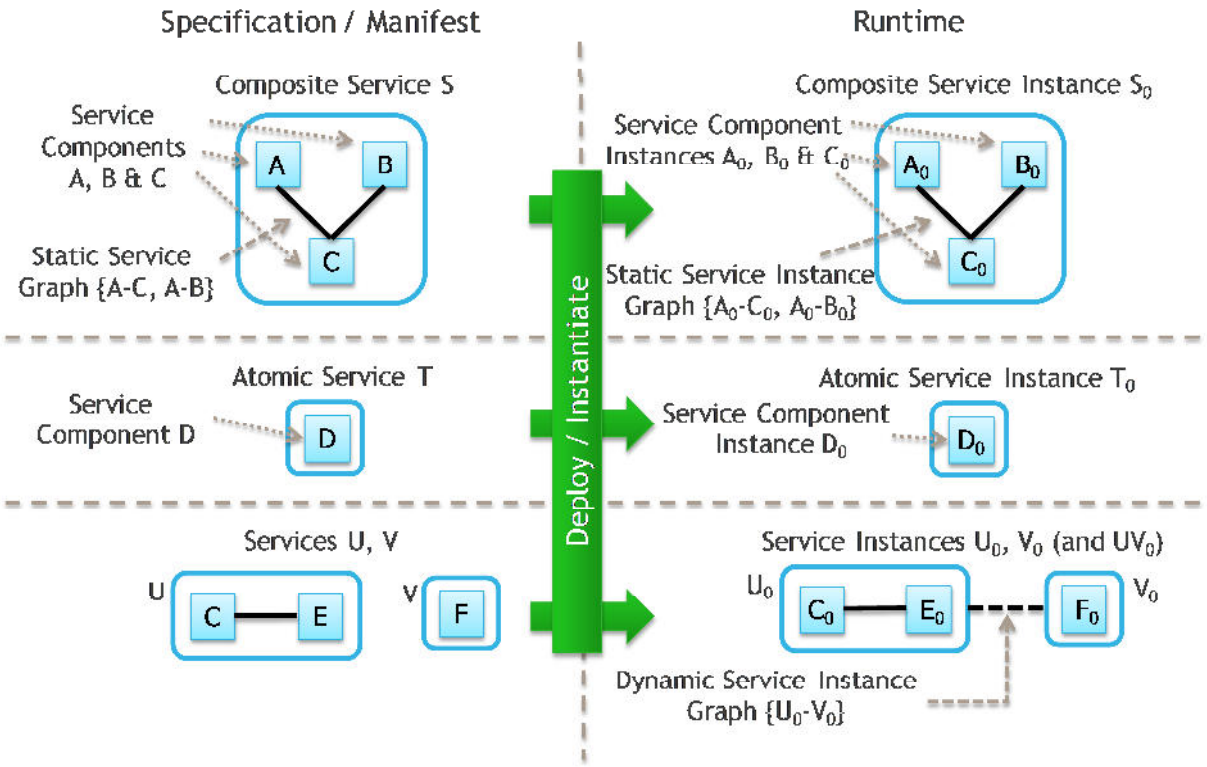


Figure 3 – Key FUSION composite service terminology

We clearly distinguish services from service instances. Services are the abstract representations of particular workloads and can consist of one or more service components, resulting in an atomic service or composite service, respectively. Each service component provides part of the functionality of the overall service and can either be reused across multiple services or can be deployed across multiple execution zones. A composite service consists of a (static) service graph, defining the relationship between service components that are expected to communicate for implementing the overall service functionality.

Service instances on the other hand are runtime instantiations of the corresponding abstract service specifications. One service can be instantiated any number of times, resulting in potentially large number of service instances of that service. Similarly to services, the instances consist of one or more service component instances, resulting in either atomic service instances or composite service instances. These composite service instances have a service instance graph that indicates the connectivity between various service component instances in the instance graph. This graph can be either static or dynamic in nature, meaning that the topology and corresponding connections either remain constant throughout the lifecycle of the service instance, or can change during the lifecycle of the service instance, involving different service component instances and connections at different

moments in time in the lifecycle of a specific service instance. In such scenario, different service instances can have radically different dynamic service instance graphs.

Of all possible variants, we decided to focus on two key types of composite services this year:

- **Static service graphs**

  In this case, the topology of the service is static and described in a TOSCA manifest (see Section 2.4.1). Although the different service components of such composite services could be deployed in a distributed manner across multiple execution zones, we decided to focus first on deploying such composite service within a single execution zone. Future work includes investigating how our placement and deployment algorithms and strategies need to be extended for placing and deploying composite services in a distributed manner across multiple execution zones.

- **Dynamic service instance graphs**

  In this variant, the graph is constructed after instantiation of a service, in which case the service components can (dis)connect to other service components dynamically. Note that this variant is complementary to the static service graphs, meaning that multiple static service graphs can also dynamically form larger dynamic service instance graphs by connecting to each other at runtime.

  We envision two key approaches on how instance graphs can be constructed at runtime:

  - **Via FUSION service resolution**

    We envision that a very natural and powerful way for creating dynamic instance graphs in FUSION is for FUSION services and service components to take advantage of the existing FUSION service resolution capabilities. For example, when a service *X* wants to make use of service *Y*, it uses the FUSION client API for requesting the FUSION resolution plane to find an optimal instance of service *Y*, possibly even relying on FUSION to automatically deploy a new instance of *Y* when no appropriate instance exists or has available slots.

    FUSION service resolution can be used in two ways, namely a simple and a more complex scenario. In the simple scenario described, individual service instances independently use FUSION service resolution for finding their optimal neighbour. For example, service *X* finds an optimal instance of *Y*, and *Y* finds an optimal instance of *Z*. Apart from being quite simple, a possible disadvantage of this method is that only the connections in between two neighbouring service instances are only piecewise optimized. Depending on the overall functionality and complexity of the dynamic service graph, this may result in suboptimal graphs and operation. However, in D4.2, some algorithms are being studied and described for service-graph based service resolution.

  - **Via a choreography component**

    This can be mitigated in a more complex scenario, where a special choreography or workflow component is responsible for the coordination and management of the dynamic service instance graph. Having the global overview and being aware of the overall application targets, this coordinating component could leverage service resolution a number of times and construct the optimal instance graph itself, rely on FUSION orchestration, or take all decisions manually. An example application pattern that could make use of such scenario is the lobby pattern, where a game lobby component is responsible for finding the optimal locations of the game server and possibly also the individual game rendering services for each client.

    This scenario has the advantage of allowing full flexibility at the expense of requiring orchestration or coordination functionality inside such components. Note that we envision that for simple patterns, a service may rely on a generic or third party choreography component rather than requiring each service provider to always implement and provide

their own choreography components. We will discuss the role of such choreography component further in Section 2.6.3.

Apart from the various types of service composition, we also investigated whether and how different (composite) service types or instances that have common service components could make efficient usage of already deployed instances of such service components, without significantly complicating service placement, deployment as well as application service design. The concept of "Multi-service configuration instances" provides an elegant solution to this problem.

2.3.2 Multi-configuration service instances

In Section 4.1.1 of Deliverable D3.1, we already described a number of variants for (incrementally) deploying composite services, reusing existing service components wherever necessary. Additionally, in Section 4.1.6 of Deliverable D3.1, we also discussed the issue of addressing composite services. Two possible approaches that were put forward in that Deliverable (see Figure 23 of D3.1) involved either providing NAT-like functionality in the Zone Gateway of the ZM, or requiring that composite services include an implicit gluing component that forwards incoming service requests to the appropriate service component.

In this section, we describe an elegant and efficient concept that tackles some of these issues by means of service overlaying. In this example, we assume a static service graph, as for dynamic instance graphs, the graph is constructed and configured at runtime, either explicitly using a coordination component or implicitly in between the individual components. In both cases, the instances of each component will be configured via session instantiation parameters passed along when a session is created, or based on session-specific events (e.g., a user trying to watch a movie).

2.3.2.1 Basic concept

The core idea is to allow more than one service (configuration) to be overlayed on top of the same component instances. In other words, to allow service component instances to be able to handle user sessions that *belong* to different (composite) services types or instances, each with their own set of instantiation and configuration parameters. The key enabling concepts to efficiently implement such mechanism are:

- **Decoupling the available/used *resource* slots from the available/used *service* session slots.**

  Previously, we assumed that one instance of a service component has a number of available slots depending on the service and the available resources, and that this was identical to the number of parallel sessions that the instance could handle (in the context of the corresponding service associated with that instance).

  By allowing one instance of a service component to be part of multiple application services instances, represented by a particular *service configuration* (i.e., service specification and instantiation parameters), the available number of r*esource slots* can be shared across multiple services, each having their own internal configuration for handling a particular application service. A resource slot is an abstract representation of the number of parallel sessions can be handled by a service component for the given set of resources, not in the context of a specific (composite) service.

  For example, imagine an instance *A* of a streamer service component that is capable of hosting up to 4 sessions in parallel. This instance could be configured to be part of an EPG service as well as a dashboard service, where the EPG service configuration may for example use all available streamer sessions, but the dashboard may only use maximum 2 of those resource slots from the streamer component instance. Depending on the session configuration, the streamer will either operate as part of the EPG service component or the dashboard service component, possibly with different encoding parameters, based on the incoming service request. At one moment in time, all four resource slots may be occupied by EPG sessions, in which case no slots are left for

the dashboard service (which needs to be taken care of by the ZM scaling and gateway components). At another moment in time, the dashboard service may be occupying its maximum of 2 resource slots, leaving only 2 available streamer resource slots for the EPG service (at least, from the perspective of that specific streamer instance.

Note that it is the responsibility of the service scaling component in the Zone Manager to decide to what extent to oversubscribe resource slots across multiple services. Note also that this mechanism also works very well in case the various service configurations represent different flavours of basically the same service type, but with different service names.

- **Uniquely identifying different service configurations by mapping them onto different ports.**

  Apart from allowing instances to be part of multiple services or service configurations, another issue is how to identify an incoming service request to be part of service configuration A or B. One possible solution is to simply map each service configuration on a different port or endpoint. For example, the streamer instance listens on port 5001 for incoming requests of the EPG service configuration, and on port 5002 for incoming requests for the dashboard service configuration. This is however not a requirement. Some services may implement this by adding a dedicated field in the signalling message. For example, one could also include the service name as a header field, identifying the specific service for which the client is making the request. With this feature, we do however allow for the possibility of multiple services to be mapped onto different endpoints of a particular service component instance.

- **Reporting both resource slots as well as all service session slots.**

  Instead of only reporting its available session slots, a service instance will report on its available/used resource slots as well as available and occupied session slots for each service (configuration). This provides the Zone Manager (and ultimately also the Service Resolvers) with accurate information regarding the available session slots for each individual (composite) service (configuration) and the overall resource utilization level of the physical component instances (e.g., for scaling purposes).

Imagine the following simple example, depicted in Figure 4, where the arrows represent the signalling paths. In this example, there are two composite services, namely *S* and *T*. As can be seen, *S* and *T* have two service components in common, namely *A* and *C*. Ideally, a Zone Manager may want to be able to share one or more instances of *A* and *C* across particular instantiations of services *S* and *T* to reduce the amount of active service component instances.

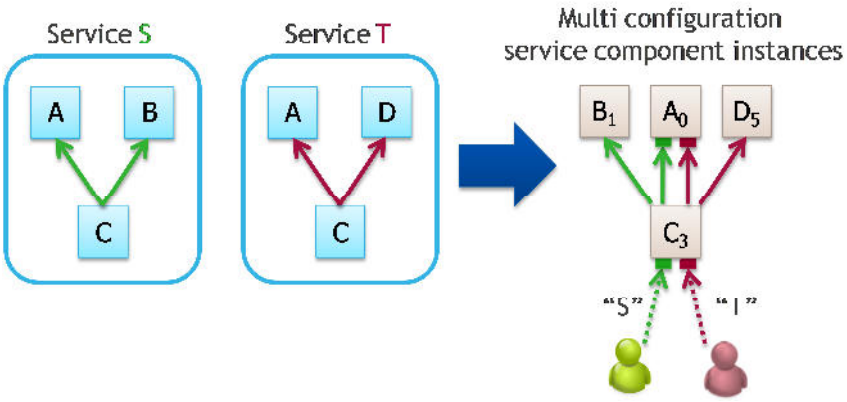


Figure 4 – Conceptual example of multi-configuration service component instances

Note that it is up to the Scaling component in the Zone Manager to decide which instances of what components will be shared across what (composite) services. The Scaling can decide to only share instances of component *C* across service instances of *S* and *T*, but not to share instances of component *A*. Note that in case the instantiation parameters for component *A* would be identical, or

in case *A* is in fact a proper service on its own (i.e., service component and parameters), then only one service configuration for *A* is needed in Figure 4.

2.3.2.2 Benefits

The key envisioned benefits of service overlaying include the following:

- **Reuse of available resources**

  This multi-configuration concept can potentially result in a significant reduction in the number of active physical service component instances, reducing the amount of resources that need to be reserved at any given moment in time. This is discussed in further detail in Section 4.1.

- **Controlled session-based resource oversubscription**

  This concept allows for service specific resource oversubscription in a more controlled manner (i.e., via session slots), rather than the current service agnostic resource oversubscription found in cloud platforms (e.g., based on physical CPU oversubscription). By implementing the oversubscription at the session slot level, FUSION can still guarantee good QoS compared to classical cloud resource oversubscription, which is especially advantageous for demanding applications. This mechanism however does require that multiple service instances can share the same physical service component instances. In other words, FUSION services should be decomposed into highly reusable service components to maximize the benefits of multi-configuration instances.

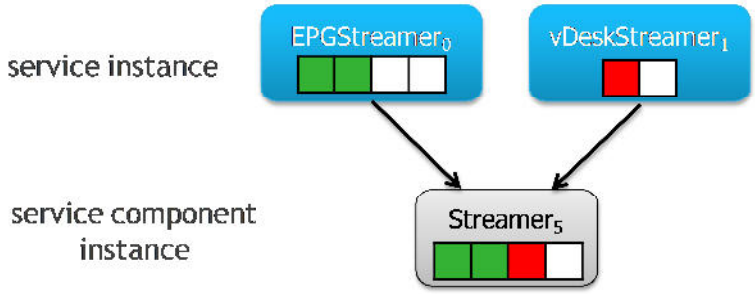


Figure 5 – Example scenario where two high-level (composite) service instances share resource slots with the same streamer service component instance

  An example is depicted in Figure 5, where the resource slots of a *streamer* service component instance are reused for two service instances, namely an EPG and Virtual Desktop streaming instance. The EPG instance can leverage up to all four of the streamer resource slots (of which it is currently consuming two slots), whereas the other service can consume up to two slots of the shared streamer component instance, of which it is currently using one slot. It is up to the Zone Gateway to provide proper load balancing, ensuring that an incoming service request is not forwarded to a component instance of which all resource slots are already occupied.

- **Faster and more flexible *virtual* elastic scaling across running instances**

  This mechanism also allows for a different type of service scaling, without actually needing to deploy or terminate physical instances, resulting in a more stable environment (i.e., a reduced number of deployment or terminations) as well as a faster scaling in or out of FUSION service session slots. We call this a *virtual* elastic scaling based on multi-configuration session slots.

  For example, looking back at Figure 4, imagine a number of instances of components *A*, *B* and *C* with a sufficient amount of available slots are already deployed in a particular zone, and the Zone Scaler decides that more instances of Service *S* need to be created. Then the Zone Scaler could decide to add service configuration *S* to a number of instances of *A*, *B* and *C*, and configure how many service session slots each selected instance should provide. In this case, the Zone Scaler does not need to deploy new instances of *A*, *B* and/or *C* to scale out service *S* in the zone,

potentially significantly reducing the scaling delay when new instances and resources need to be created. A similar strategy could be used for scaling in: instead of physically terminating instances, the Zone Scaler can also decide to simply reduce the number of available session slots for that particular services *S* in each of the components *A*, *B* and/or *C*, or even trigger the removal of that service configuration in a number of instances, while the instances themselves keep running (to handle other service configurations).

2.3.2.3 Implementation

We already implemented and validated this concept of multi–configuration instances at all layers of the prototype:

- We extended the session slot based service factory in some of our demonstrator service components to be able cope with multiple service configurations, such as adding new service configurations at runtime, sharing the resource slots across multiple service configurations, and reporting the resource and service session slots to the Zone Manager.
- We extended the DCA prototype to be able to add new service configurations to running instances (via a key/value store), and augmented the DCA APIs to be able to receive such requests from the Zone Manager.
- We extended the Zone Manager prototype to be able to detect and exploit service component instances across multiple services, and trigger the DCA to add a new configuration to an existing instance.
- We extended the prototype service manifest descriptions to be able to support this service overlaying concept, as well as allow a service provider to specify whether particular service (components) supports and enables this concept. In other words, it is not required for all services to support and implement this feature (though it can be supported with minimal effort).

2.4 Service registration

One of the key external interfaces to service providers is the service registration interface. This allows (internal or external) service providers to register their service into a FUSION domain to make use of the FUSION platform. Once a service has been registered, the FUSION service management layer will automatically deploy and manage that service according to the specified service requirements and policies. A service provider can monitor the actual state of all services registered in a FUSION domain and provide updates if needed.

As FUSION is responsible for the lifecycle management of these registered application services, all information regarding the service, such as the service graph, requirements, and policies, should be described in a service manifest. As a result, the key interface for service providers to FUSION is the registration of a fully-contained service manifest, and in the following sections, we will elaborate on this FUSION service manifest.

2.4.1 Service manifest

A FUSION manifest captures an entire service description (graph) and fusion specific information such as session slots and input on how to deploy. The manifest is a specification of design time parameters and input for deployment.

Concerning service graphs, mainly 2 types of graphs will be distinguished, notably statically described graphs and dynamically resolvable static graphs whereby the distinction between static and dynamic is based on the data plane resolution aspect.

- **Statically described graphs** are defined as the full description of a service and how the service components are composed/interconnected with one-another. The data plane communication

paths or routing are fixed for the service instance lifetime and so within a service, the service components collaborate over the service instance's lifetime.

- **Dynamically resolvable static services** (with a statically defined service graph between types and dynamic FUSION resolvable interconnections between instances) are defined as a full description of all service components that make up a service and how the service components collaborate with one-another. The data plane communication paths are determined and evaluated at each service invocation. In Figure 6, the service graph with its dashed interconnections specifies how components collaborate and therefore allows for the FUSION framework to account for locality and proximity of the different components.

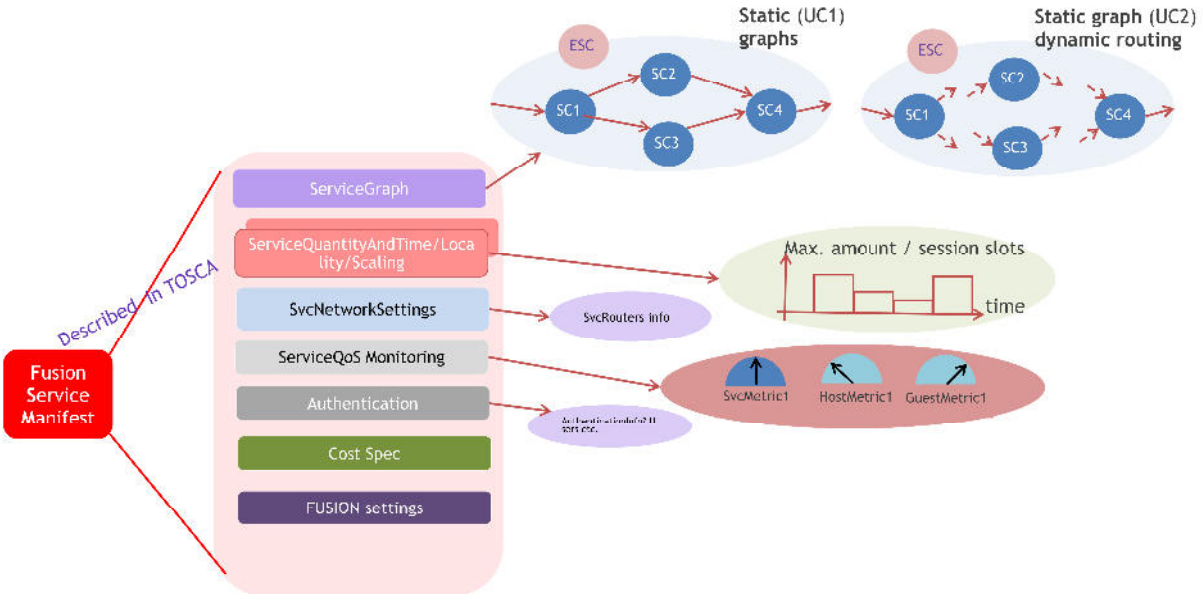


Figure 6 – High-level representation of a FUSION manifest

The service manifest has to cover Atomic Service Scenarios (ASS) as well as Composite Service Scenarios (cfr Section 2.3). Specifically for FUSION, the concepts of session slots and service evaluators also need to be contained in the service manifest. The FUSION manifest also contains information related to service QOS and resource monitoring, service scaling information (including geographical distribution and constraints), platform resource monitoring as well as costs and economical information (cfr. Figure 6).

TOSCA [TOSC01] is an existing specification language focusing on the portability of cloud applications and services. TOSCA enables an interoperable description of application and infrastructure cloud services, the relationships between service components, and the operational behaviour of these services (e.g., deploy, replace, shutdown), while being agnostic about service creator, specific cloud provider or hosting technology.

A FUSION service manifest will be described using the TOSCA language, more specifically using the standardized TOSCA simple profile in YAML v1.0 notation [TOSC02]. The simple profile is extended with the necessary FUSION concepts.

In the following paragraphs, a high-level description on the FUSION TOSCA manifest is given along with the manifest processing flow through the FUSION framework components. A full FUSION TOSCA manifest description example can be found in Section 7.6.

2.4.2 General FUSION manifest processing flow

When a request to deploy a TOSCA manifest occurs, the TOSCA manifest will be the blueprint that describes how all of the functional components (FUSION framework, data centre layer, OpenStack, ...) need to be invoked to ensure proper service deployment and service operation. The description will focus on the deployment aspects.

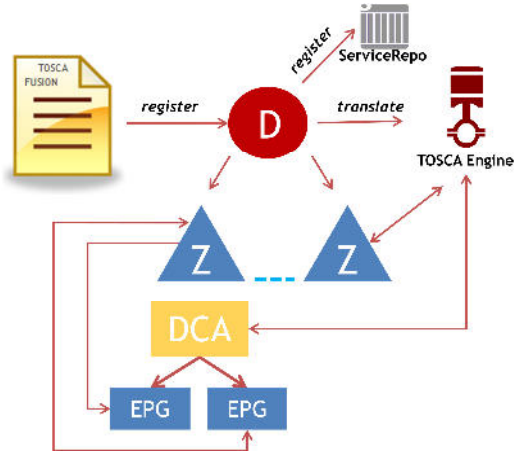


Figure 7 – Role of a FUSION manifest in the FUSION architecture

On a general level, a registration and deployment flow is described, abstracting out any complexity and implementation specific details (these will be handled in the sub-paragraphs on a per topic basis).

The manifest is registered at the FUSION Domain Orchestrator. Upon receipt of the manifest, the Domain Orchestrator will store the manifest in a service repository. This allows for inter-manifest referencing of services and service re-use at a description level (which can be leveraged at operational level via re-usable services or re-usable service components).

The domain orchestrator contains or uses a TOSCA orchestration engine that is capable of interpreting a FUSION manifest, steering the FUSION framework and steering the Data Centre Adaptor.

In the case of OpenStack, the service graph component in the manifest will be translated into a Heat Orchestration Template (HOT) and invoked on the OpenStack HEAT component. HEAT will provide the deployment of the service graph onto the cloud it controls. (for a description on HEAT architecture, cfr. [HEAT01] [HEAT02]).

HEAT contains the necessary functionality to deploy Docker containers as software components and is provided as a plug-in. Through HOT templates wherein specifying the nova server type as DockerInc::Docker::Container, containers can be instantiated. The necessary configuration information such as IP endpoint information can be specified as well as the passing of environment variables. The latter functionality possibly allows for an environment variable to be set as e.g. a reference to the ETCD store where configuration and state information can be stored. Note: the TOSCA specification related to container technology is currently under study. The latest state can be found at [TOSC03].

After initial deployment of the service graph, a configuration stage is entered in the deployment whereby the services are customized during boot and configured (e.g. through the cloud-init interface[SKIN14] as available in OpenStack). It is at this stage that the necessary linking and interconnections with other services, service components and the FUSION framework components is configured and established and normal service operation can start.

2.4.3 Manifest service graph component

Practically, a TOSCA service manifest is described using a CSAR (Cloud Service Archive) and has a structure as given in Figure 8:

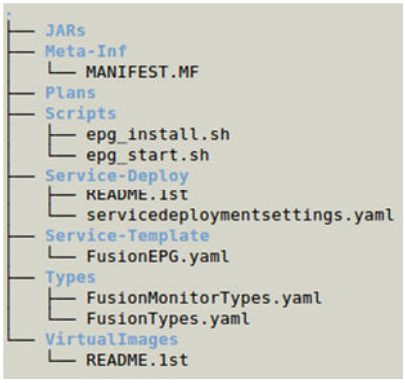


Figure 8 – Example FUSION CSAR file structure

The service-template section in the CSAR contains the description of the service graph and can rely on the types section; The types section contains descriptions of generic types.

As described in the previous Section, this part of TOSCA will eventually be translated into Heat Orchestration Template format in case of OpenStack. The translation is performed by a translator tool that is described in a OpenStack blueprint [HEAT03]. A complete description of TOSCA and TOSCA simple profile can be found at [TOSC01],[TOSC02].

The FUSION specific types that extend the current TOSCA standardization are:

- FusionService
- FusionComponent
- FusionEvaluator
- FusionSvcContextRepo
- FusionDomain
- FusionAuthentication

Their relation is described in the following UML diagram and focuses on the specific FUSION framework functionality:

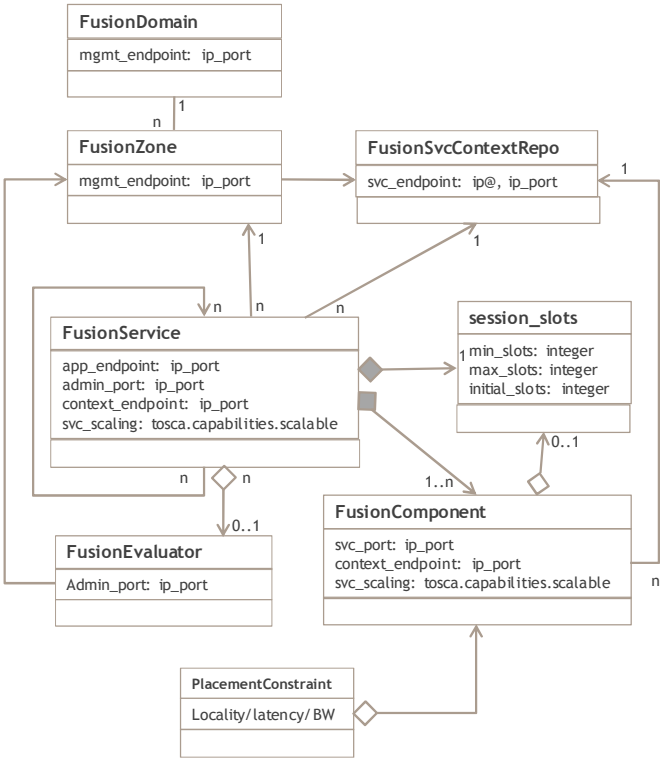


Figure 9 – UML diagram of key FUSION components and relationships

In the following paragraphs, we will briefly discuss topics concerning the various types related to specific FUSION functionality, and elaborating on the complexity that needs to be handled at the service description level. The service instantiation level and service operation level are not handled here.

A FusionService is the actual service that will be publicly addressable. Since the CSAR is submitted to a FUSIONDomain, and the FUSION domain will eventually select a zone to run the service (in case of intra-zone placement), the FUSIONService will get references to its FusionZone. Through self-referencing, composite FUSION services can be modelled. At description level, re-use of FusionService is realized by interconnecting FusionServices.

In case of a composite service graph with inter-zone placement, the inter-zone communication is established between FusionServices in the different zones. This allows the FusionDomain to map the different FusionServices onto different zones while deciding on the most optimal distribution.

Concerning composite services, upon registration and deployment request, the FUSION framework can determine whether a composite service and all of the necessary FusionComponents will be specifically instantiated or whether already existing FusionComponents and/or FusionServices can be re-used and be part of the entire FusionService.

To allow the FUSION framework to perform adequate placement, resource and communication information is to be provided to the placement component. In order to provide this information, 2 possible approaches where evaluated.

A first approach is describing resource and communication information as part of the service manifest and which is the approach taken in current TOSCA standardization. Under the form of “constraints” that can be attached to different kinds of information elements (e.g. “HostedOn” and “ConnectsTo” relationship types) or using the “Requirement Type Hierarchy” to set requirements.

This approach has been worked out in detail in the service manifest description in the Appendix in Section 7.6 and covers the aspects of late binding as described in Section 3.7.

An alternative approach is to use runtime information as provided by evaluator services. Evaluator services (ES) can leverage information collected from the environment to determine functional and operational suitability to run (a) FUSION service(s) in an environment and provide this information to the FUSION framework.

Throughout this document, evaluator services have been described at the level of estimation of execution of a service in an environment and these topics also apply here and will not be repeated. The concept of ES as collector and reporter of runtime conditions could be extended to provide information on the communication aspects of services.

The information that could be provided are available networking/communication capabilities and actual bandwidth, latencies and jitter. It allows for evaluation of actual communication patterns that a specific service or composite service exhibits.

There are however some possible issues with this approach: for example, what if the communication path only gets instantiated at service instantiation (e.g. ALU Nuage approach whereby communication path provisioning is part of service instantiation). In the case of composite services, should all possible interconnections be tested? This would require the presence of ES and full mesh testing between on all possible service hosts. Both the presence of ES at each eligible location and network testing between them incurs a considerable effort(/overhead) to determine optimal composite service deployment. Furthermore, in case of composite services, ES evaluate of the different possible locations will be generating a score, how should this score be reported and how can it be made comparable against each other to allow fusion to select the most optimal communication. We will address some of these issues in Section 2.7.4.5. Others will be subject to further study.

2.4.4 Service Deploy component

This part is interpreted by the TOSCA engine, which is the functional FUSION component that is able to parse the CSAR, and deals with service capacity over time. Since in FUSION, service capacity is specified by session slots, the expected load patterns are specified in terms of the number of session slots at different moments in time (and space).

2.4.5 Additional artefacts

A CSAR file can also contain a number of artefacts, containing additional input files, scripts and other configuration files that services can use during deployment. The service VM or container images can be externally referenced or included into the CSAR.

2.4.6 Multi-domain service registration

In a multi-domain scenario, where services could be reached from within other FUSION domains (see also Deliverable D4.2 for more details), extra care needs to be taken when registering a new service with a particular name into a domain. In case services are only visible within a single domain, the domain can easily verify whether a service was already previously registered with a particular service name. However, in case services can be visible across multiple domains, then an additional verification mechanism need to be put in place to avoid naming collisions. We conceived two main options. A first option would be for the domain orchestrator to look up the service name in the service catalog of the service resolution plane (see Deliverable D4.2).

A better option however would be for service providers that want cross-domain visibility of their services to first register their service name to an external authority and insert this authentication information into the manifest. The domain orchestrator, upon registration, first verifies with the external authority whether that specific service provider can use that specific name. Note that this mechanism also enables a service provider to register the same service using the same name in multiple FUSION orchestration domains without causing any false collisions.

2.5 Service and Platform Monitoring

2.5.1 Monitoring challenges & requirements

As discussed in Section 2.2, it is crucial to monitor and combine both service, network and platform metrics for enabling cost-efficient utilization of available platform resources as well as provide appropriate QoS towards the demanding services.

In FUSION, we designed our architecture and conceived a number of key enabling concepts to facilitate this:

- **Session slots**

  As introduced in Deliverable D3.1, the core idea of session slots is to take into account the number of parallel client sessions a particular service instance can handle with good QoS in a particular execution environment. As such, session slots condense and abstract runtime information regarding resource utilization, efficiency and load.

- **Evaluator services**

  Apart from runtime information, FUSION also needs to be able to efficiently determine where to deploy particular services. For this, we introduced the concept of evaluator services in Deliverable D3.1, that abstracts the feasibility and cost-efficiency of a particular execution environment by an abstract score, which can incorporate both static as well as dynamic information. More specifically, it can capture whether a particular execution environment has the appropriate set of resources and capabilities, how many slots would be supported by that environment, and how good that environment behaved in the past for running that service. More details and discussion on evaluator services is provided in Section 2.6.2.2.

- **Service resolution**

  As discussed in detail in Deliverable D4.2, the main idea of service resolution is to take into account service metrics, platform metrics as well as network metrics for optimally distributing incoming client requests across running instances. Service and platform runtime metrics are covered by session slot information, whereas network information comes from the network monitoring infrastructure. A FUSION domain orchestrator also incorporates high-level network monitoring information for deciding where to deploy new service instances.

- **Adaptive heterogeneous cloud platform**

  As discussed earlier, demanding applications often have stringent requirements with respect to resource utilization and behaviour. To significantly improve overall QoS delivered by a heterogeneous cloud platform on which FUSION zones are deployed, we are working on an adaptive heterogeneous cloud platform that takes into account the application requirements, platform capabilities and tries to optimize the runtime behaviour of the underlying platform for such types of applications. More details are provided in Section 3.5.

2.5.2 High-level FUSION orchestration metrics

In this section, we zoom in a bit more on two key high-level FUSION metrics and discuss their role within FUSION for efficiently managing services and resources, namely session slots and evaluator services, operating roughly at different time scales:

- **Short-loop metrics**: active session slots

  Within an execution zone, session slots provide a very lightweight metric summarizing available capacity and load towards the zone manager components such as the Load Balancer and the Scaler. Incoming requests can easily be forwarded to instances with a particular amount of available slots, depending on the balancing policy. The Scaler within an execution zone can also

make usage of the active session slots and its recent history for quickly creating new (virtual) instances. A key benefit of session slots is that this information can be done on a discrete service-specific metric (hiding the resource requirements per active session), rather than estimating the effective load and impact on the service based on low-level resource metrics such as CPU or network load.

- **Longer-loop metrics**: evaluator service scores & session slots history

  Within a FUSION domain, evaluator services can be used for estimating where services can be deployed cost-effectively (given some longer-term demand pattern). This information can be augmented with summarized session slot utilization information coming from the individual zones to give high-level but relevant information regarding the resource utilization over time of particular services as well as their popularity.

The way these key metrics are integrated within the FUSION architecture is briefly discussed in the next section, and more in detail in the second part of this Deliverable.

2.5.3 FUSION monitoring

In Deliverable D2.2, we presented a high-level view of the FUSION monitoring architecture. In this monitoring architecture, monitoring data is exchanged in between all service layers. For each layer, we briefly comment on what type of monitoring information is expected to be exchanged.

2.5.3.1 Domain-level monitoring

Between a FUSION domain and execution zone, the key monitoring data that is passed along is twofold:

- High-level session slot availability information is passed from all execution zones to the domain orchestrator, allowing the domain to take global decisions for increasing or decreasing the number of available session slots in particular regions, based on predicted load and networking information.
- For placement decisions, the scores of the evaluator services are passed back from the execution zones to the domain orchestrator, allowing the domain to optimally place services across the available zones, without requiring detailed information on the resource capabilities and availabilities in each zone. Using these evaluator services, the amount of monitoring data that needs to be exchanged between a zone and a domain can be significantly reduced and in fact is completely abstracted away, resulting in a much more scalable (and fine-grained) solution, that can take into account the heterogeneous nature of the different execution zones.

2.5.3.2 Zone-level monitoring

To efficiently manage services and resources within an execution zone, we envision two types of monitoring data to be necessary:

- **Service session and resource slot information**

  The availability of available service session slots and resource slots can be used by the zone manager both for internal scaling and load balancing purposes. This session slot information comes from the deployed services and provide the zone manager insight in the availability and resource utilization of the locally managed services.

- **Platform and execution environment information**

  For deploying new service instances, the zone manager needs to be aware of the availability of physical (or virtual) resources. To avoid that a Zone Manager needs all detailed information of the underlying resources and its current state, we abstract this by representing different types of resources or hosts as different *execution environment types*. Each type of environment may have

a number of properties attached to them that describe some of the higher-level properties of that execution environment. Of a particular type of execution environment, there may be many instances (or actual environments).

Very detailed information of these execution environments is not needed, as the evaluator services can be used to abstract all infrastructure details (e.g., how efficient is a particular environment for running a particular service). It is the task of the underlying DCA layer to present an underlying DC as a set of (more abstract) execution environments, each belonging to a particular environment type.

In theory, more than one FUSION service could be deployed in one of these environments. In that case, the DCA should also provide high-level utilization information concerning the load of that environment. This could be represented as a percentage; it is then up to the zone manager to learn how many instances of a service could be deployed in a particular environment. However, for now, we can safely assume that one environment hosts a single service instance, and a service instance determines the number of available session slots based on the available actual resources within that execution environment.

A zone manager may also be able to create new execution environments in the underlying DC (via the DC) in case there is no fixed 1:1 mapping between the available physical resources offered by the zone and the underlying DC resources. For example, a zone deployed on an existing cloud platform may decide to dynamically grow (or shrink) the size of the zone within that cloud to be able to deploy more service instances.

2.5.3.3 DC(A)-level monitoring

The Data Centre Adaptor (DCA) layer is a(n) (optional) layer in between a FUSION zone manager and the underlying DC. The main purpose of this layer is to abstract away the underlying lower details w.r.t. the resources and deployment mechanisms of the underlying DC from the zone manager, of which the main responsibility should be to manage the higher-level aspects of deploying and managing FUSION services. See Section 3.4 for more details on this DCA layer.

The monitoring within the DCA layer (of the underlying DC) can be divided into two main categories. The first category is the monitoring for internal use and optimization of the underlying resources of the DC. This aspect will be discussed in more detail in Section 3.5 when discussing the heterogeneous cloud platform.

The second category of monitoring data is with respect to the execution zones that are deployed on top of these DCA layers. As discussed in the previous section, one of the tasks of the DCA layer is to abstract the different types of resources, hosts and infrastructures as instances of particular abstracted execution environment types. These environment types could be augmented with additional metadata, allowing a zone manager to categorize or cluster particular environment types based on this metadata. One example of metadata could be the availability of a GPU resource type in that a particular execution environment. Services are deployed in such environments; the resource utilization and efficiency is reflected in the session slot utilization and availability.

2.5.3.4 Service-level monitoring

As discussed already before, the main monitoring data expected from the services is the session slot availability information. This high-level abstract information abstracts all service-specific resource requirements and is used by almost all FUSION layers for efficiently dealing with service and resource heterogeneity.

The evaluator services on the other hand can be considered to be service specific probes that can be deployed in different environments for assessing that environment in the context of that service and abstracting all detailed knowledge by a simple score. Upon this score, a service can be efficiently deployed across a heterogeneous set of resources.

2.6 Service lifecycle management

As detailed in Deliverable D2.2 and D3.1, the lifecycle of FUSION services in the FUSION architecture is managed at three levels, each covering some aspect of the lifecycle of FUSION services:

- **FUSION domain layer**

  At the domain layer, service lifecycle management is focused on ensuring that an appropriate amount of session slots of all registered services are available in particular regions, or that new instances can quickly be deployed on-the-fly when necessary.

  A key role of a domain orchestrator thus is coordinating global service lifecycle management across all execution zones, while leaving the lower-level details to the lower layers for scalability and efficiency reasons. To be able to efficiently deploy FUSION services across multiple zones, we developed the concept of evaluator services, which allows to take into account zone and service specific requirements in an elegant and scalable manner at the domain level.

- **FUSION zone layer**

  Being in between the domain and DC layer, the main role of lifecycle management at this layer is to coordinate and manage the various FUSION instances of services, (pre)provisioning them on the appropriate physical or virtual environments, etc. This may include sharing particular service component instances across multiple (composite) services.

- **DC layer**

  This layer deals with the low-level aspects of service lifecycle management, such as configuring and managing the physical and/or virtual environments, allocating and managing the corresponding resources, starting, stopping or migrating services, status monitoring, etc.

In the following sections, we discuss service (pre)provisioning, various deployment strategies and implementations, and the impact of composite services.

2.6.1 Service provisioning

We define service provisioning as the act of preparing a particular (physical or virtual) environment to deploy a new instance of a particular service. This includes the following key steps:

1) Fetching all necessary VMs, containers, files and/or other artefacts needed to deploy and start the service;
2) Select and allocate all necessary (physical and/or virtual) resources that will be assigned to the new service instance, creating a new environment for that service;
3) Prepare and configure the newly created environment for the service so that the service can be started afterwards.

Especially the first step can be quite time-consuming. For example, imagine a VM image of 1 GiB that needs to be fetched from a (centralized or decentralized) repository to a particular (node in an) Execution Zone. Even at 1 Gbps, this would still take about 8 seconds, and at 100 Mbps, this would take up to 80 seconds. As such, efficient service provisioning mechanisms such as (i) preprovisioning and (ii) more lightweight packaging mechanisms can significantly reduce the provisioning time and bandwidth requirements, which is especially important in the on-demand deployment scenario.

Preprovisioning strategies can be implemented as part of the Service Scaler component (both at the domain and the zone level) by taking into account the expected provisioning delay and the deployment scenario, leveraging historical data to learn based on past behaviour.

Another strategy is to reduce the size of the service data that need to be fetched in order to provisioning a service onto a particular environment. Two primary mechanisms we will briefly discuss are the following:

- **Incremental download**

  Instead of creating full-blown VM images that contain all data, one possibility for service providers is to create minimal VM images that only contain the bare minimal for starting the service, and to incrementally download necessary additional libraries and data just-in-time. For example, if the base VM image size could be reduced to 100 MiB, then the time to fetch an image could be reduced to below one second in case of 1Gbps download bandwidth. Two possible downsides of this approach are however:

  - *In-session delays*

    Clients may experience possible delays and stall times while accessing the service, especially when they connect almost immediately follow after a new instance is deployed (e.g., in the on-demand scenario);

  - *Limited caching*

    In case all data is within the VM, once the VM is downloaded and cached into an execution zone, all subsequent instantiations can be much faster, amortizing the provisioning penalty across multiple instances. In case of incremental downloads, large chunks of data are fetched after service instantiation, meaning that each service instance will need to fetch these additional data chunks. If these data chunks cannot be cached somewhere locally, this may easily result in a larger bandwidth consumption as well as potential in-session delays and stalls experience by all first clients (i.e., the clients that occupy the first session slots of an instance).

  Note that this mechanism does not necessarily require specific support from FUSION to enable such mechanism (though FUSION could take it into account for its preprovisioning and scaling strategies).

- **Stackable VM images or containers**

  A second technique is to allow multiple VM images to share common data chunks, either by expressing different VMs as deltas with respect to each other or with respect to a common base image [SAT09], or by using a stackable file system and imaging system as is used with Docker containers [TUR14], where an container image is constructed by stacking a number of image layers on top of each other, where each of the layers can be shared across other container images in a tree-like fashion. A general example is depicted in Figure 10. In Figure 36 in Section 4.2, a detailed example is shown for some of our FUSION prototypes. In case of Docker, the base layers are typically shared by large numbers of different containers, where only the application-specific layers need to be (pre)provisioned when a new container instance is to be deployed. This can even impact service placement decision within a Execution Zone, as it may be preferable to quickly deploy a new service on demand in locations where the amount of provisioning is minimal to reduce startup delay.

  As Docker automatically fetches and caches the necessary layers, this provides the benefits of incremental downloads (but now at provisioning time) without losing the benefits of caching. Obviously, this mechanism can also be combined with incremental downloads to fetch session-specific or user-specific data.

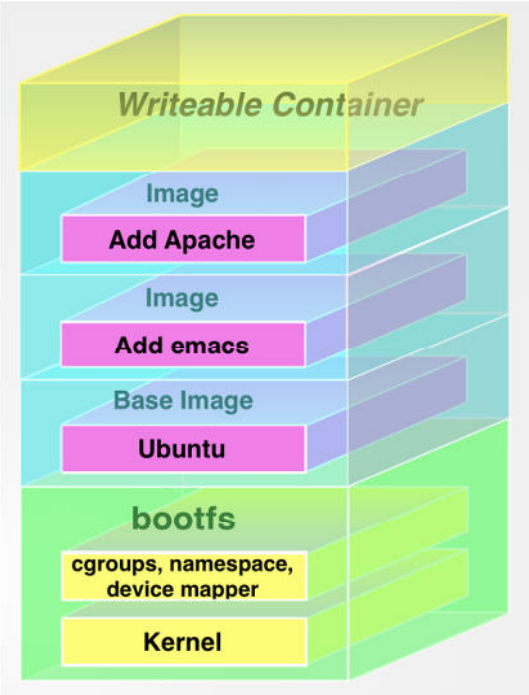


Figure 10 – Docker container stacked image layers

In FUSION, we actively promote and investigate the capabilities of lightweight containers such as Docker for efficiently packaging, provisioning, deploying and running FUSION services. Some of the experimental results are described in Section 4.2 as well as in Deliverable D5.2.

2.6.2 Service deployment

Service deployment is the general process of creating, configuring and starting new instances of particular services on a particular execution environment. Similarly, service termination is the reverse function of stopping, destroying and removing instances and/or their corresponding environment.

As discussed in Deliverable D2.2, we envision two main deployment scenarios, both at the domain level as well as zone level, though the impact will be more substantial at the domain level compared to the zone level:

- **Pre-deployed services**

  In this scenario, services are deployed in advance (or terminated afterwards), based on predicted load patterns across execution zones (or across nodes in a zone). In this scenario, we assume there is sufficient time for provisioning and optimally placing instances of services, i.e., there is no stringent time pressure.

- **On-demand service deployment**

  In this scenario, services need to be deployed as fast as possible in a particular region. This can be triggered either due to unexpected flash crowds in particular regions or within a region (even for very popular services), or to support a lazy deployment model for the long tail of unpopular services for which having at least one instance on standby in all possible regions of a FUSION domain would be too expensive for a service provider.

Two crucial aspects that need to be considered when deploying a service are:

1) Where to deploy the new instance (in what zone, in what execution node, etc.)?
2) How to configure that environment for optimally deploying an instance of that?

The first aspect is strongly related to service placement and is discussed in more detail in Section 2.7. Two key enabling FUSION concepts include evaluator services as well as multi-configuration service instances (i.e., service overlaying), on which we elaborate later in this section.

The second aspect is regarding how a zone or node needs to be configured and prepared for deploying that service. This is discussed in more detail in Section 3.5. In the next section, we first discuss the concept of lightweight containers as a new lightweight isolation and deployment mechanism for services.

2.6.2.1 Lightweight containers

A classical cloud platform typically uses full-system virtualization (using either a type 1 or a type 2 hypervisor [ALA09]) for packaging, sandboxing and deploying services on top of physical resources. In this model, each virtual machine encompasses a virtualized resource model (CPU, networking, storage, etc.), a guest OS, the application and its runtime environment. This allows for ultimate flexibility at the expense of additional overhead with respect to resource utilization, packaging and performance.

Lightweight containers provide a new deployment model, where applications are deployed and managed in lightweight isolated environments on top of a single host OS that supports such containers. In this mode, the resources are not virtualized, but they can still be isolated and configured for particular containers, enabling very fine-grained resource control mechanisms with low overhead. Thirdly, containers also enable lightweight software packaging models, as only the applications and their libraries need to be packaged (some of which can even be efficiently shared across containers), instead of full VMs incorporating their own OSes. A high-level comparison of the software stack between classical VMs and containers is depicted in Figure 11.

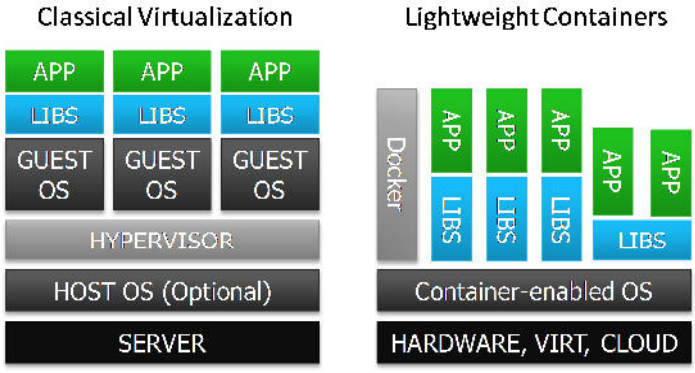


Figure 11 – Classical virtualization versus lightweight containers

Lightweight containers offer a wide range of benefits for different environments. A high-level summary of some key benefits is depicted in Figure 12.

	Key advantages of containers	IaaS	PaaS	Perf	Dev
Lightweight Virtualization	10x faster instantiation & termination	+	+	o	+
	Low resource allocation overhead (e.g., 10-100MB less mem/storage per virtual instance)	+	++	+	o
	Bare metal-like performance & predictability (e.g. < 8ms network latency with < 1ms jitter)	+	o	++	o
Resource management	More fine-grained accessibility, monitoring and control of specific resources	++	+	++	o
	Runtime adaptable oversubscription control	+	o	+	o
Lightweight software packaging	Disposable, isolated configured environments	o	+	o	+
	Sharing, easy deployment and extensibility of preconfigured contained environments	o	+	o	++
	Minimal packaging overhead & maximal reuse (container image stacks, bind mounts, etc.)	o	+	o	o

Figure 12 – Summary of key advantages of lightweight containers for different environments

In case of IaaS cloud environments, the key advantages mostly relate to virtualization and more fine-grained resource control, wheras for PaaS environments, the low packaging and runtime overhead are extremely beneficial. For performance sensitive environments, the bare metal performance and predictability provided by containers is key, whereas for devops, agile development and management are important.

With respect to FUSION, some of the key features include efficient packaging of service components, fast deployment (in distributed execution zones), bare-metal performance (for demanding applications) and corresponding fine-grained resource control.

Some remaining drawbacks of containers currently involve security, the shared OS and kernel and the lack of process management (which is less of an issue w.r.t. FUSION as FUSION provides its own orchestration and lifecycle management layers).

In FUSION, we actively focus and stimulate lightweight container for packaging and deploying FUSION (application and management) services, although we also support full-system virtualization. This raises the issue of how FUSION can deal with these different types of deployment, especially considering the fact that the zones and their underlying DCs can be very heterogeneous with respect to what service packaging and deployment models they support, potentially preventing particular services to be deployed (efficiently) in particular execution zones.

An overview of the various combinations and their implications is depicted in Table 1:

Table 1 – Compatibility of different service packaging/deployment models w.r.t. different DCs

DC	Lightweight containers	Classical VMs
Bare-Metal DC (MaaS)	• Easy to provide densely packed low-overhead base container-enabled bare images to deploy on the bare nodes (e.g., CoreOS)	• Need to provide a base image supporting the particular VM technology.
VM-based DC (Iaas)	• Create basic container-enabled VM images to deploy in whatever format is supported (cfr Heroku model, CoreOS-in-VM, etc.) • Efficiency and QoS of VMs depends on hypervisor of DC	• Compatibility depends on supported hypervisor and VM image format.

Container-based DC (Iaas & PaaS)	• Compatibility depends on what container technologies the DC supports. • Note that one of the true potential of Docker is their push towards a standard API for container mgmt: libcontainer [LIBC01]. • In the PaaS model, the efficiency and QoS depends on underlying resource management layers.	• Not easily supported (unless a virtualization layer is supported within these containers).

From the table, it can be concluded that containers provide more flexibly (and more efficient) deployment strategies, as it is more easy to embed containers in VMs on a hypervisor than vice versa. Due to the recent focus on standardizing the management APIs for containers, some of the issues regarding incompatible VM images and hypervisors can be avoided.

2.6.2.2 Evaluator services

We already introduced the concept of evaluator services in Deliverable D3.1. Its initial goal was to have a more flexible mechanism for determining whether a particular execution environment is suitable for deploying and running a particular service or not, rather than trying to capture all service requirements into a complex static manifest and require all FUSION orchestration layers to be able to interpret and deal with these complex manifests.

An evaluator service is basically an active service probe that returns a score regarding the feasibility of a particular execution environment for (cost) efficiently running a particular service. These evaluator services are deployed within the execution zones, can be provided by the application service and are triggered by FUSION prior to service placement and deployment.

For example, services may require specific hardware or a certain location near to other, already existing service instances or have other requirements towards the execution environment. These requirements may be related to specific hardware or may be runtime dependent, like the network distance to other services. For a rendering service for instance it may be important that a GPU or even specific GPU hardware capabilities are available and a streaming service providing a video stream that is merged into the rendering is located only at acceptable distance. Describing these kinds of restrictions in static manifest files will result in very complex rule sets on how the environment has to be checked before a service can be deployed onto it. The descriptions would also have to be maintained with each new hardware or even hardware revision that comes available.

Therefore the static approach using manifest files will not be sufficient. It will however be a good and efficient shortcut for simpler services. But to provide a solution that works also for more complicated services like the before mentioned rendering service, it will be necessary to physically test the environment before a service is deployed to it. For this reason evaluator services will be used.

Evaluator services implement logic that can test an execution environment for all features necessary to run the application. They can basically be a smaller version of the application. For applications that have the same requirements towards the environment, it would also be possible to use one single evaluator service. This may be the case for different software that are built on the same middleware with common requirements. One example are multiple games that are developed using the same game engine and hence have similar or even the same requirements towards the hardware.

After running, an evaluator service can calculate a rating for each execution zone it was executed on. In a naïve approach, an evaluator service will always return a higher ranking for a better hardware since it can achieve better performance on it. It therefore does not take into account the increased costs it causes to the data center operator or the service provider (depending on the business model)

by using better hardware. As one possible approach, the service could also take into account the different prices of the execution zone. This could be seen as a binding offer that the execution zone makes to the service. If the service is executed on this specific execution zone, the price must not change for a certain amount of time. These costs could then also be taken into account when calculation a rating for the execution zone.

All this information can be condensed into a single value, for example a normalized floating point number. By comparing the different scores from different execution environments, FUSION can easily find the best environment from the evaluator service's point of view.

For efficiency reasons it may be necessary to store the evaluation results and not run an evaluator service for each deployment request. Another optimization would be to run the evaluator service only once for different execution environments if it is guaranteed that they will physically behave the same. This requirement has to be handled very strictly, because the Zone Manager cannot know which features are tested by the evaluator service.

This is also true for caching the evaluator results. Any influence from outside which may have an impact on the evaluator results would require a new evaluation. More details on how evaluator services can be used for service placement is provided in Section 2.7.3.

2.6.2.3 Multi-configuration service instances

In Section 2.3.2, we already described the concept of multi-configuration service instances or service overlaying, where instances of service components can be part of multiple (composite) service configurations. As such, service deployment in FUSION not only entails deploying new physical instances (e.g., VMs, containers, etc.) in a new execution environment, but also involves configuring existing service component instances to take in a new service configuration and configure itself appropriately for hosting as well the new service configuration.

Similarly, terminating particular service instances may imply simply removing that service configuration from all involved instantiated service components rather than physically stopping and removing the running instance(s).

The overall process is summarized below:

- First, service components that support this feature need to specify this in their manifest description. Similarly, a service may specify whether existing service components can or cannot be reused for a particular service due to policy restrictions.
- The Scaler component uses this information to decide how to scale services across existing or new instances.
- In case the Scaler decides to reuse existing service component instances, it triggers the Deployment module to add a new service configuration with the provided parameters.
- This request is forwarded to the DCA, which will manage all low-level management such as adding the new configuration parameters into a corresponding key value store, which explicitly or implicitly triggers the instance to absorb the new configuration. The Deployment module adjusts its internal state accordingly.
- The service component instance configures itself to host the new service configuration. This may include doing internal bookkeeping, opening new ports to listen for incoming client requests for the new service configuration, etc. When the new service configuration is activated, the endpoint and corresponding available session slots are pushed to the Zone Manager, announcing the existence of new session slots of a particular service type.

- The Zone Manager will update its statistics accordingly, adding the newly available session slots to the existing ones for that service type (if already present), and provide an update towards the Service Resolution plane (either immediately or after some delay).

As discussed before, this allows for efficient scaling of FUSION services and reuse of running instances without directly impacting the amount of physically deployed instances. A key role of the Scaler component is to determine when to reuse existing instances and when to deploy new instances, based on the resource session slot availability and the predicted load patterns.

2.6.3 Composite services

In general, lifecycle management of composite services can very quickly result in scalability and complexity issues in case a FUSION domain orchestrator would have to coordinate all this complexity for every instance of all composite services. As such we need more scalable solutions that can apply to a large set of composite service patterns.

In this section, we will elaborate on this topic and present the current vision of the consortium. In the final year of the project, we will extend upon this vision and approach. We divided the discussion in a number of topics, namely regarding service deployment, session slot management, instance visibility and the choreography component.

2.6.3.1 Choreography component

For non-trivial composite services, we propose a choreography component that typically will be provided by the service provider. This choreography component will be responsible for the overall coordination, lifecycle and state management of the overall composition and may be involved in the load balancing as well. This orchestration and choreography could be done at deployment time, in which case the components and communication links are set up when the composite service is created, at service request time, in which case the choreography component configures particular instances and connections for that specific request, or even based on timing or event-based policies.

2.6.3.2 Service deployment and management

As mentioned in Section 2.3, we focused this year on two main classes of composite services, namely static service graphs and dynamic instance graphs:

- **Static service graphs**

  In case of deploying simple static service graphs, the overall deployment of such simple service compositions could be managed in a scalable manner by FUSION orchestration, either built into a FUSION domain (e.g., in case of inter-zone deployment) or into a FUSION zone (e.g., in case of an intra-zone deployment). The role of FUSION in such cases should however be limited to making sure that all components are placed, deployed and get connected to each other (i.e., you can find each other's endpoints), which could easily be done by setting up and passing along the location of a shared key value store that all service components can use for registering and finding the peering components. In this scenario, FUSION orchestration however should NOT be responsible for the further lifecycle management of such composite instances.

  When the service has a nontrivial static service graph or non-trivial lifecycle management, this should be handled by a choreography component, that could be provided by either FUSION (for simple standard patterns), a third party provider or by the service provider as a key component in the service graph. It could also be incorporated into one of the existing application components.

  This choreography component acts as a (possibly very service-specific) orchestration or workflow component that takes care of all complex details for managing and coordinating the various components. This component may interact with the FUSION orchestrator for optimally deploying

and placing new or finding existing components. With this choreography component, all state information is stored and managed within this component (in a distributed, decentralized and possibly service specific manner) rather than in the FUSION domain orchestrator, resulting in a much more scalable and flexible solution. This means however that a service provider may have to provide its own choreography component, resulting in a higher complexity for service providers. Third party providers could however provide standard implementations for particular patterns that handle this complexity.

- **Dynamic instance graphs**

  In case of simple implicit dynamic service instance graphs, the knowledge of the composite graph can be completely implicitly or explicitly encapsulated in the service component instances themselves. These instances leverage the FUSION service resolution protocols for finding and connecting to other service components. Each service component has service session slots of their own. The service overlaying feature (i.e., multi-configuration service instances) could be used for creating different specific flavours of particular basic services (e.g., an EPGStreamer or VDeskStreamer service, which are overlaid on top of a generic Streamer service).

  For more complex or centrally coordinated dynamic service instances, a choreography component could be used as well to keep track of the overall composition and state of each of the components. Due to the dynamic nature of such service graphs, such choreography components will very likely be service specific. This component may or may not have a full view of the entire composition and may also leverage some of the public FUSION orchestration protocols for optimally deploying and placing new instances of particular components, or finding existing instances via the service resolution plane.

In summary, for nontrivial composite services, we envision a choreography component that handles all the complexity, rather than the FUSION orchestrator, resulting in a more scalable and flexible (decentralized) solution, at the expense of pushing some of the complexities towards the service providers (though this approach also allows a lot of freedom to the service provider).

2.6.3.3 Service resolution and session slot management

With respect to service resolution and service session slot management, a FUSION service request typically returns one endpoint (or a list of endpoints) per service type. In case of a composite service, there can be two types of endpoints: the endpoints of each individual service component instance (which FUSION considers as instances of different atomic service types), as well as the endpoint of the composite service instance.

In case of the individual service component instances (that are part of a particular service instance composition), we envision that each service component itself can be considered to be an independent standalone service type, and this has its own FUSION service name and session slots. In general, there may be a case where some of these components should not be individually publicly visible via the service resolution plane. In such cases, it still makes sense to attach a(n) (internal) service name to such service components as well as internally manage its session or resource slots, but the zone gateway simply does not inject this service name and session slot tuple into the service resolution plane. Obviously, in such cases, these components will not be addressable via the service resolution plane (e.g., in case of dynamic graphs).

With respect to the composite service name, incoming service requests typically need to be forwarded to a particular component of the composite service. This endpoint can map onto one of the components of the composite service, which in many cases may be the implicit or explicit choreography component that will also set up all necessary connections when a service request arrives.

In Section 2.3.2 we introduced the concept of multi-configuration service instances, where we allow to overlay more than one service configuration/type onto a particular service component instance. This mechanism can also be used here to overlay the composite service (i.e., its name and available slots) onto one of the service components (e.g., the choreography component). As such, by leveraging this mechanism, we have a natural and efficient mechanism for routing service requests for composite services to an actual service component that is part of the composition.

2.6.3.4 Visibility of individual service (component) instances

In a FUSION execution zone, we assume that in many cases, all existing service (component) instances of a particular service type will not be individually visible by the resolution plane for scalability and efficiency reasons. Instead, for each service type, the number of available slots will typically aggregated per zone and only one (or a few) zone service endpoints will be directly addressable from outside the zone. This allows a zone manager to do local load balancing across the various deployed instances based on its own policies (instead of deferring this to the service resolution plane). In Section 2.9, we provide an example of how this could be implemented efficiently.

However, in case a client (or another FUSION service) needs to be able to have multiple data connections to different components in the service composition, where components also need to communicate with each other internally to serve this client, these various components need to be able to find each other in the context of this served client (to avoid that they get connected to wrong instances of the same service component type).

One possible solution is to allow service components to also expose their endpoints, but then this requires that all such service components need to be publicly visible and addressable, which may not be desirable (e.g., this would result in a huge amount of floating IP addresses that need to be available).

Another solution is to use a sessionID or other identifier, that is shareable and can be passed along as parameter of a service request, and to extend the operation of a zone load balancer. This identifier could be used in at least two possible ways. We will elaborate further on this topic in the final year of the project.

- Either this sessionID is incorporated in the data connection of the request and the FUSION (or service-specific) load balancer is responsible for forwarding the incoming connection to the correct instance. Note that in case this load balancer is service-specific, this functionality could actually be part of the service graph of the composite service and is shared across multiple instances and sessions of that composite service. In such scenario, the choreography component would first program the service specific load balancer and then returns the endpoint of that load balancer to the client.
- Alternatively, a session-specific IP port is enabled on the zone or service load balancer. This way, the load balancer immediately knows what internal endpoint it needs to forward the incoming data connection. These ports could be added or removed dynamically based on the required publicly visible instances or sessions. Consequently, instead of creating a huge amount floating IP addresses, this approach would create a dynamic amount of floating IP ports. Floating IP ports likely should not be immediately reused to avoid collision of old data connections onto unrelated new service instances.

2.7 Service placement

The goal of service placement is to find where/which execution zones (EZs) to deploy service instances to achieve the service provider objectives. In this section, we consider a service as an atomic service. In general, the service providers consider different objectives: some may seek to minimize the network latency between clients and service instances, others may focus on load

balancing across all EZs, while still others may try to minimize the deployment cost of their services. In this work, we define a utility function corresponding to user satisfaction. In detail, we use latency between a user and an EZ as a metric to measure that user satisfaction. We focus on a multi-objective optimization problem in which we first guarantee *max-min fairness* between users and then *maximize* the total utility of all users. We also consider a trade-off between the service deployment cost and the performance (total utility) of users. We model the problem as a linear programming formulation.

In the second part of this section, we present an evaluator-based service placement heuristic, and discuss how this heuristic could be extended towards composite services.

2.7.1 Problem description

For the mathematical model, the problem can be described as follows:

- **Inputs**: estimated user requests, network performance model (e.g. latency between users and EZs), deployment cost of service instances in EZs and resource constraints (e.g. number of session slots that each EZ can support).
- **Assumption**: we use a centralized model in which each Domain Orchestrator (DO) has full input information of all EZs and users. Time is divided into fix-length windows; then the DOs run the optimization formulation at the beginning of each window time. Based on the results, we know which service instances to be deployed in which EZs.
- **Objective**: maximize the performance (total utility) of users while achieving max-min fairness between users. The objective also considers the trade-off between the performance and the service deployment cost.

2.7.2 Mathematical model

2.7.2.1 *Utility function:*

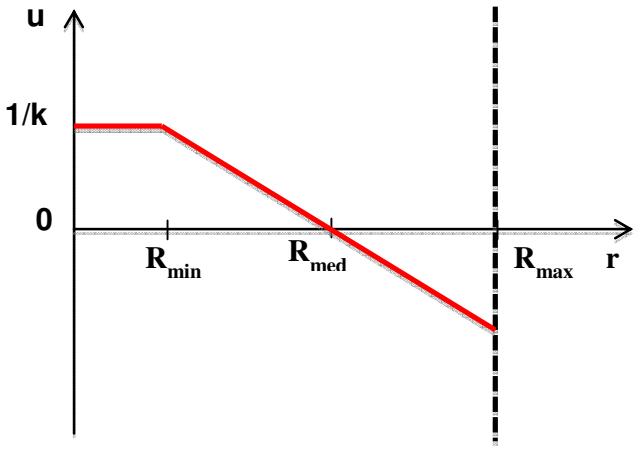


Figure 13. Utility function vs. response time

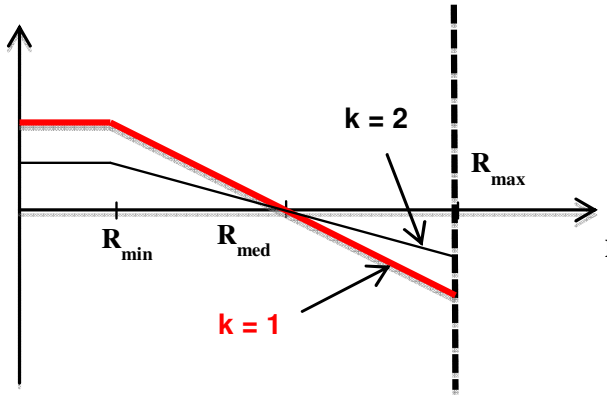


Figure 14. Utility with different "k"

As shown in Figure 13, the utility function should express the following meanings:

- If $r \leq R_{min}$, users are very happy. Depending on service type, we can choose an appropriate value of R_{min}. For some services, even if reducing more the response time cannot improve QoE, therefore the utility keeps the same value if $r \leq R_{min}$. For example, voice over IP requires $R_{min} = 20\ ms$ [Wiki]). However, for other services like Web, the shorter the response time is, the better QoE the users get, so in this case we can set R_{min} = 0.
- If $R_{min} < r \leq R_{med}$: the utility value is positive, meaning that the users are quite happy. However, the user satisfaction is decreasing when the response time is increasing. We call R_{med} as the expected response time.
- If $R_{med} < r \leq R_{max}$: the utility value is negative but the QoE is still in an acceptable range.
- If $R_{max} < r$: the utility value is $-\infty$ meaning that we consider the service request is blocked.

To satisfy the above requirements, we can define a utility function as follows (similar to but more complicated than the one in [NOMS08]):

$$u(z) = \frac{R-z}{kR} \quad (1)$$

where:

$R = R_{med} - R_{min}$ and

$$z = \begin{cases} 0 & \text{if } r \leq R_{min} \\ r - R_{min} & \text{if } R_{min} < r \leq R_{max} \\ -\infty & \text{otherwise.} \end{cases} \quad (2)$$

A constant $k \geq 1$ is used to indicate the importance level of the user request. Large value of *k* means that the request is less important. By changing the values of *k, R_{min}, R_{med} and R_{max},* we can control the shape of the utility function (Figure 14).

2.7.2.2 Optimization Formulation: Linear Programming (LP)

The objective of the optimization formulation is to achieve max-min fairness between users while maximizing the total utility. The objective also considers the trade-off between the performance and the service deployment cost. The algorithm works in two steps:

- **Step 1**: the objective function is to maximize the minimum user utility. In this step, we guarantee that the solution achieves max-min fairness between all users.
- **Step 2**: after the first step, let the value of the objective be $U_{max\text{-}min}$. Then, in the second step, we add the following constraint to the formulation:

$$\min(u) >= U_{max\text{-}min}$$

By adding this constraint, the second step guarantees that the solution we find will be at least as fairness as in the first step. In addition, with the new objective: $max\ [\alpha \sum u - (1-\alpha) cost]$, the solution maximizes the total utility while considering a trade-off with the deployment cost (α is a parameter to say the relationship between the total utility and the deployment cost in the optimization objective).

Inputs:

- Estimated user requests: $\boldsymbol{d_{ij}}$: User "*i*" requests "*d*" session slots from service "*j*".
- $\boldsymbol{b_{ij}}$: link bandwidth required by user "*i*" to get service "*j*".
- $\boldsymbol{l_{it}}$: latency (response time) between user "*i*" and execution zone (EZ) "*t*".
- $\boldsymbol{C_t^j}$: capacity (session slots) of service "j" at EZ "*t*".
- $\boldsymbol{DeployCost_{jt}}$: unit deployment cost of service "*j*" on EZ "*t*".
- $\boldsymbol{BwCost_t}$: unit bandwidth cost at EZ "*t*".
- $\boldsymbol{R_{min}^{ij}, R_{med}^{ij}, R_{max}^{ij}}$: three response time thresholds defined in Figure 13
- $\boldsymbol{R^{ij} = R_{med}^{ij} - R_{min}^{ij}}$

Main variable: $\boldsymbol{x_{it}^j} \in [\mathbf{0}, \mathbf{1}]$ - fraction of user "*i*" connects to EZ "*t*" to get service "*j*". We assume that user "i" is a set of individual users that are grouped by near geography locations.

LP formulation of Step 1:

$$U_{max-min} = max\ U \tag{3}$$

s.t.

$$\sum_{t \in EZ} x_{it}^j = 1\ \ \forall (i,j) \in D \tag{4}$$

$$\sum_{i \in user} d_{ij} x_{it}^j \leq C_t^j\ \ \forall j\ \in service, t \in EZ \tag{5}$$

$$r_{ij} = \sum_{t \in EZ} l_{it} x_{it}^j\ \ \forall (i,j) \in D \tag{6}$$

$$z_{ij} \geq 0\ \forall (i,j) \in D \tag{7}$$

$$z_{ij} \geq r_{ij} - R_{min}^{ij}\ \forall (i,j) \in D \tag{8}$$

$$u_{ij} = \frac{R^{ij} - z_{ij}}{k_{ij} R^{ij}}\ \forall (i,j) \in D \tag{9}$$

$$U \leq u_{ij}\ \forall (i,j) \in D \tag{10}$$

$$cost_{deploy} = \sum_{t \in EZ} \sum_{ij \in D} DeployCost_{jt}\ d_{ij} x_{it}^j \tag{11}$$

$$cost_{bw} = \sum_{t \in EZ} \sum_{ij \in D} BwCost_t\ b_{ij} x_{it}^j \tag{12}$$

$$cost_{deploy} + cost_{bw} \leq TOTAL\_COST \tag{13}$$

$$x_{it}^j \in [0,1]\ \forall (i,j) \in D, t \in EZ \tag{14}$$

Explanation:

- In the formulation, we use the notations "i" as user id, "j" as service id and "t" as EZ id.
- Objective function (3): guarantee max-min fairness where U = min(utility) (as constraint (10)).
- Constraints (4): all the requests of a user "*i*" for a specific service "*j*" have been served.

- Constraints (5): each EZ has a limited number of session slots dedicated for a service type. This may happen that some special services (e.g. the ones that need GPU processing) can only be deployed at some specific EZs. This constraint is used to make sure the number of session slots of EZ is enough to serve user requests.
- Constraints (6) are used to computed the average latency for the user "i" to get the service "j". Assume that the connection between users and EZs are a full mesh, it means that each user can connect to any EZs. For the inputs of the formulation, we simply remove all pairs of (user, EZ) that have latency which is larger than R_{max}^{ij}. This step guarantees that the latency for any user "i" to connect to any EZ "t" to get service "j" has to be less than R_{max}^{ij}.
- Constraints (7) – (8) ensure that $z_{ij} \geq 0 \; if \; r_{ij} \leq R_{min}^{ij}$, otherwise $z_{ij} \geq r - R_{min}^{ij}$.
- Constraint (9) is used to model the utility function (1). Note that when $r_{ij} \leq R_{min}^{ij}$, *z* can be at any value that is greater or equal to 0. However, thanks for the objective function, we would like to maximize the utility (9). It means that the formulation will choose a minimum value of *z*, or in other word, *z* is set to 0. Similarly, the formulation will choose $z_{ij} = r_{ij} - R_{min}^{ij}$ when $r_{ij} \geq R_{min}^{ij}$. Moreover, as mentioned in the constraints (6), a feasible solution does not allow any response time that is larger than R_{max}^{ij}. In other word, we can say that when $r_{ij} > R_{max}^{ij}$, the utility function is *–infinity*. In summary, the constraints (6) – (9) model exactly the utility function as defined in (1) (or in Figure 13). In the formulation, a user is group of individual users. For simplicity, we consider all group users are the same. If users are with different group sizes, we can multiply the right hand size of (9) with a parameter corresponding to the size of the group user "i". This helps to distinguish the group of users with different number of individual users.
- As shown in Figure 14, in the negative region, with the same value of response time r_{ij}, the request with low "k" gives lower utility (we call it a bad request). As the objective function is to maximize the utility of the bad request, the formulation will try to increase the bad request's utility by setting small value of r_{ij} for it. This intuitively means that the request with low "k" is more important.
- Depending on the users and the service type, we can set the appropriate values of the important level "k". Then, by playing with "k", R_{min}, R_{med} and R_{max}, we can control the shape of the utility function (Figure 14).
- Constraint (11) measures the deployment cost of services in EZs.
- Constraint (12) measures the required bandwidth cost of services in EZs.
- Constraint (13) set a limit for the total budget of the service provider.

LP formulation of Step 2:

$$max[\alpha(\sum_{ij \in D} u_{ij}) - (1-\alpha)(cost_{deploy} + cost_{bw})] \quad (15)$$

s.t.

(4) – (14)

$$U \geq U_{max-min} \quad (16)$$

Explanation:

In this step, we add the constraint (16), where U is the minimum utility of users and $U_{max\text{-}min}$ is the objective value from step 1, to ensure that the solution should be at least as fairness as the max-min

fairness in step 1. We keep the same constraints (4) – (14) and change the objective function as (15). α is a parameter to say the relation between the total utility and the cost. If we set $\alpha = 1$, we get the solution with maximal total utility (while the cost is only restricted by the constraints (13)). On the other hand, if $\alpha = 0$ the solution we get achieves max-min fairness while minimizing the total cost.

There is always a trade-off between the utility and the cost. In case the service provider know how to choose the value of α, they simply run the formulation as step 2 above and get the optimal solution. However, if it is hard to estimate α, we have another way to give hint for the service provider to find a suitable solution. In general, given a solution, we can plot its cost and utility on a 2-D plane (Figure 15).

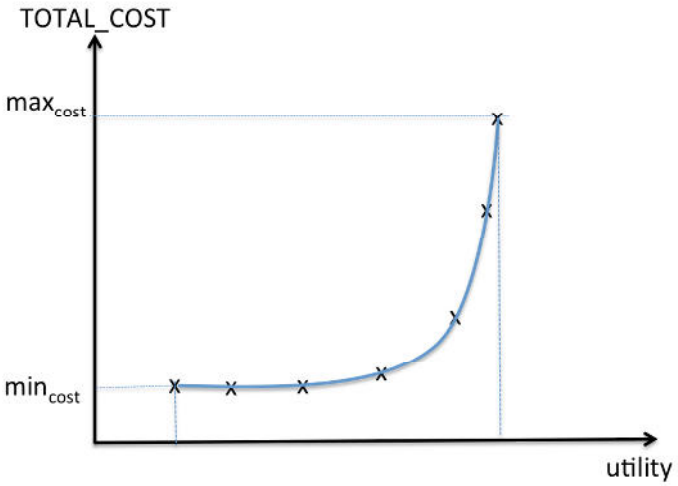


Figure 15: trade-off utility and cost.

By setting $\alpha = 1$ and $\alpha = 0$, we can find two solutions called $\max_{utility}$ and $\min_{cost}$, respectively. From $\max_{utility}$, we can get the corresponding cost called $\max_{cost}$. Then, we set many STEP_COST points in between the range [$\min_{cost}$, $\max_{cost}$]. Next step, in constraint (13), we set the value of TOTAL_COST to be equal to each STEP_COST point. We set $\alpha = 1$ and find the corresponding total utility for each value of the STEP_COST point. Depending on the granularity of the graph and how much time we can pay for computation, we can choose a suitable number of STEP_COST points. Finally, we get a trade-off relationship of the cost and the utility as in Figure 15. Based on this figure, the service provider can easily choose a solution with their desired trade-off.

It is noted that the optimization formulation above is a linear programming model; hence it can be solved efficiently. The number of variables x_{it}^{j} in the LP problem is $|I| \times |T| \times |J|$ where $|I|$ is the number of users, $|T|$ is the number of EZs and $|J|$ is the number of service types. Since $|T|$ and $|J|$ are usually much smaller than $|I|$, the worst case complexity of the LP problem will be $O(|I|^{3.5})$ [LPWIKI].

2.7.3 Relationship with service selection

The objective of service placement is to find which service instances to be deployed in which EZs. Given this deployment, we know the actual session slots at each EZ. The server selection phase is based on these real resource constraints to allocate user requests to appropriate EZs. In general, the formulations that we used in the service placement and the service selection are quite similar. We list in below the commons and the differences between them.

2.7.3.1 Similarities between service placement and service selection

- They share the same idea of utility function.

- The algorithm works in two steps to achieve max-min fairness between users while maximizing the total utility. The algorithm also considers the trade-off between the performance and the cost.
- The constraints in the optimization formulation are similar, except the ones relating to the cost (constraints (11-13) in Section 2.7.2).

2.7.3.2 Differences between service placement and service selection

- The service placement considers the deployment cost of service instances in EZs while the service selection cares about the data transit cost between multi-domains.
- The capacity constraints (5) in service deployment are about the capacity of hardware resource at EZ. In service selection, these are the constraints on the available session slots of service instances in EZs. In general, the capacity of the service selection is less or equal to the one of the service deployment.

The service placement is executed less frequently than service selection. In the service placement, we assume that a user can connect to any EZs. However, in the service selection, to reduce the input size for the optimization, we can assume that each user can only connect to a subset of EZs.

2.7.4 Evaluator-based service placement strategy

Next to the central mathematical placement algorithms presented above, in this section we present an evaluator-service based distributed service placement heuristic for finding an cost-optimal execution environment for a particular service. This strategy allows for service providers to decide on where to deploy services, trading off cost versus QoS.

2.7.4.1 Considerations

The decision where to place a service is based on different criteria, taking account hardware requirements, its proximity to external sources or the clients, but also costs for running the service on a specific execution zone and zone policies. The service provider will be charged for running the service, so in the end the service provider has to decide what he is willing to pay for. For example, is the service provider willing to pay for the more expensive GPU-enabled environments, or for the more expensive small data centres very close to the end users (see also [VERM14]).

The most flexible way is to keep this decision completely up to the service provider: he has to find the balance between costs and usage that optimizes his gain. If the service requires best hardware and maximum performance and the service provider is willing to pay for it, this should be a valid choice.

Therefore, there should be as much freedom to choose from different possibilities for the service provider as possible, because in the end he will be charged for the service. So if there are decisions to be made that may influence the service, it should be up to the service provider to make the decision.

On the other hand the service provider would always use the best solution available if it has no impact on his costs. But any choice he makes may have an economic outcome, be it higher usage of hardware or reduced performance of other services. It is necessary to avoid that every service is becoming greedy and only takes into account its own advantage. This is normally achieved by policies of the zone manager or similar regulations. These policies guarantee that each services gets its share of hardware usage.

Instead of just applying the zone policies with the service provider having no chance to react on a certain decision, another possibility would be to charge less for services that behave according to the policies and to charge more for services that claim more CPU, better network connection or any other resources. In this case the service provider could decide, whether the increased usage of resources is worth the increased costs of if another behaviour would be more economic.

The service provider influences the execution zones where a service will or will not be executed by providing an evaluator service (see Section 2.6.2.2). This evaluator is provided upon service registration and will be run on the execution zones where the service may be executed. It may perform service specific measurements and calculate a rating for the execution zone. The execution zone with the best rating will then be chosen by the orchestration for deploying the service.

This rating has also to take into account the price for execution the service in that specific zone (which is deployed in a specific location in the network). A good rating for a zone based on only the available hardware may therefore turn into a bad rating if the price is not reasonable for the service provider.

The price is therefore a lever to influence the decision of the service provider represented by his evaluator service. If a participant in a FUSION domain wants to influence the placement of a service, this can be done by adjusting the price. If a service for example claims resources in a way that contradicts the policy, it may have to pay a much higher price for it.

2.7.4.2 Description

In this strategy, the idea is to do service placement within a domain and within a zone completely based on the concept of evaluator services. As these evaluator services can be very application-specific, this means that this strategy gives complete control over where instances of particular services should be placed, typically taking into account the following information:

- **Service requirements**

  Services can have very specific requirements with respect to particular resources and resource capabilities, but also the required or preferred proximity to a particular external entities (e.g., a central database, CDN nodes or end users), which directly translates into the number of session slots a service can provide towards the end users with good QoS in a particular execution environment.

- **Historical monitoring/runtime data**

  Apart from static information regarding capabilities and requirements, evaluator services could also take into account the runtime behaviour of services during previous deployments in particular environments, and adjust the scoring accordingly.

- **Pricing**

  In the end, the true purpose of the evaluator service is to estimate how many session slots (with good QoS) a particular execution environment can provide at a particular price; in other words, how cost-effective an environment is for a particular service, both in terms of available internal resources as well as the location of the zone in the network.

  As discussed earlier, a domain and execution zone may still want to impose their policies regarding resource utilization. This can be done through proper pricing of a particular zone and execution environment within a zone. By allowing a dynamic pricing model where the price of execution environments can change based on changing policies or changing runtime behaviour, domains and zones can steer the decisions of the evaluator services. It is then up to the evaluator service to decide at what price particular services should run in particular environments. Note that in case a generic domain or zone evaluator service is used, applying particular policies becomes trivial.

2.7.4.3 Functional modelling

In general, an evaluator service is a function that, given a set of input parameters (including the environment, historical data, policies, etc.), returns a value that can be considered as a score or rank,

indicating how suitable that environment is for deploying a number of session slots of a particular service:

$$score = evaluator(Service, InstParams, Env, Cost(Env, t))$$

In this formula, *Service* represents the service requirements, *InstParams* the instantiation and configuration parameters with which that service needs to be deployed (e.g., UHD quality, best effort QoS, etc.), *Env* represents the environment (both static and runtime environment as well as its location in the network), and Cost(Env, t) represents the cost of using such environment for some unit of time. Note that this cost factor not only can be time-dependent, but can depend on other parameters as well, such as the service type, etc.).

The value that is returned is expected to be a numerical value and can be an integer (e.g. representing a rank) or a float (e.g. in the range [0, 1], representing a score), depending on the service evaluator. As this score is used to compare one environment with another, only the relative value is important and not the absolute value or magnitude of the score (i.e., is score S_1 smaller or larger than S_2). One notable exception is to indicate that a particular environment is completely unusable. In case we restrict the numerical values to be strictly positive, the value 0 could be used for that purpose.

From a semantic perspective, the score will often be some measure of the number of session slots per hour per Euro, weighed with the proximity importance of a service deployed in a specific environment. As discussed in [VERM14], small zones that are close to the edge will typically be much more expensive (i.e., limited capacity, higher real estate and maintenance costs, etc.), so it is up to the service provider to trade-off the cost of an environment w.r.t. its relative location and service requirements.

In this model, an important observation is that an evaluator service can also be regarded as an objective function in a multi-dimensional search space, where the goal is to find the argument values or coordinates that minimize that objective function:

$$Env_{Opt}(Service, InstParams) = \underset{e \in \{E\}}{\operatorname{argmax}}(eval(Service, InstParams, e, Cost(e, t)))$$

where E is the set of all possible logical, virtual or physical execution environments in which a new instance of that service could be deployed. The result is the environment that maximizes the evaluator service score. Note that to incorporate various pricing policies, a single virtual or physical environment could be wrapped into multiple logical environments, for example each with their own cost model (and possibly corresponding behaviour). Different search space exploration algorithms could be used for quickly finding an optimal (or close-to-optimal) solution.

Note that by assigning an expiration date to each evaluation score, subsequent evaluations and optimizations could be significantly faster (possibly also covering a larger search space) and with less runtime overhead (as each evaluation may be done by an active evaluator service that is consuming resources).

2.7.4.4 Simple heuristic

A simple top-down heuristic for doing evaluator-service based service placement is the following and is based on the divide-and-conquer algorithm already presented in Deliverable D3.1:

1) First, a domain could preselect or filter a number of execution zones for which an evaluation will be done. This filtering step could be done for example based on (network) monitoring data, bandwidth or latency requirements (e.g., if a service needs to be deployed in some region of a domain, it makes no sense to evaluate zones at the other side of the domain), etc. Note that a zone may be selected multiple times, for example in case a domain wants to evaluate a zone in different contexts (or policies).

2) Next, for each of the selected zones, the domain triggers each execution zone in parallel for making an evaluation (e.g., within a particular amount of time) by returning an offer matching the service deployment request.

3) Each execution zone subsequently preselects or filters a number of (physical, virtual or logical) execution environments onto which the evaluator will be deployed and/or called. The preselection of environments could be done for example based on static service requirements and corresponding environment capabilities (e.g., availability of a required GPU), where particular evaluators are already deployed, different cost models, etc.

   With each environment, a corresponding cost must be associated by the zone manager, on which the evaluator will operate. A domain could also have provided domain-level cost factors for deploying in a particular execution zone (e.g., based on load or network information).

4) Next, a zone manager triggers all selected evaluator services in parallel to make an evaluation and return a score (possibly within a particular amount of time).

5) All evaluator services evaluate the environment based on its capabilities, service requirements, use of historical data, etc. How this is implemented is completely up to the evaluator. An evaluator may do an active evaluation for each incoming request, do continuous evaluations in the background, etc.

6) All scores that are received in time by the zone manager are compared and the best one(s) are selected to be included in the offer. Note that more than one environment and score may be selected in case one environment cannot handle all requested session slots. A zone manager may decide to share this information with the domain orchestrator.

   This offer is stored and time stamped by the zone manager, along with all relevant information to be able to later deploy the services in the correct environments and price setting when deployment requests come in from the domain. This offer is returned to the domain.

7) All offers received in time by the domain orchestrator are compared and the best one(s) are selected for service deployment. Note that more than one zone may be selected for deploying particular amounts of session slots of that service. During service deployment, the domain passes along the corresponding offer ID so that the zone manager can correlate the deployment request with the offer, check the validity and subsequently configure and deploy the instances onto the appropriate environments.

2.7.4.5 Composite services

The evaluator based placement heuristic as described above focused mainly on atomic service deployment. However, in case the (partial) service graph is already known at deployment time, the optimal location of where to deploy each of the service components should be determined. In Deliverable D3.1, we already described two extreme approaches:

- **Exhaustive recursive approach**

  With this approach, the FUSION orchestrator triggers the top-level evaluator service (across multiple selected zones), which will in its turn trigger the evaluator services of dependent service components (interacting with FUSION for the overall coordination). Although this mechanism may work very well for relatively simple service graphs, for nontrivial service graphs, this quickly results in an exponential amount of possibilities that need to be evaluated.

- **Divide-and-conquer approach**

  In this approach, the FUSION orchestrator triggers the evaluator service of each service component independently for assessing the optimal locations for each individual service components. When all scores have been collected by the orchestrator, the latter then decides on

the optimal location of each of the components, taking into account the inter-service affinity or anti-affinity, and communication bandwidth and latency requirements that are statically described in the manifest of the composite service. Although this approach linearizes the overall placement complexity, a key disadvantage is that this does not include the service provider in selecting the optimal relative location of each component. For services with relatively simple inter-service communication requirements, this could however be more than sufficient.

In this section, we shortly present two variants of both extreme approaches, making the former more efficient (but less accurate), and making the latter more flexible (but slower).

- **Pruned recursive approach**

  Instead of trying all possible combinations, the overall idea is to apply various conservative or aggressive pruning techniques to significantly reduce the number of combinations, using specific application knowledge of the service, service component and inter-service component communication requirements. For example, the total amount of combinations could be kept constant, and depending on the importance or sensitivity of particular requirements, some portions of the service graph could be evaluated more thoroughly than others. Also, by leveraging historical data from previous service evaluations, many non-optimal combinations could already be pruned a priori.

- **Two-phase divide-and-conquer approach**

  With this extension, the divide-and-conquer approach is actually performed twice in two subsequent phases. In the first phase, each the list of scores (or offers) for each service component is determined individually. Once collected, this information is passed in the second phase to an evaluator service of the composite service that returns a final global score for the composition, including the selected offers for each individual component.

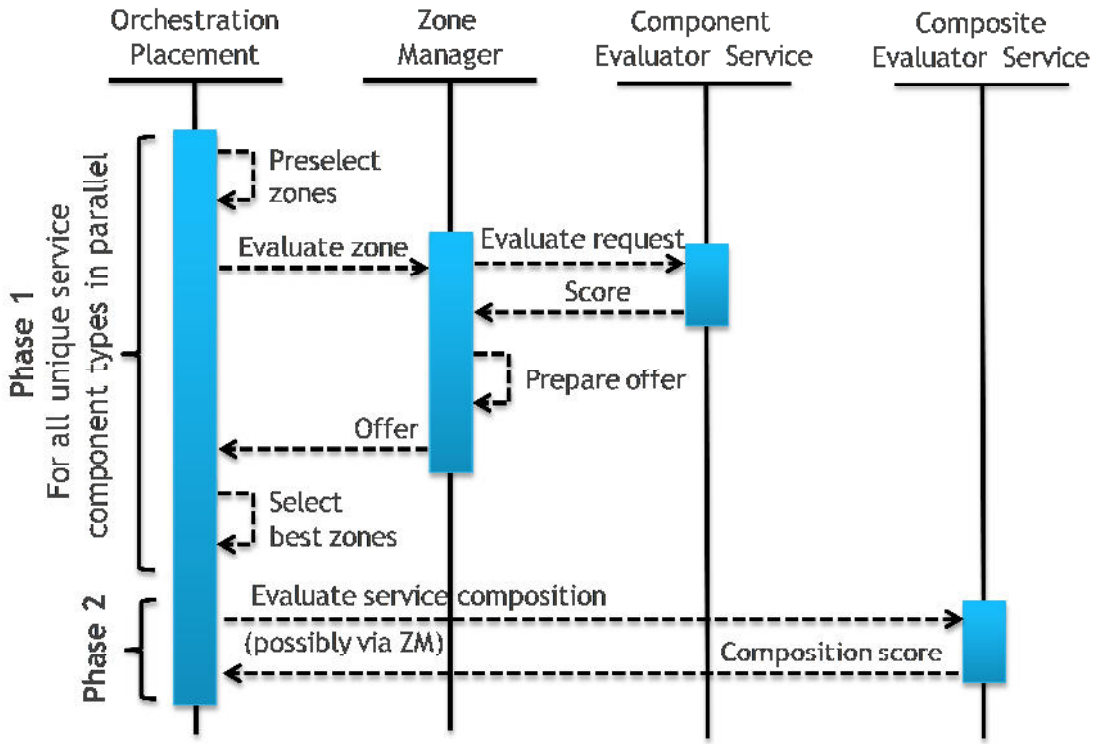


Figure 16 – Two-phase evaluator-service based composite service placement

As input for the evaluator service of the composite service, the previously determined scores (or offers) of the components should be taken into account, possibly including available session slot information of already deployed components. The latter is needed to be able to also leverage

already deployed instances of particular service components. The composite evaluator service may request particular information regarding the cost and QoS in between particular zones, in case the evaluator service itself has no available information (e.g., based on historical data from previous deployments). The overall approach is depicted in Figure 16.

In the current approach, the location of the composite evaluator service can be arbitrary. Alternatively, it could also be deployed strategically along with the possible location(s) of the most critical component in the graph (to be able to do actual measurements).

In the final year of the project, we will evaluate these heuristics as part of the integrated prototypes for a number of composite service graphs.

2.7.4.6 Implementation

We already implemented an initial version of this heuristic in the integrated end-to-end demonstrator, which we are currently evaluating in WP5 in a number of scenarios. In the prototype, all steps described earlier are done for services that expressed a dependency on an evaluator service. As a proof of concept, we also implemented an initial simple evaluator service to verify the corresponding APIs and overall concept. In the final year of the project, we will quantitatively evaluate the effectiveness of evaluator services in the integrated prototype in a heterogeneous environment (see Deliverable D5.2).

2.8 Service scaling

2.8.1 Session slots

FUSION targets long-lived sessions of multimedia applications running on heterogeneous clouds. The maximum number of concurrent sessions a service instance can handle depends on various parameters, such as the number of resources assigned to the execution environment (container, VM, etc.) and the capabilities of the underlying hardware (e.g. GPU capabilities). These cannot be predicted by the application developer.

Introduced in Deliverable D3.1, we created the concept of session slots to address this heterogeneity in execution zones and deployment policies. Service instances report the number of new connections they can serve with sufficient QoS[1] to their Zone Manager, who in turn will advertise service load information to its gateway service resolver. This advertised number may be the mere sum of all available session slots reported by the instances, but it could as well hide zone scaling policies. For example, if each instance is expected to report *N* session slots, but currently only one instance is actively running, the zone might decide to report *M*x*N* session slots, with *M* the maximum number of instances that can be deployed in this zone (e.g. taking into account cost considerations). This way, the zone-specific scaling policy remains hidden, and new requests can be routed to the zone.

2.8.2 Slot-based zone scaling

Instance load information is of vital importance to the service resolution layer; but they can be an important driver for implementing scaling policies inside a single zone. Scaling policies are defined by the orchestrator when the service is registered in the zone.

When deploying an instance, the orchestrator specifies the number of session slots that should be available at all times. If the number of available session slots, aggregated over all running instances, drops below a threshold predefined in the service manifest, the Zone Manager instructs the DCA layer to deploy additional instances.

[1] The notion of session slots may not be suitable for all service types, e.g. services with a request queue. Such services may report a different indication of load, e.g. the average waiting time in their request queue. This will be studied later in the project.

When service demand decreases, instances may be shut down to reduce infrastructure renting costs. Many scaling policies are already available in cloud software like OpenStack; e.g. based on average CPU usage across all VMs of the same service in the evaluation period. However, default scaling down policies shut down the instance with the shortest lifetime. For FUSION services with long-lived sessions, this solution is unacceptable as it would result in users being disconnected abruptly.

Instead, the Zone Manager must interface with the DCA to

- Identify the specific instance to be stopped, e.g. the instance that currently has the most available session slots. We refer to such instances as *decommissioned*.
- Ensure that new clients are no longer redirected to the decommissioned instance, e.g. by reconfiguring the service's load balancer.
- Shut down the decommissioned instance once the last client has disconnected. This can be monitored if the number of available session slots equals the maximum number of session slots that an instance can handle.

2.8.3 Implementation

To showcase the feasibility of this approach, we have opted for a tight integration with OpenStack, the leading open source cloud computing solution. The relevant components are shown in Figure 17.

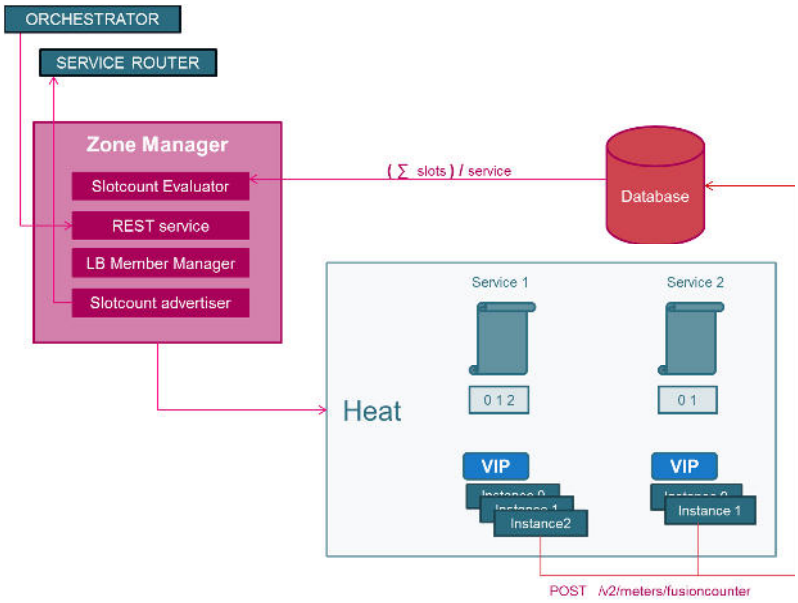


Figure 17 – Implementation of instance scaling in a single zone; triggered by session slots

2.8.3.1 Collection of monitoring information

Deployed service instances use the FUSION Zone Manager API to post their currently available session slots. The API is implemented following the REST principles, with instances sending updates via POST messages to Ceilometer, which is the metering framework provided with Openstack. We have implemented a new meter in Ceilometer, namely `fusioncounter`[2]. Instances will then provide updates on their session slots to this meter; of which they receive the IP address and port by reading from the configuration variable, stored in a key value store.

[2] The default meters available in Ceilometer can be found at:
http://docs.openstack.org/developer/ceilometer/measurements.html

The body of the POST message to Ceilometer follows the following JSON syntax:

```
[ { "counter_name": "fusioncounter",
     "user_id": "",
     "resource_id": "",
     "counter_unit": "sessionslots",
     "counter_volume": "$sessionslots",
     "project_id": null,
      "counter_type": "gauge",
      "resource_metadata": {
                  “instanceId" : "$iid",
                   "serviceId”:“$sid”
      }
 }]"
```

The number of available session slots is encoded in the variable `counter_volume`, following the Ceilometer policy.

2.8.3.2 Scaling

The Zone Manager contains four components that are involved in the zone-internal scaling process.

- **REST webserver**: this webserver implements the Zone Manager API, and is used by the Orchestrator to register and deploy services.
- **Slotcount Evaluator**: this module continuously evaluates the aggregated number of available session slots of all running instances and starts up- or downscaling if needed.
- **LB Member Manager**: this bookkeeping module keeps track of which instances are behind the same load balancer. It is the IP and port of the load balancer that are announced as service endpoint to the service resolution plane.
- **Slotcount Advertiser**: this module is responsible for injecting service load information into the service resolution plane. In the current implementation, this is the sum of all available session slots of running instances, plus the number of session slots that could be offered in addition by deploying the maximum number of instances allowed.

When the number of service instances must be scaled down, the Zone Manager will flag the instance that currently has the most available session slots to be decommissioned. The load balancer will be configured not to forward any new clients to this instance. The Zone Manager will notice that the last client has disconnected from the decommissioned instance when the number of available session slots reported by this instance equals the maximum number of session slots. Then, the Zone Manager will instruct HEAT, the built-in orchestrator of OpenStack, to stop this instance.

The current HEAT implementation was extended to support session-slot based scaling. Scaling operations in HEAT are carried out by evaluators that are triggered by alarms. However, the current HEAT implementation of the scaling evaluator simply stops the instance with the shortest lifetime. Therefore, we have made the HEAT scaling evaluator instance-aware. The evaluator reads from a file provided by the Zone Manager which instances should be running. The Zone Manager may add or remove instance IDs to this file, and triggers the HEAT alarm accordingly.

2.9 Intra-zone load balancing

There are good reasons for a FUSION Execution Zone not to expose the endpoints of all individual service instances to the service resolution plane. Some of these reasons are as follows:

- **Scalability**

  Reducing the number of items that need to be injected and managed by the service resolution plane, by aggregating the session slot availability information per service type rather than per instance;

- **Internal zone or DC policies**

  A FUSION zone manager may want to have better control on the balancing of incoming requests across different instances. Reasons for this include congestion avoidance or mitigation, terminating particular physical racks or hosts, etc.

- **On-demand deployment**

  A zone manager may want to be very aggressive in terms of the number of running instances compared to the number of reported available session slots. For example, it may want to expose more slots than what is supported by the already deployed instances (if any). Especially in case of fast lightweight deployment mechanisms such as containers, the penalty of deploying a new instances on-the-spot may be almost negligible in some cases. In such scenarios, not having to expose individual instances is an advantage.

- **Floating IP real estate**

  A zone manager may want to minimize the number of floating IPv4 addresses that he has to use for making all services directly accessible. With a load balancer, only the IP address of the load balancer needs to be publicly addressable.

- **Security**

  For security reasons, it is also interesting not to expose the endpoints of all individual instances. Specific attacks can be better mitigated by a load balancer.

As FUSION services already explicitly expose session slot availability, it makes sense to take this metric into account when doing load balancing. The session slot information can be used in combination with internal zone or DC policies for deciding to what specific instance a request needs to be forwarded.

This FUSION load balancer could be implemented in a number of ways. For example, existing applications or flow-based load balancers such as haproxy or OVS could be extended (e.g., by adding a specific plug-in) to support this session-slot based load balancing.

As an example, in Figure 18, we present a possible FUSION-aware flow-based load balancer using OVS in an execution zone. When a new incoming data connection arrives in the execution zone for the Streamer service[3], the OVS switch will percolate to the FUSION-aware controller, as this is a new flow for which it has no information yet. At this point, the FUSION-aware flow controller can select an appropriate instance of the Streamer service for handling that new connection/session. The OVS switch subsequently is configured accordingly to automatically forward all subsequent data packets of that new connection to the private IP address of the selected FUSION instance (e.g., $Streamer_0$). For other incoming requests, the same procedure would be repeated, allowing to balance different connections across different instances.

[3] It is important to note that we assume here that the service resolution step already has been done. As such, the “Streamer” label in the Figure actually represents the public endpoint for that service in that zone (e.g., the endpoint of the load balancer handling that service).

Also within an execution zone, this flow-based load balancing can be done. For example, in case the $Streamer_0$ instance wants to open a connection to an EPG service (running inside the zone), the outgoing connection will be intercepted by the local OVS switch, allowing the FUSION-aware controller to select the optimal instance of EPG and configure the OVS switch to forward all packets for that flow to the private IP address of the selected EPG instance.

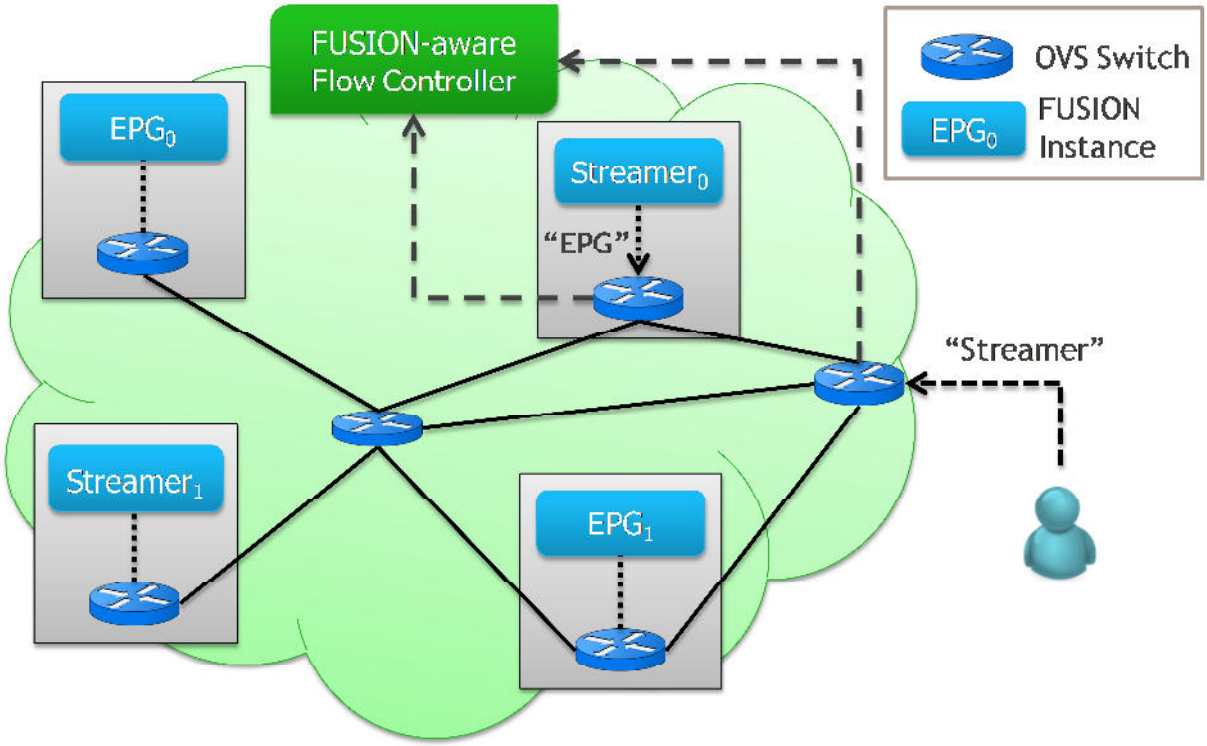


Figure 18 – Example OVS flow-based FUSION-aware load balancing

3. DESIGN & IMPLEMENTATION

3.1 High-level architecture and design considerations

In Figure 19, a high-level overview of the key FUSION components and their key interactions are depicted:

- **A FUSION domain orchestrator** (D), managing all registered FUSION services across a set of distributed execution zones.
- **A FUSION execution zone** (Z), managing deployed FUSION services in a execution environment.
- **A FUSION data centre adaptor layer** (DCA), an optional adaptor layer, abstracting the details of the lower-level orchestration and management layer of the underlying execution environment.
- **A FUSION service resolver** (R), providing optimal service selection for client service requests.
- **A FUSION evaluator service** (EVAL), providing application-specific feedback for optimal service placement.
- **A FUSION application service** (EPG), leveraging FUSION architecture for providing excellent QoE towards clients.
- **A FUSION service provider** (blue avatar), registering and monitoring FUSION application services.
- **A FUSION client** (green avatar), connecting to FUSION application services.

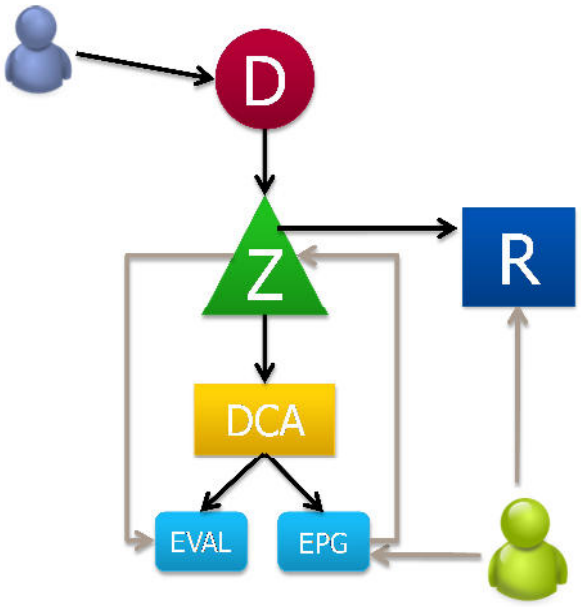


Figure 19 – High-level view of all key FUSION components

This architecture has been discussed in more detail in Deliverable D2.2. In this chapter, we will focus on the design of the individual components, their primary interfaces and their implementation status. In this section, we shortly elaborate on some of the cross-layer design considerations.

A first design consideration is the usage of a adaptor layer in between a FUSION execution zone, managed by a zone manager (ZM), and the underlying data centre. In fact, for implementing an execution zone and zone manager, we envision a number of approaches or modes, as depicted in Figure 20:

- All-in-one, where the data centre also incorporates all ZM APIs and functionality.

- A fat DC-specific ZM, where the zone manager also includes the DC-specific DCA backend.
- A thin DC-independent ZM, where the zone manager only includes the higher-layer FUSION functions, but uses the public DCA APIs for all lower-layer DC lifecycle and resource management functions. Note that in this mode, the FUSION DCA component could either run on top of the DC as an opaque service, or could also be integrated into the DC (API).

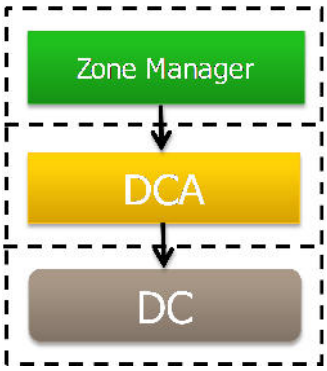


Figure 20 – Layered execution zone design

The advantages and disadvantages of each of the modes are summarized in Table 2. In this section, we have chosen to focus on the thin ZM mode, thus assuming a clean separation between the higher-layer FUSION lifecycle management functions in the ZM and the lower-layer DC-specific lifecycle management functions implemented in the DCA and DC layers. As such, we will also describe the public interface between the ZM and the DCA in following sections. Obviously, blending one or more of these components allow for better integrated and optimized solutions.

Table 2 – Advantages and disadvantages of various ZM/DCA/DC integrations

Mode	Advantages	Disadvantages
All-in-one	• Highest integration & optimization opportunities • Direct control of hardware • Lowest overhead	• Requires control over entire stack: from DC all the way to ZM • May require different implementations for different DCs
Fat ZM	• Reasonable integration & optimization opportunities • Can be deployed on DCs managed by other providers, allowing to elastically leverage external resources	• May not have direct control over hardware • Different DCs may require different fat ZM implementations, or fat ZM needs to extra-fat and contain back-ends for multiple DCs
Thin ZM	• Clean separation of components • Only one lightweight ZM implementation required	• No cross-layer integration & opportunities, especially when DCA and DC are also separated

In Figure 21, the various deployment combinations are depicted of data centres, DCAs, zone managers and domain orchestrators. Option 1 represents the simple case where one ZM is deployed on one DCA on top of one DC. In general, we envision that both data centres as well as DCAs could host multiple execution zones, possibly from multiple independent FUSION domains. For example, one could easily deploy multiple execution zones from one or more FUSION domains on public cloud infrastructures such as Amazon EC2 (see options 2 and 3). The other options represent some of the

other cases mentioned above, such as the thin ZM where the DCA is integrated in the DC (option 4), one or more fat ZMs deployed on top of a DC (option 5), and finally an all-in-one ZM (option 6).

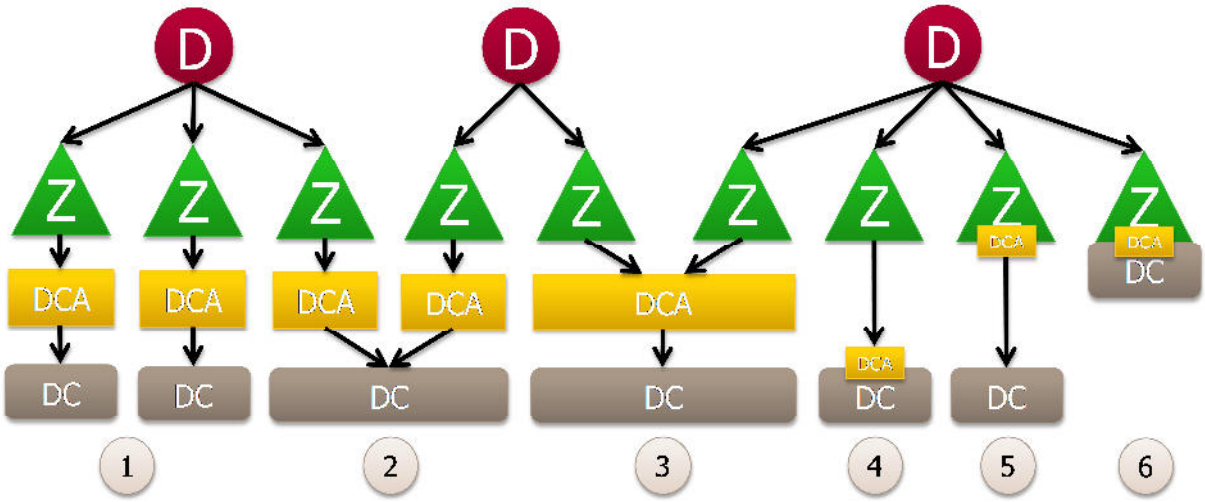


Figure 21 – Different deployment combinations of ZMs, DCAs and Data Centres

A final consideration is regarding the prototype implementation of each of these components to validate and evaluate their functionality and interworking. We started by decomposing all components into the key functional blocks and defined their corresponding interfaces, as already partially described in Deliverable D3.1. Subsequently, we mapped these high-level interfaces onto a set of REST protocols and clearly defined their scope and purpose. More details of these protocols can be found in Sections 3.6 and 7.

This allowed us to start implementing one or more initial prototypes of each of these functional blocks independently from each other in various programming languages and environments, using the modular design approach as discussed in Section 3.9 of Deliverable D2.1. Different partners can implement their own zone manager, DCA, session slot based scaling mechanism or domain orchestrator placement module according to their requirements, and integrate it easily in the overall prototype.

In the following sections, we elaborate on each of the layers, providing an update on their design followed by a description of their prototype implementation.

3.2 FUSION domain orchestrator

3.2.1 Design

An updated version of a FUSION domain orchestrator design is depicted in Figure 22. In the graph, we depict all key elements and design decisions:

- **A common public interface**, shared by all external entities that need to interact with a domain orchestrator. Using proper user/role based authentication and authorization mechanisms, particular registered users acting in a particular role can query and/or modify particular aspects in a domain. For example, a zone manager or service provider can only see the subset of services to which it has access to. Similarly, only a zone manager can push session slot monitoring information regarding owned services.
- **A modular decomposition** of each of the key domain functions into independent internal software components, each of which having its own internal standardized (REST) interface. This allows to efficiently upgrade or add new functionality or features in existing or new modules without impacting the other modules.

- **State and configuration data** is maintained in a highly available distributed key value store such as ETCD [ETCD01], facilitating the implementation and management of a distributed modular domain orchestrator. All configuration and service state data are automatically replicated and synchronized across multiple peers, allowing seemingly local access to the stored data.
- **Scalability and high availability** of each of the functional components can be implemented, by replicating across multiple (potentially distributed) instances, leveraging the replicated key value store for keeping all instances in sync with each other. For implementing high availability, the various replicas of the functional components of the domain orchestrator could be deployed across multiple execution zones that are registered in that domain.

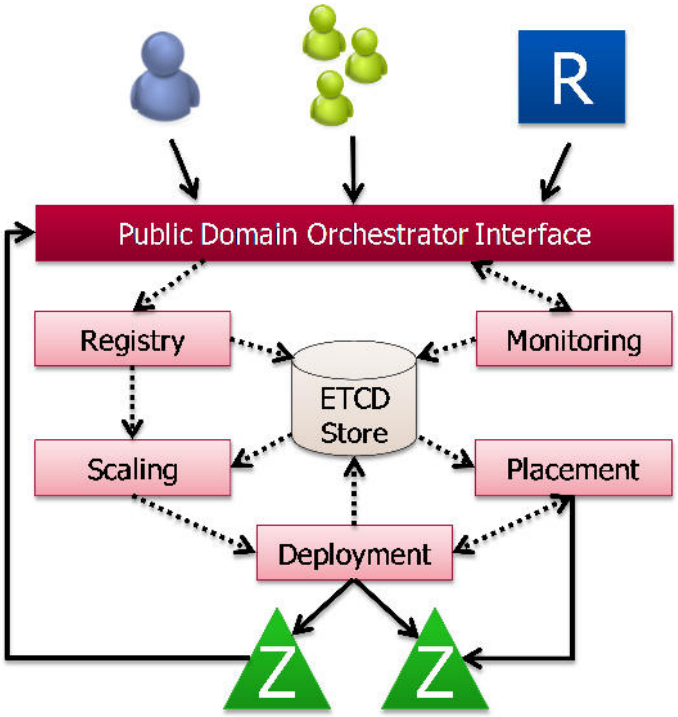


Figure 22 – Updated high-level design of a FUSION domain orchestrator

We now briefly discuss each of the functional components depicted in Figure 22. The corresponding concepts and algorithms have been discussed already in detail in Section 2. Evaluation of these algorithms and concepts can be found in Section 4 as well as in Deliverable D5.2.

- **Registry**

  This component is responsible for the registration of new and querying of existing FUSION services. Upon registration, the service information (specified in the TOSCA manifest) will be stored in a data or file store along with all necessary bookkeeping. At the end of service registration, the scaling component may be triggered implicitly or explicitly to start scaling that service across a number of zones, based on the load patterns provided in the manifest. Note that service updates provided by the service provider (e.g., to change some of the operational parameters or provide a (minor) update of the service), will also be handled by this module.

- **Scaling**

  Based on load patterns or an explicit deployment/termination request, this component will appropriately deploy or terminate sessions of a particular service in one or more zones. This component only is responsible for *deciding* whether and when some scaling action needs to be performed at domain level, for which various algorithms can be used. Note that this scaling component could also be used for deciding whether the FUSION domain itself need to be scaled up or down, for example in terms of number of active execution zones in a particular region.

- **Deployment**

  The role of this component is to coordinate the effective deployment or termination of a service in one or more zones (or even of execution zones in particular DCs). This component will first trigger the placement component to decide in what zone(s) a particular number of session slots need to be created, added or removed. Afterwards, this component will communicate with the necessary execution zones to coordinate the deployment actions in each zone.

- **Placement**

  This component decides how many session slots need to be created, added or removed in particular execution zones, based on scaling information, the execution zones and networking information as well as the application-specific requirements as expressed by the evaluator services. For the latter, the placement function communicates with the corresponding selected execution zones to trigger the relevant evaluator services. This may involve first registering the service and its evaluator service, for which it will also contact the zones. Several algorithms and strategies for domain placement have been described in Section 2.7.

- **Monitoring**

  The role of this component is to provide monitoring information regarding the services (e.g., session slots), execution zones, as well as particular networking information, which can be leveraged by several entities, such as the scaling and placement component as well as the service providers or domain admin to query the current runtime state and health of his services or environment, respectively.

3.2.2 Implementation

For the implementation of an initial domain orchestrator prototype, we started with a more simplified implementation in which all functional components are integrated within a single module. The initial goal was to validate the inter-layer domain orchestration protocols by building an initial skeleton implementation.

For this skeleton prototype implementation, we use Python as programming environment and leverage the Flask framework and Flask-Restful module to be able to implement the REST APIs with little overhead. Ongoing work involves breaking down the prototype in smaller functional units and implementing the private REST APIs in between these subcomponents.

A summary of the implementation status is depicted in Table 3.

Table 3 – Implementation status of domain orchestrator prototype

Status	Task	Comments
Done	Initial API skeleton	Setup of initial skeleton implementation with all key public REST APIs in Python/Flask
Done	Docker container	Wrapping the prototype in a Docker container
Done	Initial implementation	Initial implementation of the key domain orchestrator functionalities, such as registering users, zones and services, as well as placing and deploying new services in the zones
Done	Persistent state	Dumping all relevant state in a key value store
Active	Functional decomposition	Decompose the monolithic domain orchestrator into several subcomponents. Initial decomposition

		already implemented for domain placement.
Planned	Automatic scaling	Implement domain scaling based on load patterns.
Planned	TOSCA	Implement support for TOSCA manifests. The current implementation assumes simple JSON manifests.

3.3 FUSION zone manager

3.3.1 Design

In Figure 23, an updated version of a zone manager is depicted. We assume here that the zone manager is a lightweight overlay layer on top of a DC and DCA, where all lower-level and DC-specific deployment details are handled. As discussed earlier, other possible designs include integration of the DCA layer and the DC itself, in which case the design would change and would have to incorporate also more lower-level MANO functionality.

As such, a lightweight zone manager is mainly concentrated around two key high-level FUSION concepts:

- Handling session slots for doing service scaling, load balancing of incoming service requests, and propagation of service availability into the resolution plane;
- Handling evaluator services for evaluating the placement of FUSION application services in particular DC environments (e.g., different DC instance types).

Many other design decisions, such as a common public interface, a modular decomposition of scalable and high-available components with a automatically replicated data store, are similar to those of the domain orchestrator.

Two notable differences are the role of the load balancer, where FUSION clients (i.e., the light-blue avatars) (in)directly connect to when trying to connect to a particular service, and the zone gateway, for injecting session slot information into the FUSION resolution plane. The load balancer forwards incoming service connections to the best service instance (based on zone policies and slot availability) based on incoming TCP connections instead of REST calls.

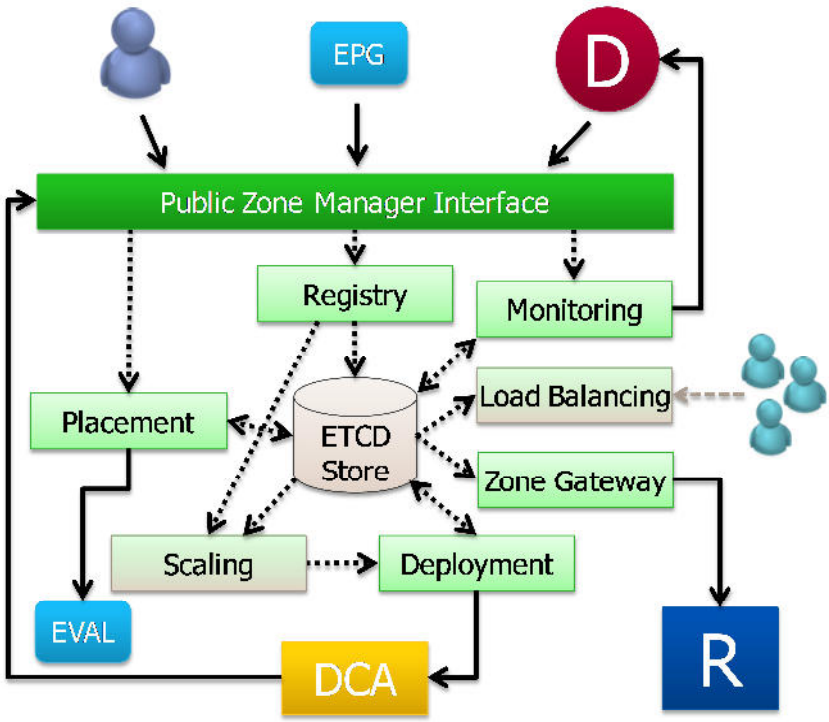


Figure 23 – Updated high-level design of a FUSION zone manager

We now briefly discuss each of the functional components depicted in Figure 23. From a high-level point of view, the design looks slightly more complex than a domain orchestrator. This is because the zone manager is in the centre of all FUSION layers, with the domain orchestrator above, the service resolvers and DCA below, handling the incoming service connection requests and the interaction with the evaluator services. Below an overview of the core functionality of each of the key components.

- **Registry**

  This component handles incoming requests from a domain orchestrator regarding registering and changing the number of session slots of particular services. This may directly or indirectly trigger the scaling component to start deploying or terminating service instances, based on the updated amount of session slots and the internal zone manager policies (e.g., eager versus lazy scaling).

- **Placement**

  The main role of the placement component in a thin ZM model is to manage the incoming evaluation requests from the domain orchestrator and to trigger the corresponding evaluator services for calculating the necessary scores, and for eventually preparing a proper offer towards the domain orchestrator. A key aspect in this role is to be aware of the various execution environments of the underlying DCA, the requirements of the application service, and the internal deployment policies (e.g., a zone may want to reserve its GPU-capable nodes for the most demanding services that may bring the most revenue, which will be reflected in the pricing strategy).

- **Scaling**

  This component on the one hand is responsible for implementing the deployment requests coming from the domain orchestrator and that were negotiated via the evaluation offers. On the other hand, we also envision that a zone manager may have the flexibility to autonomously scale the number of active instances, based on the actual incoming demand, rather than always be

required to have the maximum number of slots available. This allows for better resource consolidation and higher revenue. It is however the responsibility of the zone manager to ensure that enough slots are available when there is a sudden spike of incoming service requests. As such, apart from intelligent scaling algorithms, fast lightweight deployment mechanisms such as containers can help implementing more lazy deployment strategies with minimal penalty. Note that we envision that this Scaling component in the future could also delegate some of its elastic scaling functionality by implementing them as a plug-in or hook onto the underlying cloud orchestration layer. We hence coloured this component partially green and partially grey.

A second function includes deciding whether to dynamically scale the effective size of the execution zone on top of the underlying DC. For example, on a cloud DC, a zone manager may start by enabling only a limited amount of (virtual) environments in which to deploy FUSION service instances, but dynamically scale as more services are being handled by that zone. This elastic scaling of a zone is managed by the zone admin, which can set and update upper and lower bounds.

- **Deployment**

  This component is responsible for deploying or terminating service instances by delegating and coordinating with the underlying DCA layer. A secondary function includes requesting the DCA layer to add or remove environments within the underlying DC, dynamically growing or shrinking the effective size of the execution zone in that DC.

- **Monitoring**

  This component manages (and aggregates) the available resource and service session slots, and updates them in the data store. Changes in session slot availability will typically trigger the scaling and zone gateway components to take some actions. Another role of this component is to collect information on the utilization and efficiency of the various DC environments offered by the DCA.

- **Load Balancing**

  In FUSION, we envision that execution zones can aggregate session slot information from multiple service instances and present them as coming from a single service instance to the service resolution plane. In such case, incoming service requests need to be properly balanced across the various service instances. Although this functionality could be implemented by the services themselves for finer-grained control, we also envision that a zone manager can provide a default implementation for services that do not require service-specific balancing. For this load balancing, the available session slot information can be leveraged for doing intelligent load balancing based on available slots, and take into account execution zone (scaling) policies, instead of random or CPU-load related load balancing mechanisms.

  As with the Scaling component, we envision that this component could be implemented as partially integrated or leveraging the capabilities of the underlying cloud infrastructure.

- **Zone Gateway**

  Last but not least, this component has the crucial role of providing updates regarding service availability by timely injecting session slot updates into the connected service resolution plane, as discussed in significantly more detail in Deliverable D4.2.

3.3.2 Implementation

For the implementation of the zone manager prototype, we used a similar approach as for the domain orchestrator. The goal of the initial skeleton prototype was to validate the public interfaces. As such, in the initial prototype, each of the subcomponents are integrated in a single module. We used a similar Python/Flask based environment and framework for efficiently implementing the

skeleton prototype. Current work involves breaking down this monolithic prototype in the various subcomponents as depicted in Figure 23 and implement their corresponding private interfaces. A summary of the implementation status is depicted in Table 4 below.

Table 4 – Implementation status of zone manager prototype

Status	Task	Comments
Done	Initial API skeleton	Setup of initial skeleton implementation with all key public REST APIs in Python/Flask
Done	Docker container	Wrapping the prototype in a Docker container
Done	Initial implementation	Initial implementation of several key zone manager functionalities, such as handling high-level service state management, managing service instances, evaluator services, and session slots
Done	Multi-config. instances	Adding support for sharing service component instances across multiple services
Active	Persistent state	Dumping all relevant state in a data store
Active	Functional decomposition	Decompose monolithic zone manager into several subcomponents
Planned	Automatic scaling	Integrating the scaling mechanism of Section 2.8 into the main prototype
Planned	Placement	Efficiently dealing with heterogeneous environments for doing (evaluator-based) placement
Planned	Zone Gateway	Integrating with the zone gateway / service resolver implementation of WP4

3.4 FUSION DC adaptor

3.4.1 Design

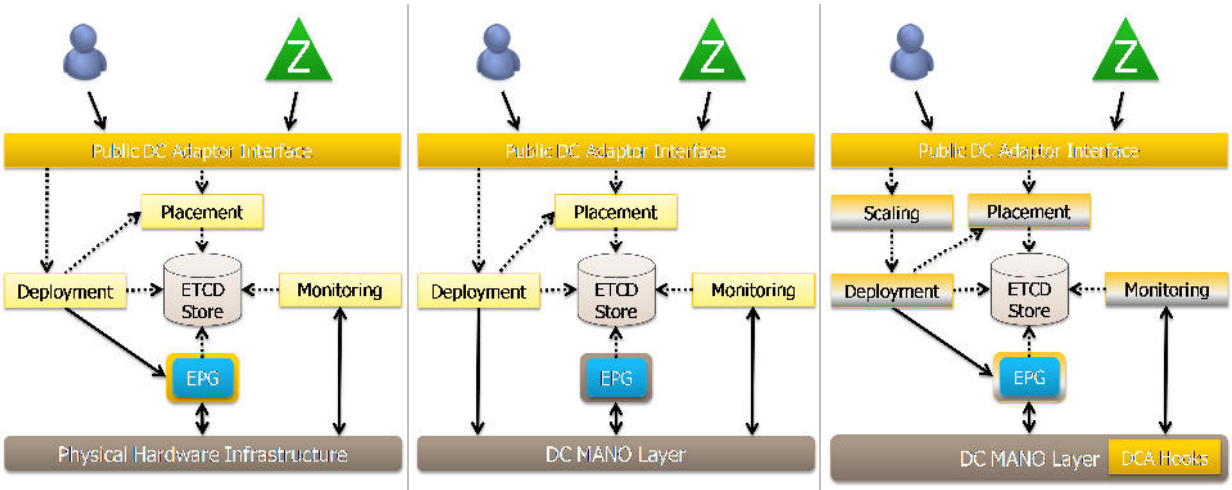
The goal of the DC adaptor layer (DCA or DCAL) is to be the glue between the high-level FUSION zone manager and the underlying DC and its corresponding DC management and orchestration (MANO) layer. As such, depending on the characteristics of the DC and its MANO layer, different DCA implementations may be completely different in nature. We see three main cases:

- *Physical DCA*: in one extreme case, a DCA has direct access to the physical compute nodes and network infrastructure, all of which can be fully configured and customized by the DCA layer.
- *Virtual DCA*: in the other extreme case, a DCA needs to run on an existing cloud environment such as Amazon EC2 or OpenStack in an OTT-like model, meaning it only has access to the set of virtualized environments and capabilities offered by the DC MANO layer.
- *Hybrid DCA*: in between, one could imagine the case where the FUSION DCA functionality or capabilities are (partially) implemented in the existing DC MANO layer or has hooks into the existing MANO layer for better QoS management of the demanding FUSION applications on the existing DC, by exposing particular enabling APIs and capabilities.

To be able to accommodate all three cases and to avoid overlap in implementation of the FUSION zone manager and the underlying DC MANO layer, one of the design goals of the DCA layer we set, was to put the northbound interface towards the zone manager as high as possible, meaning that a zone manager should only be dealing with the FUSION-specific functionality and not have to deal with the lower-level details regarding managing the physical resources or mapping FUSION instances onto the underlying physical or virtual infrastructure.

As a result, the thickness and complexity of the DCA layer may vary significantly depending on the three cases mentioned above. In the physical DCA case, the DCA layer needs to implement all these lower-level resource management functionality (possibly using existing infrastructures such as OpenStack, CoreOS, etc.), but has the advantage of having direct access to the underlying physical resources. In the virtual DCA case, the role of the DCA layer is mainly to translate FUSION deployment requests onto the underlying DC MANO APIs, resulting in a rather thin DCA layer, at the expense of possibly limited control of the underlying resources and QoS. For services with very stringent requirements, this may be difficult or result in elevated runtime costs, which eventually result in elevated service deployment costs; for less demanding services or DCs with proper QoS management capabilities, this may be more than sufficient. The hybrid case combines the best of both worlds but requires proper agreements and support from the DC provider.

In Figure 24, three example DCA designs are depicted, one for each case. In the following sections, we describe these designs for each of the three cases in a bit more detail. Obviously, multiple designs are possible depending on the characteristics of the underlying DC infrastructure and platform.



(a) Physical DCA (b) Virtual DCA (c) Hybrid DCA

Figure 24 – Example DCA designs

As discussed in Section 2.5.3.3, a key role of the DCA layer with respect to monitoring is also to abstract the underlying infrastructure and platform capabilities in a number of (possibly annotated) abstract *execution environment types*. Each environment represents a particular set of physical or virtual resources (a classical Xeon-based rack server, a micro-server, an EC2 c3.xlarge instance type, etc.) that are configured by the platform in a certain manner (e.g., a NUMA-aware environment, a RT-guaranteed environment, etc., see also Sections 3.5 and 4.3).

For particular DCs and DCA layers, the number of environment types can easily become quite significant, in which case it is up to the zone manager to efficiently deal with these types of environments, for example by smartly grouping or clustering different environment types based on the annotated data or evaluation scores for particular services, and by smartly deploying a number of evaluator services across a number of these environment types.

3.4.1.1 Physical DCA design example

Figure 24 (a) depicts a possible design for a physical DCA, where the DCA has direct control over the physical infrastructure. Key functional components there include the following:

- **Monitoring**

  This component collects both static information on the available hardware resources, as well as runtime information on how these hardware resources are being used. This is mainly used for internal purposes. The DCA layer automatically extracts a number of environment types out of this information, which is presented as such towards the zone manager.

- **Placement**

  This component is used for two purposes. First, the zone manager will use this to determine the optimal type of environment a particular service instance should be deployed, leveraging the evaluator services for that purpose. Second, the deployment component uses this component for determining where to deploy a new FUSION instance (given a particular environment type) and how that environment need to be configured appropriately.

- **Deployment**

  This component is responsible for preparing both the chosen environment (i.e., configuring both the hardware resources as well as underlying platform), as well as preparing the service instance to be deployed for actual deployment. This includes appropriately encapsulating the service instance, doing all provisioning, preparing the ETCD store with all service configuration and instantiation parameters, and booting the instance. When the service is terminated, the instance and its environment need to be cleaned up appropriately so that it can be reused for other instance deployments.

3.4.1.2 Virtual DCA design example

Figure 24 (b) depicts a possible design of a virtual DCA (e.g., a DCA deployed on top of a full OpenStack environment), where the FUSION DCA service provides a FUSION-enabled PaaS layer on top of the existing DC platform. In this case, the DCA typically only has control over a limited set of virtualized environments (e.g., different flavours in OpenStack or instance types in Amazon EC2). As such, the main task of the DCA components is about mapping FUSION requests onto the DC-specific APIs and vice versa, as well as optimally mapping/wrapping FUSION services on top of the virtualized environments. For example, if the underlying DC only supports VMs and the FUSION services are containers, the DCA needs to create appropriate VMs in which it can efficiently deploy one or more FUSION service containers. The key functional blocks and their functionality include:

- **Monitoring**

  The role of this component relates to both identifying the various types of virtualized environments, as well as monitoring their runtime behaviour and virtual resource consumption as particular (sets of) FUSION instances are deployed in such environments. This can be used to evaluate the performance and reliability of each virtual environment over time, as well as to determine the amount of spare capacity in each virtual environment for future service deployments (e.g., in case the same virtual environment is reused for deploying multiple FUSION instances). From these virtualized environments, more abstract (annotated) environments are derived, which are presented as such to the zone manager.

- **Placement**

  The role of this component is to encapsulate the various virtual environments as different environment types towards the zone manager as well as provide runtime-based information to

the deployment function regarding what existing virtual environment could be reused, or whether a new virtual environment (instance) need to be created and prepared first.

- **Deployment**

  This component is responsible for mapping FUSION instances onto the virtual environments. In case there is a one-to-one mapping between a FUSION instance and a virtual environment (e.g., a VM), this deployment operation could be relatively straight-forward. In case a single virtual environment is reused for deploying a number of FUSION instances, then a more complex deployment implementation is required.

3.4.1.3 Hybrid DCA design example

In Figure 24 (c) an abstract representation of a hybrid DCA is depicted. In this case, the underlying DC MANO layer has appropriate FUSION-enabled hooks for integrating several FUSION-specific aspects and concepts more tightly in the underlying software platform. Each of the key functional components (e.g., scaling, placement, monitoring and deployment) could have special hooks that allow for better control or a tighter feedback loop. One notable example of this is regarding session-slot based elastic scaling (see the next section).

3.4.2 Implementation

We currently have two implementations. One implementation is an example of a hybrid case, where we implemented a FUSION session slot based scaling mechanism by adding and integrating extra FUSION-aware capabilities in an OpenStack environment (see also Section 2.8.3). More specifically, we added functionality to Ceilometer [CEIL01] (which is the OpenStack monitoring module) to collect session slot information in OpenStack. Secondly, we augmented HEAT (which is the OpenStack orchestration module) to enable automated scaling in or out of service instances based on predefined boundaries and actual session slot usage.

By integrating this functionality in an existing cloud MANO infrastructure, FUSION service scaling can be delegated towards the underlying layer, allowing for a shorter feedback loop and tighter integration with the underlying infrastructure, and reducing the role and overhead of the zone manager in such environment.

A second prototype implementation is related to the integrated end-to-end prototype implementation. In that implementation, we developed a prototype of a simple physical DCA implementation, where FUSION services can be deployed on a single physical host using either Docker containers or KVM-based VMs. The main goal of this implementation is to validate the core FUSION interfaces between a zone manager and a DCA implementation, as well as to validate the end-to-end actual deployment and usage of FUSION (application and evaluator) services wrapped in Docker containers on a physical environment.

Ongoing work involves integrating that prototype in the vWall infrastructure, integrating the various components from the various partners on the shared environment, as well as providing a more advanced implementation of a DCA prototype that integrates with the heterogeneous cloud platform discussed in the next section.

A summary of the implementation status of the various DCA prototypes is depicted in Table 5.

Table 5 – Implementation status of DCA prototypes

Status	Task	Comments
Done	Openstack/HEAT scaling	Evaluation and validation of session-slot based service scaling in an industry de-facto standard cloud environment

Done	Initial API skeleton	Setup of initial skeleton implementation with all key public REST APIs in Python/Flask
Done	Docker container	Wrapping the prototype in a Docker container, managing other containers on the host from within a container. Note that we also added support for deploying all higher-level FUSION functions (e.g. a zone manager) on top of our DCA layer.
Done	Initial implementation	Initial implementation of a simple physical DCA, implementing the basic DCA functionalities within a single host, supporting both Docker containers and KVM VMs.
Done	Multi-config. instances	Added support for sharing service component instances across multiple services
Done	Persistent state	Managing both DCA state as well as service and instance configuration state
Active	vWall integration	Integrating this prototype on top of the vWall, both native as well as on top of OpenStack and/or an environment such as CoreOS.
Planned	Heterogeneous platform	Add support for more heterogeneous cloud platforms with different environments

3.5 Heterogeneous cloud platform

3.5.1 Design

Referring back to Section 2.2.1, the goal is to be able to **automatically** deploy demanding or time-sensitive applications in a cost-efficient manner on existing an novel cloud platforms and infrastructures, while providing a particular QoS towards these types of applications, for example by providing a better *performance isolation* in between such applications (see also Section 4.3.2.6). A conceptual drawing of how we envision automated service deployment in a heterogeneous cloud platform is shown in Figure 25.

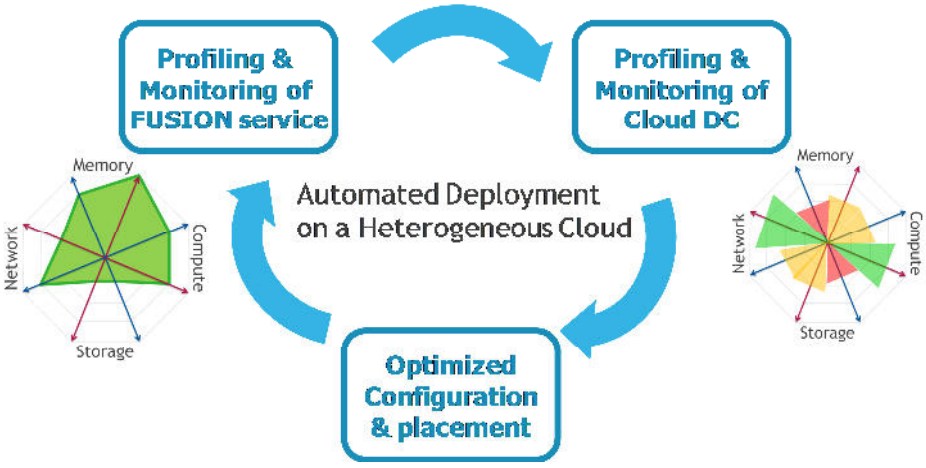


Figure 25 – Conceptual diagram of a heterogeneous cloud platform

From a high-level point of view, we envision three key steps that need to be integrated with each other in a continuous optimization cycle:

- On the one hand, we need to capture the requirements of applications and services through offline and online profiling and monitoring, to get an accurate view of the application behaviour on particular execution environments. For example, is the application memory-intensive, network-intensive, is it sensitive to packet latency, etc.
- On the other hand, we also need to characterize the capabilities and constraints of the various resources, hardware infrastructures and platforms, capturing both static and runtime information regarding each of these environments.
- Thirdly, we need to (continuously) optimize the placement and configuration of the different workloads on the available execution environments, by selecting the optimal resources and platform for deploying each application, minimizing cross-interference amongst the various workloads, and configuring the platform and resources to maximize throughput while minimizing jitter. An example of this is discussed in Section 4.3.2, where we illustrate an improvement of up to a factor 2x-3x when doing NUMA-aware placement configuration optimizations on a particular infrastructure for memory-bound applications. In that section, we also demonstrate the positive impact of providing better RT guarantees towards demanding and time-sensitive applications.

To implement this, we constructed the following initial monitoring architecture and design as depicted in Figure 26.

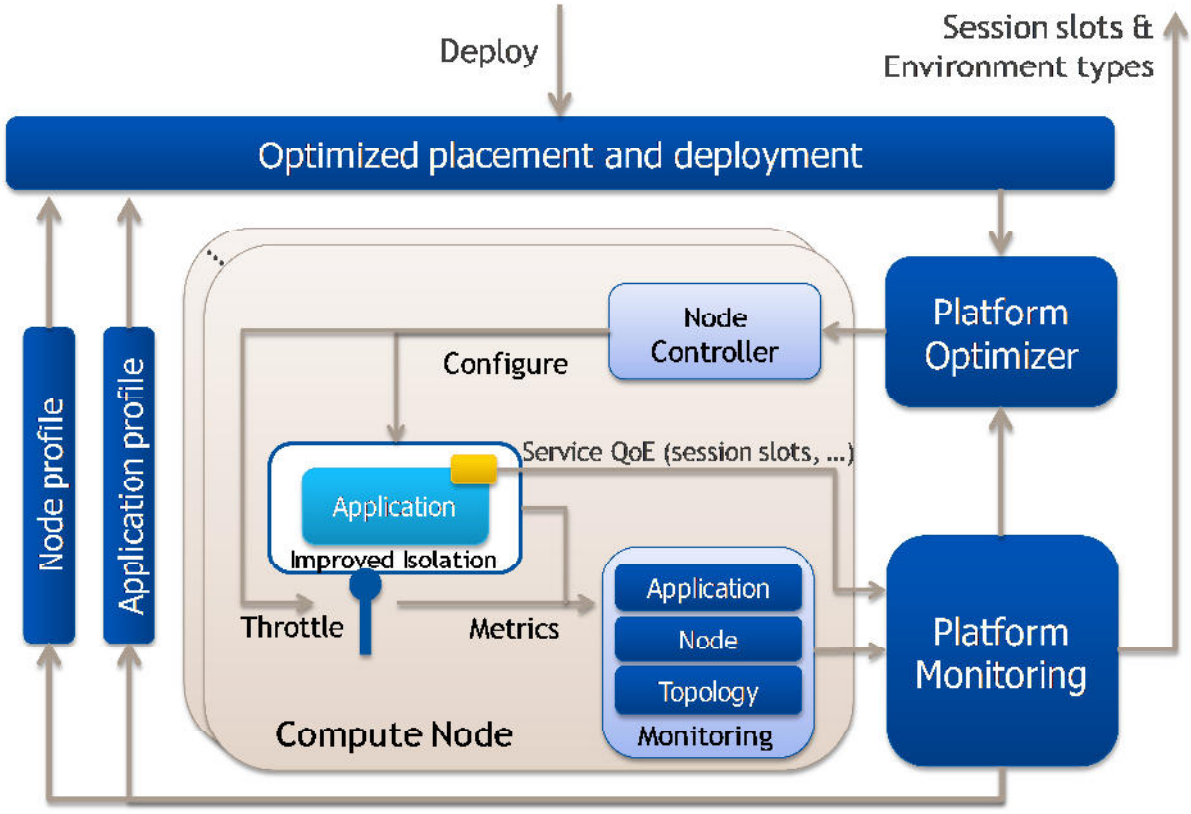


Figure 26 – Initial high-level design of a heterogeneous cloud monitoring platform

At the platform level, key building blocks and components include the following:

- **Optimized placement and deployment**, where the main responsibility is to determine the most optimal compute node for deploying a particular applications, based on the compute node and application profile. Note that in first deployments of unknown applications or on unknown hardware, the initial deployment may be suboptimal, but may improve for further deployments.

- **Platform optimizer**, whose responsibility is to manage the runtime state of applications and infrastructure resources, and trigger changes when needed (e.g., in case an application is causing too much destructive interference).
- **Platform monitoring**, whose responsibility is to collect, aggregate and analyze all monitoring data from all compute nodes, and forward this to the other components. Some of this information will be forwarded to the higher FUSION layers, such as application metrics and the set of abstracted environments.
- **Application and node profiles**, that are constructed from the monitoring data and which is subsequently used for further deployments of applications and management of the various compute nodes.

Within a compute node, we foresee the following key blocks:

- **Node controller**, which is responsible for deploying an application service on the compute node, creating the necessary environment and configuring the platform and (performance) isolation mechanisms. This controller is also responsible for throttling or reconfiguring applications or their surrounding environments in case something goes wrong at runtime.
- **Monitoring agent**, which is responsible for measuring and collecting all relevant data from both the compute node itself as well as the application environment.
- **Application environment**, which is the environment created by the node controller for isolating applications on a compute node from a functional as well as performance perspective.

In the final year of this project, we will present a more detailed design of a heterogeneous platform as well as how this is integrated into the FUSION service layers, specifically the DCA and Zone Manager.

3.5.2 Implementation

Based on the various lower-level dedicated experiments we have done (see also Sections 3.7.3 and 4.1), we are currently starting to implement a prototype implementation for such feedback-driven optimized heterogeneous cloud platform. The goal is to have a PoC environment that demonstrates the benefits of overall concept through a number of specific use cases. We will also link this with a corresponding DCA implementation and evaluate the role and impact of specific evaluator services in the overall platform. For this, we will leverage many of the enabling technologies and mechanisms for providing better resource efficiency and RT guarantees, for which we already have done a wide range of experiments, of which a summary is provided in the evaluation part in Section 4.3.

As a first step, we already started implementing a profiling tool for characterizing the key runtime characteristics of a particular VM or container. In Figure 27, an example is depicted of a few of these characteristics for a media format conversion benchmark. Specifically, we show CPU utilization, memory throughput, and CPU power consumption (block storage and networking metrics are not depicted to save space). The application-specific monitoring data is depicted by the yellow regions; the rest involves the booting and termination of the VMs. Note that these are just a few of the numerous low-level infrastructure and platform metrics we intend to capture (dynamically and intelligently) while profiling and monitoring new applications on a new infrastructure.

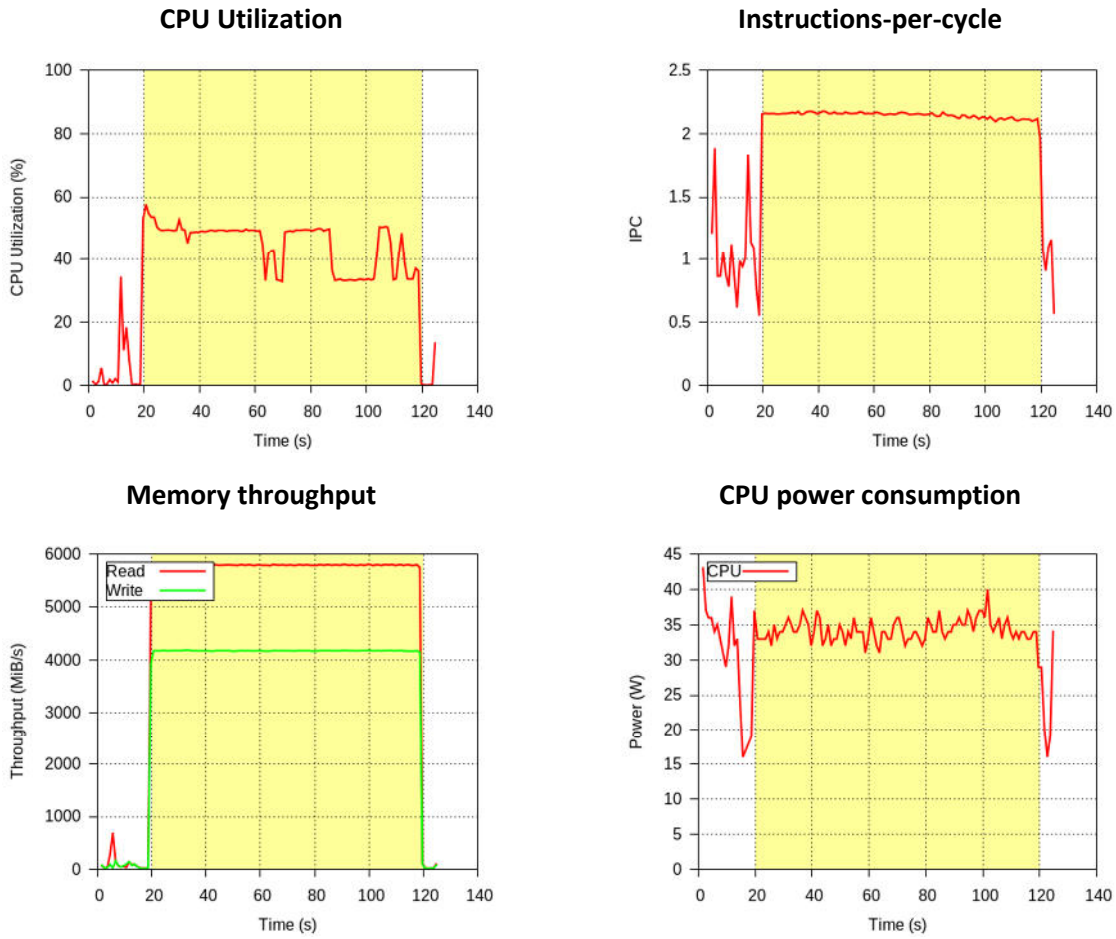


Figure 27 – Example profiling data, covering the most relevant resources for that benchmark

Based on this profiling data, we then create an overall application profile on a particular environment. An example normalized high-level profile is depicted in Figure 28 for the same media conversion benchmark used earlier. In this example, the application clearly is memory-bound, for which memory-specific platform optimizations may apply. This high-level profile provides insights in the main characteristics of a particular application with respect to the type and intensity of the various resources that are being used. We plan to extend and use these profiles for optimizing how applications should be deployed on particular resources.

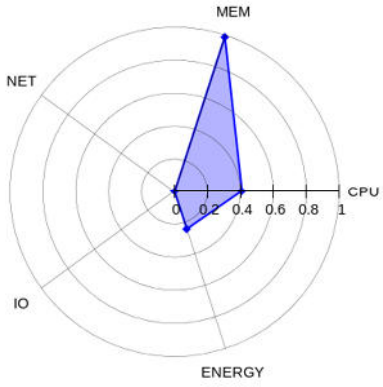


Figure 28 – High-level normalized profile for the memory-intensive media conversion application

A summary of the current status of this work is provided in Table 6 below.

Table 6 – Implementation status of a heterogeneous cloud platform

Status	Task	Comments
Done	Initial high-level design	Define initial high-level approach for efficiently deploying demanding applications in a heterogeneous cloud environment.
Active	Profiling infrastructure	Setting up an initial profiling infrastructure, building a framework based on existing tools.
Active	Refined design	Refining initial design based on initial profiling data and experimental results.
Planned	Dynamic profiling	Improved dynamic and flexible profiling and monitoring infrastructure.
Planned	DCA Integration	Integration into a FUSION DCA prototype and demonstrate benefits for FUSION applications

3.6 FUSION orchestration protocol specifications

This section covers the overall design decisions and format for the FUSION orchestration and management interfaces. We also elaborate on the current implementation status of these specifications. For the full overview of REST APIs, we refer to Section 7.

3.6.1 Design

As mentioned before, we designed both the inter and intra layer protocols based on RESTful principles. REST has the advantage of being a lightweight cloud-friendly protocol on top of the prevalent HTTP protocol for which many excellent implementations and frameworks already exist. For these good reasons, REST has become one of the de-facto standard protocols in the Internet and Cloud for managing resources and state.

As such, we designed all FUSION orchestration and management related interfaces using REST principles, focussing on the various resources that FUSION manages (e.g., services, instances, zones, slots, offers, etc.).

With respect to authentication and authorization, we rely on the available mechanisms built into HTTP and HTTPS for authenticating incoming requests, which could also be combined with authentication tokens as is done in OpenStack [KEYS01]. All entities, both physical entities as well as software components, first need to be registered to the relevant component before being able to perform particular actions on that component. For example, a service provider first needs to be registered before it can start registering and deploying services. Similarly, a zone manager software component first needs to be registered to a domain orchestrator before it can register itself and interact with a domain. Each registered entity has one or more associated roles with which it can perform a subset of all operations as well as have access to a subset of all operations. Typical roles include an admin (user), service provider (user), domain orchestrator (SW component) and zone manager (SW component).

This concept of users and roles enables to define only a single one public (REST) interface for each key FUSION component, where all interactions with the external entities are managed. Within such FUSION component (e.g., a domain orchestrator or zone manager), there may however also be

internal APIs so that each subcomponent (e.g., FUSION domain scaling) can communicate directly with the other subcomponents (e.g., FUSION domain deployment).

We mainly used JSON in our prototype implementation. As such, we also present the REST API functions using JSON syntax.

Note that for all JSON response messages, we currently assume the following basic structure:

```
{
 “status”: “FUSION status code”,
 “message”: “FUSION status message”,
 “response”: <FUSION response body>
}
```

As such, to avoid unnecessary repetition in all REST API specifications, we will only provide an example response body. The FUSION status codes augment the existing HTTP status codes when resources are for example created (201), queried (200) or deleted (204), and apply especially when something goes wrong.

Below an example REST specification regarding FUSION services in an execution zone. The HTTP headers regarding authentication, message format, expiration, etc. are not depicted.

<table>
<tr><th colspan="4">/1.0/services</th></tr>
<tr><td rowspan="6">GET</td><td colspan="3">Returns a list of all registered services in this execution zone, along with their current state and session slot information.</td></tr>
<tr><td colspan="3">REQUEST PARAMETERS</td></tr>
<tr><td>Parameter</td><td>Type</td><td>Short description</td></tr>
<tr><td>None</td><td></td><td></td></tr>
<tr><td colspan="3">EXAMPLE RESPONSE</td></tr>
<tr><td colspan="3">[
{ “serviceid”: “epg1.bell-labs.be”,
“state”: “deployed”,
“slots” : {“total”: 100, “free”: 25}
},
...
]</td></tr>
<tr><td rowspan="6">POST</td><td colspan="3">Register a new service in this execution zone.</td></tr>
<tr><td colspan="3">REQUEST PARAMETERS</td></tr>
<tr><td>Parameter</td><td>Type</td><td>Short description</td></tr>
<tr><td>serviceid</td><td>String</td><td>Unique service name or identifier</td></tr>
<tr><td>manifest</td><td>TOSCA</td><td>Full description of the service to be registered in the zone.</td></tr>
<tr><td colspan="3">EXAMPLE RESPONSE</td></tr>
<tr><td colspan="3">{“state”: “registered”}</td></tr>
</table>

3.6.2 Implementation

As mentioned before, the FUSION prototype components are being implemented in Python, using the Flask and Flask-Restful modules. This allows us to focus on the APIs and their corresponding functionality, rather than having to spend much effort in implementing and parsing the REST interfaces, or using complex but flexible and highly optimized frameworks.

Using these frameworks, we already implemented the key public interfaces, implemented initial working implementations, as well as integrated already all key components in a working initial prototype using these public interfaces, as is described also in more detail in Deliverable D5.2. This integration effort serves two purposes, namely to validate the APIs and to evaluate the FUSION functionality and coordination of the various FUSION layers.

We are currently designing and implementing also the internal interface of the various subcomponents, upon which we will document in more detail in the next deliverable together with the finalized version of the public interfaces.

The full details on the public interface specification can be found in Section 7.

3.7 Inter-service communication protocols

3.7.1 General

Composite services and collaborating services are interconnected services whereby overall service behaviour and performance is partly determined by the interconnecting communication and the characteristics of the interconnecting platform.

In FUSION, an architecture model is envisioned where re-usable components are maximized when architecting a service so that at run time, the reuse of components can be maximized. This inherently implies the use of inter-service communication protocols.

Depending on where collaborating services are instantiated, communication channels and their characteristics can differ. An exemplary overview of throughput and latency for some available interconnect technologies and service models is given in Table 7[4].

Table 7 – Performance characteristics of different inter-service communication paths

	Intra-host communication							Intra-host	
	Inter-Process			Inter-VM			Inter Container	Inter-Process	
	TCP via loopback ip	TCP via loopback IP +NUMA-pinning	TCP via Linux bridge +NUMA-pinning	TCP via Linux bridge +NUMA-pinning	ivshmem	netcat	TCP via Linux Bridge	TCP ISCP via 1Gb itf	TCP offload RoCE via 40Gb itf
Throughput (Mbps)	27120	30028	8791	870	367	266	9200	960	24353
Latency (usec)	~15	~12	~13	218	12	220	17.7	74	962

Note: for IVSHMEM, only 367 Mbps was currently measured, although for IVSHMEM, at least 15000Mbps should be attainable (and possibly up to 100000Mbps). Only 367Mbps was listed here because of memcpy taking too long. Likely this is because of non cacheable memory copies needing to go over pci for every memory access. The root cause analysis of this problem was not yet finished at the moment of writing of this document)

From those results, it becomes clear that choosing and optimizing the most appropriate inter-service communication channel can have significant impact on throughput and latency. Concerning the architecture of services and specifically regarding the communication between two services or service components, the following observations can be made:

1) Previous technologies like CORBA [CORB01], DCOM [DCOM01], etc. were designed to have services communicate with one another as if all communication is modelled as a local function interaction, even if the communication takes place over a e.g. a switched network. In other words, the underlying communication mechanism/infrastructure is abstracted away. The issue however is that the bandwidth and latency are vastly different in case the communication is over:

   - an actual network
   - between processes or OS threads on a single machine
   - between threads within a process

   The infrastructure and the way services are deployed on these infrastructures can impact the overall design of such services. For example, in case a service has such high performance or soft real-time requirements, it may not be possible to model such a service as a graph of collaborating service components, especially if the various abstraction layers are unaware about

[4] The figures listed were obtained on and between 2 SuperMicro H8DGG servers with two AMD Opteron 6174, 1Gbps Intel NICs and Mellanox connectX3 interfaces 40Gbps with RoCE.

the service requirements and vice versa, the service is unaware of the underlying interconnecting network capabilities.

The above indicates that in order to optimize the overall system behaviour, there is a need to describe the various characteristics of the services themselves and the interconnecting infrastructure, e.g. in a manifest or through evaluator services that can specifically test for the existence of particular communication channels and their corresponding capabilities.

2) The current approach to service architecture and modelling is to have collaborating services communicate over a network even when there is no real network involved (e.g. in case of communication within a single host). This architecture mainly originates in the use of virtualization (virtual machines, multi-tenancy onto single hosts etc.) and clear separation of functionality under the form of re-usable component software. This approach is also made possible through e.g.:

   - the use of highly efficient lightweight threading such as go-routines or gevent [GEVE01] in Python;
   - the use of message-oriented patterns popularized by Go channels or Erlang/OTP [ERLA01].

   In this second approach, the "network" is actually exposed and parameterization and optimization of the "network" can be achieved to the benefit of service performance and robustness.

However, (considering the second option described above), with the introduction of a network between services components, concepts like packetization, MTU, routing etc. get introduced as well. The "network" between components is tunable or parameterizable into a wide variability, e.g.:

- On a single host, when passing a message from one service component to the next, the MTU could actually be set quite large in such a way that the entire message can fit into one packet. For example, in case of shared memory communication protocols such as IVSHMEM, one entire message or frame could be sent as one packet.
- In case of collaborating services in a single data centre with networks that generally are characterized by high bandwidth and low latency interconnects, the overall benefit at the level of the data centre is optimal use of nearness/locality of collaborating components through minimizing network resources. Further improvement can be obtained by tuning the service component interconnections (e.g. via DCTCP, see further).
- In case of collaborating components residing on two different hosts with high compute capability and with an interconnecting network that is characterized by low bandwidth and high latency, the injection of compression functionality can contribute to overall service improvement and load density.
- Non-functionals like injection of security only have to be provided on a need basis. For example, if one owns the host and the services that run on it, one can get higher service loading density with no security).

In case tuning is not done or in case the characteristics of the network layer are not taken into account, suboptimal network loading and behaviour can impact composite services or collaboration services and also impact the overall usage of the infrastructure that translates in direct economical impact.

From the previous arguments, in order for cloud orchestration to take advantage of the benefits, it needs information about:

- the configurability of the hosting and networking infrastructure and parameterization

- the service's adaptability and configurability towards the infrastructure that it will be hosted upon. Specifically for networking, this indicates that the service description needs to be able to describe what metrics it wants from the interconnect, or be able to test available capabilities via the evaluator services.

The TOSCA specification already provides service graph specification, is standardized in OASIS and has traction in cloud and NfV domains. Through extension of TOSCA at the level of

- service and infrastructure monitoring specification
- specification of service and interconnection constraints (allowing to describe what is expected of the realization of a service graph description.

and through the use of appropriate TOSCA engine functionality in the FUSION framework, the envisioned inter-service communication can be optimized.

3.7.2 FUSION late binding

Currently, the communication paths between collaborating services are mostly determined at service design time and therefore are rather static and still largely oriented towards TCP/IP because of general availability. However, at service instantiation or relocation time, the actual infrastructure onto which the service is realized is selected and configured.

Late binding or determining the appropriate communication channel between two services at service instantiation or even service invocation time allows for a better use of the infrastructure both at service level as well as overall orchestration level. Tuning the selected communication channel further improves efficiency of both levels. The following paragraphs describe the approach taken to realize late binding.

To categorize the different levels in service deployment and how efficiency can be introduced, the following service deployment states in orchestration are considered/distinguished:

1) Select platforms and network interconnects (based on metrics describing the network)
2) Instantiate service decision (e.g., evaluating requirements described for the service against the available infrastructure)
3) On actual instantiation, tune service component interconnects (e.g., setting default MTU in VMs, setting default TCP window size, ...)

To enable late binding for each of these stages, the following is required:

1) Through specification in service description and through infrastructure description and runtime generated information
2) Placement algorithm needs to account for different inputs and requirements
3) With service specification as input, and input of infrastructure configuration information, perform the necessary configuration actions in the infrastructure (VMs, containers, network layer, ...)

An alternative approach is that at instantiation time, the infrastructure (host) and data centre management activates a network optimization without FUSION or the service being aware that the underlying infrastructure is tuned or optimized. In such case, FUSION cannot directly impact the overall optimizations except from observing (e.g., through the evaluator services) that the communication path on such infrastructure is more optimal.

3.7.3 Late binding considerations

Different networking technologies can be used for inter-VM communication from which some are discussed in the following paragraphs. The technology evaluation and actual measurements are described in Section 4.5. The aim is to indicate:

- the intricate configuration aspects of the different networking technologies;
- describe in short the data plane benefits each technology can offer;
- the configuration aspect of their use in inter-service communication.

The latter topic will indicate the diversity of the technologies and serves as an input for deciding the abstraction and specification level that a service manifest should deal with when specifying networking requirements of a service at design time.

3.7.3.1 SR-IOV Inter-VM communication

3.7.3.1.1 Technology

An explanation on Single Root IO Virtualization technology can be found in [SRIO01]. In essence, SR-IOV provides a hypervisor bypass so that VMs each can attach to a separate Virtual Function and share a single physical PCIe device (e.g., a NIC or GPU card).

An SR-IOV NIC supports multiple contexts on top of a single physical context. Each context is then represented as a PCI device that can be passed through to the VMs. A context on top of the physical context is used when passing through this SR-IOV interface towards a VM. The physical context is under control of the host itself. Communication between SR-IOV and host (regular) or between SR-IOVs is performed on the NIC itself. (Figure 29 is obtained from figures 4 and 5 in [SRIO02] and highlights the concepts of SR-IOV technology.)

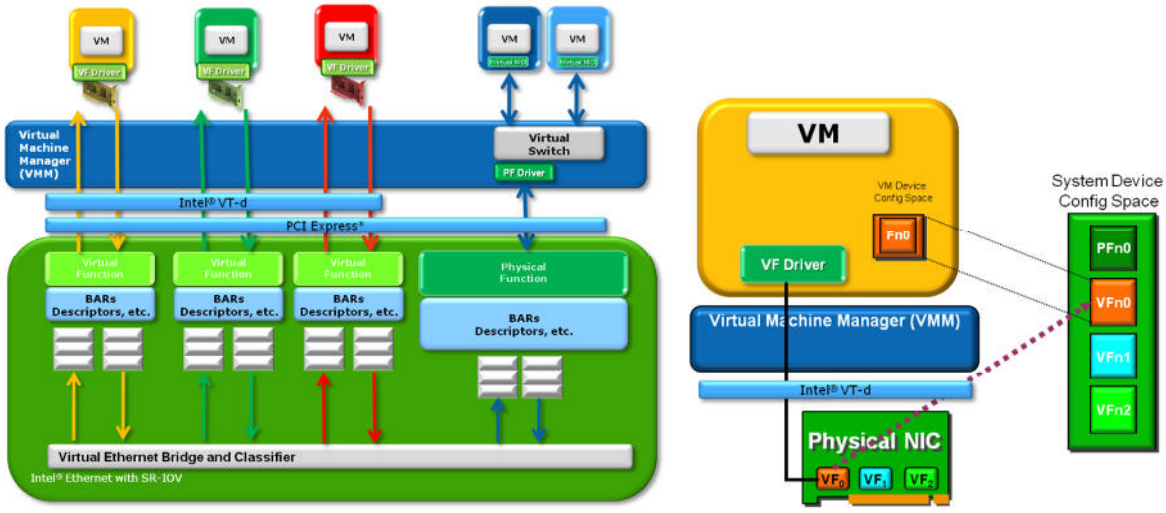


Figure 29 – SR-IOV architecture

3.7.3.1.2 Late binding

From the technology description, SR-IOV is a technology that improves mainly on throughput and situates itself, from a configuration point of view, at the level of the network, hypervisor and host. It requires network interface cards (or other peripheral devices such as GPUs) that are SR-IOV enabled (due to the HW user context support).

From a service point of view, SR-IOV interfaces are presented as regular interfaces that can be used by a service.

Given the above, a service manifest merely needs to specify an affinity of a service towards throughput. This will allow the cloud orchestration to select the appropriate networking technology to use for a service given the actual (and predicted future) state of the infrastructure it governs.

Evidently the service manifest should contain a minimum throughput parameter to ensure normal service operation.

3.7.3.2 IVSHMEM inter-VM communication

3.7.3.2.1 Technology

With IVSHMEM, VMs can communicate with each other through shared memory. The overall architecture is depicted in Figure 30. In brief, on a host, a shared memory region is allocated. This memory region is passed into the guest as a PCI device. Through the PCI registers, the guest can communicate with the host. The IVSHMEM kernel module, handling the PCI device, provides this functionality.
On the host, an IVSHMEM server coordinates the communication between host and its clients and between clients. Using a doorbell mechanism modelled on top of the PCI registers, guests can send and receive notifications between one-another.

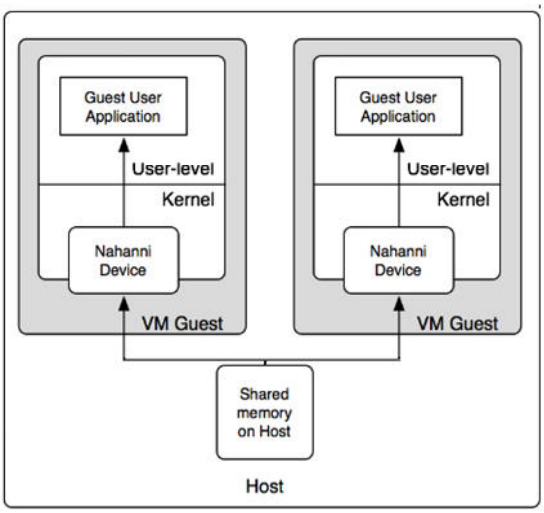


Figure 30 – High-level architecture diagram of IVSHMEM inter-VM communication

For a more detailed description of the envisioned shared memory technique, notably, IVSHMEM or also called Nahanni, see [IVSH01].

3.7.3.2.2 IVSHMEM Late Binding

IVSHMEM is a technology that improves mainly on throughput and latency in case of communication remains “on host”. It requires the presence of a kernel module in the guest OS and a coordinating server application on the host OS. The applications need to be designed to work with the shared memory communication channel. Furthermore, the collaborating VMs/applications/services need to learn the IVSHMEM destination ID of their peer services for the collaboration to work. Shared memory technology imposes that collaborating services trust one another.

Given the above, a service manifest has to indicate that the service is “IVSHMEM” enabled and to which PCI device ID the interface should be mapped. Orchestration can then deploy the collaborating services onto the same host, setup the coordinating server at host and provide the necessary VM configuration parameters for IVSHMEM to work.

Evidently the service manifest should contain a minimum throughput parameter to ensure normal service operation. From security point of view, the manifest should indicate a trust relationship with

collaborating services. Note that some of this functionality could be hidden behind a higher-level API, shielding the applications from all intricate details of this communication mechanism.

3.7.3.3 Dctcp

3.7.3.3.1 Technology

A lot of internet communication is transferred using TCP. TCP offers the following functionality:

- connection setup and teardown
- ports so multiple connections between systems can occur simultaneously
- flow control via a sliding congestion window
- reliable, ordered delivery via independent, bidirectional sequence numbers

TCP's sliding congestion window allows a sender to send a certain amount of data on the wire without having yet received an acknowledgment from the receiver. The size of the sliding window is optimally equal to the bandwidth delay product (BDP) [WIKI01) and when configured appropriately allows to fully utilize the interconnect.

TCP window size is a possible tuning factor with important benefits in case of increased round trip times (RTT) or low bandwidth links. In data centres however, RRT delays are rather short (in the order of milliseconds) and links are high bandwidth, so additional benefit can be gained from congestion control.

For increasing BDP, more unacknowledged data is in transition of being delivered. In case of network hiccups/irregularities/transmission errors, the risk of clogging networks increases as more data might be lost and/or needs to be retransmitted. Therefore controlling or handling congestion is an important factor in overall throughput. And this is the aim of DCTCP.

In the below paragraphs a few different versions of DCTCP are mentioned.

- **DCTCP**

  DCTCP is an enhancement to the TCP congestion control algorithm that uses Explicit Congestion Notification (ECN). In case of congestion, DCTCP sources extract multi-bit feedback from the ECN marks by estimating the fraction of marked packets. In doing so, DCTCP sources react to the extent of congestion, not just the presence of congestion as in TCP. This allows to work with very low buffer occupancies and simultaneously achieve high throughput. A full analysis of DCTCP can be found at [ALIZ11].

- **D2TCP**

  This is a deadline aware TCP. For further information, see [BALA12].

- **D3TCP**

  D3 is a deadline-aware control protocol that is customized for the data centre environment. D3 uses explicit rate control to apportion bandwidth according to flow deadlines: presented and described in [WILS13].

- **DIATCP**

  Deadline and Incast Aware TCP for Cloud Data Centre Networks by Jaehyun Hwang, Joon Yoo, and Nakjung Choi, representing recent work of Bell Labs on data centre and TCP [HWAN14].

3.7.3.3.2 Late Binding

From the technology description, DCTCP and variants are technological improvements on TCP.

1) As with the other technologies, a service manifest should also specify an affinity of the service towards throughput and latency. To cover the throughput variability of TCP, a margin could be specified.
2) From a service point of view, it suffices to indicate the use of TCP thereby covering all TCP improvement protocols.

Given the two topics above, this will allow the cloud orchestration to select the appropriate networking technology to use for a service given the actual (and predicted future) state of the infrastructure it governs.

Evidently the service manifest should contain a minimum throughput parameter and maximum latency to ensure normal service operation.

3.7.3.4 Docker libchan

3.7.3.4.1 Technology

Libchan is an ultra-lightweight networking library which lets network services communicate in the same way that goroutines communicate using channels:

- Simple message passing
- Synchronization for concurrent programming
- Nesting: channels can be sent through other channels

Libchan sessions that remain on the host can benefit from a zero copy go channel to provide efficient inter-container communication.

Remote libchan sessions are regular HTTP2 over TLS sessions, and can be used in combination with any standard proxy or authentication middleware. When configured properly, libchan communication channels can be safely exposed on the public Internet.

Libchan is designed so that any message serialization format can be plugged in, e.g.: json, msgpack, xml, protobuf, etc.

3.7.3.4.2 Late Binding

From the technology description, libchan is a networking library that can be used in case of Go programming language designed services. The main benefit of libchan is the injection of functionality (HTTP2 and TLS) and from security point of view allow to work cross data centres.

A service manifest describing such a service should indicate that the service is “libchan” enabled. This allows orchestration to deploy “libchan” collaborating services and to select the appropriate networking technology to use for a service given the actual (and predicted future) state of the infrastructure it governs. Evidently the service manifest should contain a minimum throughput parameter to ensure normal service operation. From security point of view, the manifest should indicate any security settings for the collaborating services.

3.7.3.5 RoCE

3.7.3.5.1 Technology

RDMA over Converged Ethernet is in fact an hardware accelerator for inter-host communication. For a brief introduction and overall architecture, see [LEE10].

Principally, RoCE uses an specific Network Interface Card that provides for network transport as well as an entire optimized OFED network stack (cfr [OFED01] and [OFED02]).

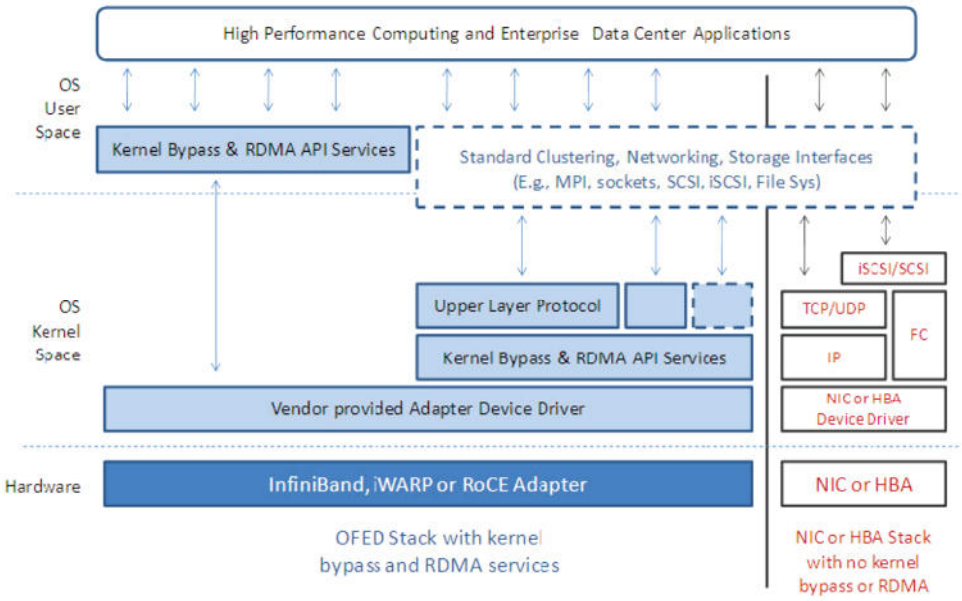


Figure 31 – Comparison of OFED stack versus classical IP software stack

OFED provides the following functionality:

- a technique of channel-oriented **R**emote**D**irect**M**emory**A**ccess,
- send/receive operations,
- kernel bypasses of the operating system,
- both a kernel and user-level application programming interface (API),
- services for parallel message passing (MPI),
- sockets data exchange (e.g., RDS, SDP).

3.7.3.5.2 Late Binding

Given that RoCE is a HW accelerator for protocol stack offloading, a service manifest describing a service should indicate that the service is “RoCE” enabled.

This allows orchestration to deploy “RoCE” collaborating services and to select the appropriate networking technology to use for a service given the actual (and predicted future) state of the infrastructure it governs.

Evidently the service manifest should contain a minimum throughput parameter and maximum latency parameter to ensure normal service operation.

3.7.4 Performance measurements

Some performance related measurements of each of these technologies are discussed in Section 4.5.

3.7.5 FUSION service manifest and late binding

The FUSION service manifest specifies a service and its dependencies at design time and at service registration time.

Concerning late binding and its information as described in the service manifest, the manifest serves as an input for orchestration and placement so that services can be deployed according to their requirements and conforms to their SLAs and overall at zone and domain level to achieve overall optimization.

Additionally, the FUSION service evaluators could provide information about a service's networking dependability, affinity and provide information for late binding of collaborating services. FUSION service evaluators work at deployment time. FUSION orchestration and placement (domain and zone) can use both manifest and evaluator information as input to select the most appropriate interconnect technology in the perspective of an overall domain and zone optimization.

On a general level, a FUSION service manifest should:

- Specify an affinity of a service towards throughput or latency sensitivity, on service and service component level.
- Contain a minimum throughput parameter and a maximum latency parameter on a service and a service component level to ensure normal service operation.
- In case specific networking requirements are needed (e.g. RoCE, DPDK), specification of the specific parameters that are needed for the given technology and that allow orchestration to perform an optimal placement. All of the networking technologies that a service or service component support should be enumerated. Note that this could be specified also one more abstract level, for example in case the service supports some optimized communication library, which supports a number of specialized communication channels.

3.7.6 FUSION service manifest, framework and DCA interaction

Service manifest contains complete service description with all supported interworking functionality as well as an evaluator service. The manifest and supported networking enumerations can be used by the FUSION framework (domain manager, zone manager) to combine this design time information with runtime information from the evaluator service and eventually constrains the possible selectable hosts.

This part of the manifest can be passed on to the DCA layer as metadata so that the DCA can filter the number of available hosts and provide the necessary configuration settings towards the hosts upon which the service will be instantiated.

4. EVALUATION OF ENABLING TECHNOLOGIES AND ALGORITHMS

This section describes first evaluation results of specific WP3-related concepts and technologies, including feasibility studies, evaluation of specific (e.g. placement) algorithms or specific prototypes of specific subcomponents. Whereas all integration-related evaluation work of this year is done in WP5 and described in detail in D5.2, this section focuses on the evaluation of specific individual components, algorithms or enabling concepts, defined and evaluated outside the scope of the integrated prototype. This typically involves specific experimental setups outside the setup for deploying and evaluating the integrated prototype.

4.1 Multi-configuration service instance modelling

We already discussed the concept and benefits of sharing the available resource slots of service component instances across more than one service (configuration). In this section, we quantify some of these benefits by representing the session slots and session slot sharing as a simple MMC queuing model. In this model, *C* represents the number of available slots for a particular service. These slots can either come from a single instance or from multiple instances. In the latter case, we assume a load balancer for mapping the incoming service requests onto the available session slots from the different instances.

In this model, we assume a constant service request rate λ and a constant service processing rate μ. Unless stated otherwise, we assume μ to be 4 requests per hour. In other words, one session lasts on average 15 minutes. We assume a classical Poisson distribution for the arrival and processing rates. Using this MMC model, multi-configuration services can be modelled as depicted in Figure 32.

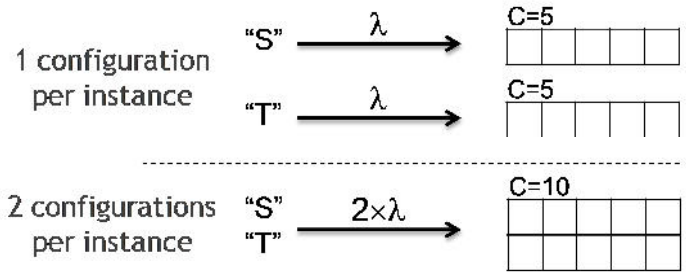


Figure 32 – Separate versus shared session slots with single and multi-configuration instances

In this example, there are two services, named *S* and *T*, with the same service request rate λ and service processing rate μ. In the single-configuration case, they each having 5 resource/session slots available. If all 5 slots of a service are occupied, the clients need to wait before they can access that service. In the dual-configuration case, the 5 session/resource slots of both services are combined in a larger pool of 10 available slots. In this case, both the incoming requests for service *S* and *T* will be handled by one of those available slots. The aggregate arrival rate for this pool of resource slots is $2\times\lambda$.

In the first set of results, we assume a constant amount of session slots per service. In the second set of results, we will then compare the minimum required session slots for a target maximum waiting time or waiting probability.

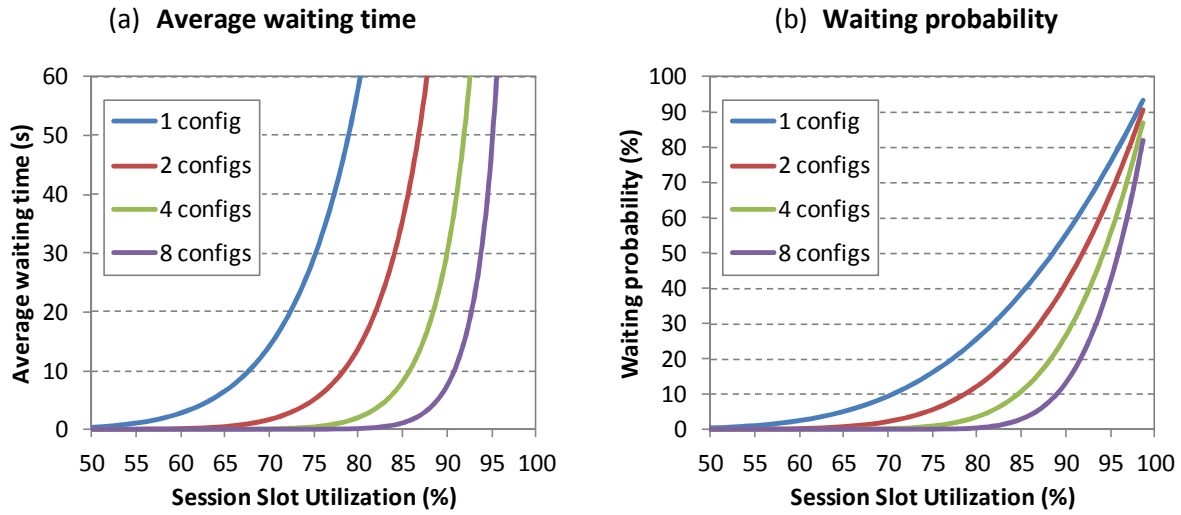


Figure 33 – Average waiting time and waiting probability as a function of the average session slot utilization for C = 20 (per service) and μ = 4/hour.

In Figure 33, the average waiting time and waiting probability is shown as a function of the average session slot utilization factor ρ. We assume 20 available session slots per service. As can be observed, being able to share a larger pool of available session across multiple service configurations results in a much higher session slot (i.e., resource) utilization factor for a similar waiting time or waiting probability. This follows the intuition that a larger shared buffer can cope better with particular outliers, as the probability that multiple outliers coincide is much lower than only a single outlier. As an example, in case the waiting probability should remain below 1%, then the session slot utilization can increase to 80% when eight configurations can be mapped onto the shared pool of 160 resource slots, compared to only about 50% in case each service has its own limited pool of 20 resource slots.

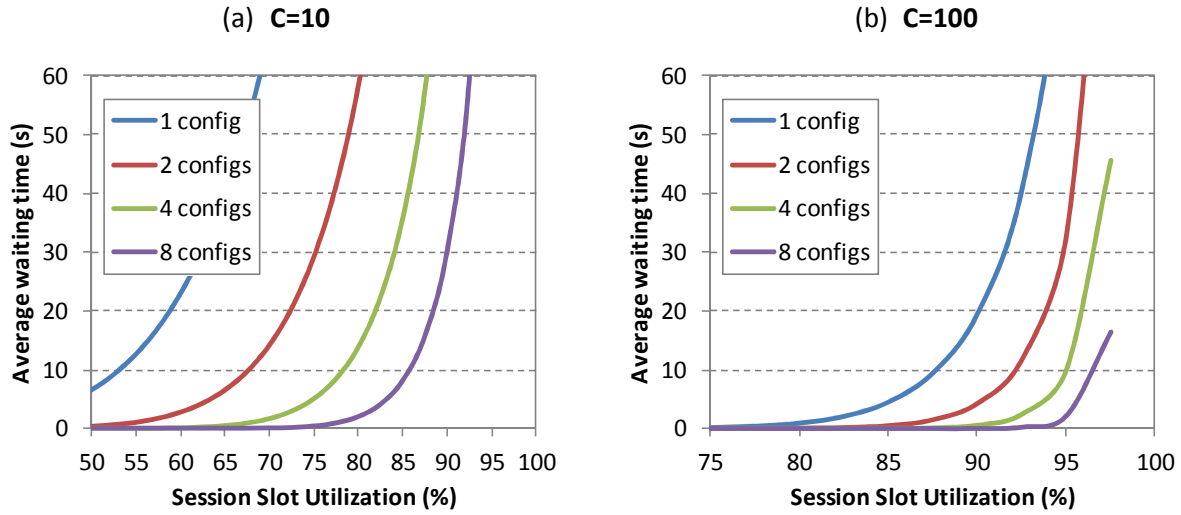


Figure 34 – Average waiting time as a function of the average session slot utilization for C = 10 and C = 100 (per service) and μ = 4/hour.

In Figure 34, the average waiting time is shown when the number of available session slots per service is smaller (C=10) or larger (C=100). As can be expected, the relative impact is larger for smaller individual pools of available session slots. In case of 10 session slots per service, the average waiting time for an average session slot utilization of 50% is already 8 seconds, whereas for eight service configurations mapped onto the same resource slots, the average waiting time for a session slot utilization of 75% is still below 1 second. As we envision in FUSION that services can be deployed

in many small execution zones, each supporting only a limited number of session slots per deployed service, this multi-configuration feature can significantly improve the resource utilization factor of these already scarce and expensive resources.

Vice versa, this also means that for a particular target maximum waiting time or probability, fewer session slots (i.e., resources) need to be reserved to handle the same incoming load. This is depicted in Figure 35.

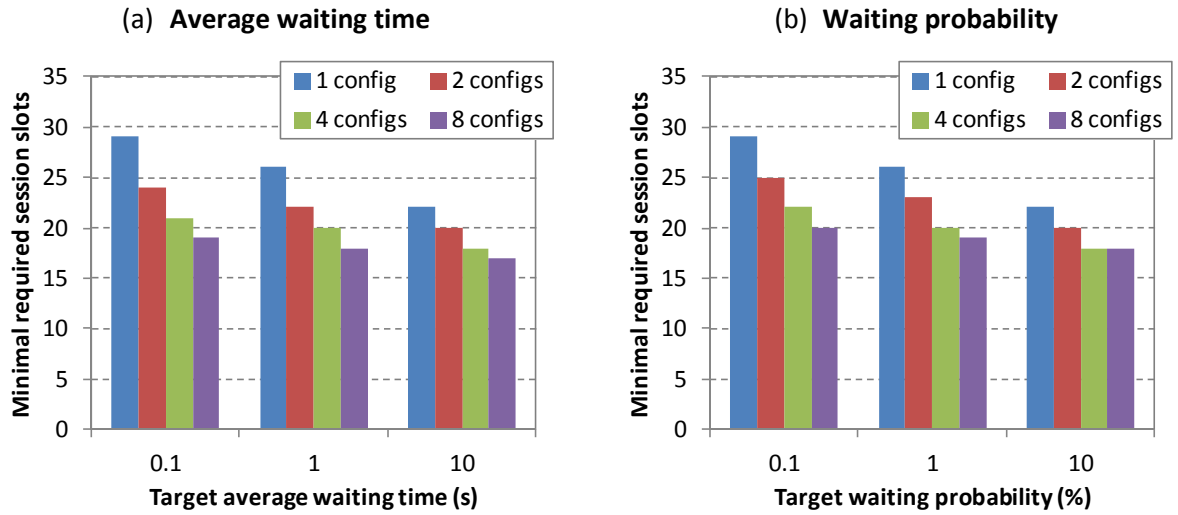


Figure 35 – Minimal required session slots as a function of the target average waiting time or waiting probability for a constant service request rate of λ = 1/minute and μ = 4/hour.

For a target waiting time of 0.1 seconds, the minimum amount of session/resource slots that need to be available can be reduced by up to 33% (i.e., from 29 to 20 per service) when the resource slots can be shared across eight service configurations. Note that in FUSION, we target almost zero average waiting time, by combining a service resolution plane, session-slot based scaling and a lightweight deployment model (i.e., clients should never have to wait). For higher service request rates (and a constant service processing time), the relative reduction is lower (due to a higher amount of session slots that need to available to handle such load); for lower service request rates, the relative reduction in required session slots is even higher.

Although the main findings are already well-known in the Telecom community, in this section, we specifically showed the benefits and necessity from aggregating and overlaying multiple services on top of a similar pool of session slots, especially in smaller DCs with limited resources onto which FUSION services will be deployed.

4.2 Docker-based service provisioning

In this section, we present some initial results regarding the efficiency of Docker for provisioning Docker container images onto a Docker host environment. In a distributed environment such as FUSION, the images of new FUSION services first need to be downloaded from a (distributed or centralized) image repository before they can be deployed and instantiated on a particular system. Similarly, within a particular execution environment, it may be necessary to first download the image from a file server onto the target host node (e.g., in case not a distributed file system is used).

We focus mainly on the average time to download the appropriate image and artefacts from a remote repository or file server, as this typically represents the largest fraction of the provisioning latency in case the image is not locally available. For example, in case of Docker, creating a new environment for a container instance (i.e., *docker create*) only takes about 0.05 seconds on our

server[5], and starting a created container instance (i.e., *docker start*) typically takes less than 0.2 seconds on the same server.

As discussed already in Section 2.6.1, Docker uses a stacked imaging system, where containers typically depend on other more general container images, thus creating a stack of image layers on top of each other. Multiple (independent) containers can share the same intermediate image layers in a tree-like fashion (one container can only have one parent container image). Docker supports both private and public registries for storing and fetching these container images.

For example, for many containers, the base layer (i.e., the lowest image layer) typically represents a particular Linux distribution (e.g., ubuntu, debian, fedora, etc.), containing the corresponding basic libraries and executables of that Linux distribution (but excluding the Linux kernel). All containers depending on the same base layer image effectively will share that image layer. As Docker only needs to fetch each image layer once from a remote registry, this means that being able to share the base and intermediate layers can greatly reduce the overall provisioning time of a new Docker container on a particular host, if one or more of these layers have already been provisioned earlier in the context of other (possibly even independent) containers. As such, the application-specific top layers should be kept as small as possible.

In Figure 36, the current Docker image tree is depicted of the various FUSION application and system prototype components that have been created so far for the integrated prototype of WP5. The total accumulated virtual size of all Docker images (assuming no sharing of image layers) is roughly 4.5 GB, and the full virtual size of the EPG (including all layers) is about 630 MB. At 1Gbps, downloading the EPG image would take about 5 seconds and fetching all images uncompressed would take over 30 seconds.

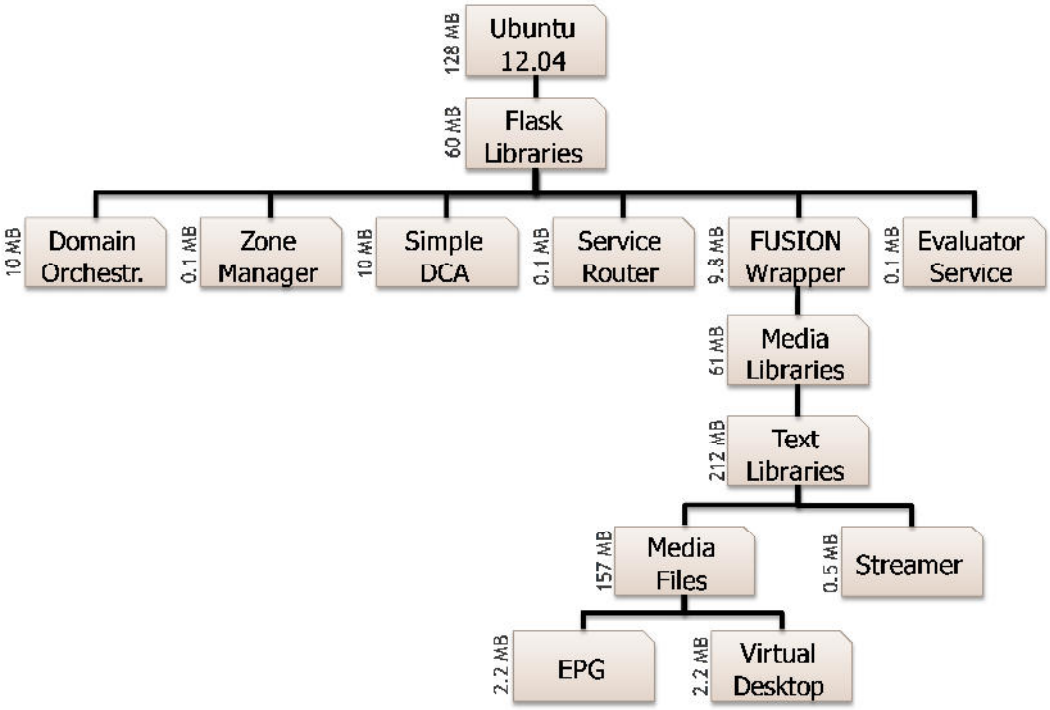


Figure 36 – Docker FUSION prototype containers with their incremental image size

Given the Docker tree depicted in Figure 36, and assuming the different image layers only need to be fetched once, then the total accumulated incremental size of all individual layers drops to only about 660 MB (compared to 4.5 GB), which is a factor 5 reduction in provisioning time and bandwidth:

[5] Our evaluation server is a HP DL380 G8, containing a dual-socket Xeon E5-2690v2, with 64 GiB RAM, running Ubuntu 14.04 and Docker 1.3.1.

fetching all layers uncompressed now would take about 7 seconds. Docker however also compresses all image layers when storing them in the remote registry, and uncompresses the layers after they are fetched at the local host. When inspecting the total size of the Docker registry, the actual size of all accumulated image layers of Figure 36 is further reduced by almost a factor of 2x to only about 370 MB (compared to 660 MB). Fetching all these layers from all applications at 1Gbps would now take only 3 seconds.

For the EPG container, the number of layers (and thus the amount of data) that needs to be fetched heavily depends on the presence of other shared layers in a particular execution zone. For example, in case only the Python/Flask image layer would available, then about 440 MB of uncompressed image layers (or about 250 MB of compressed data) would need to be fetched from the remote registry, resulting in a provisioning delay of about 2 seconds. In case the Streamer container is already available, then only about 160 MB of uncompressed image layers would have to be fetched remotely, and in case the VDesk container is available, only 2 MB of uncompressed data would need to be fetched remotely.

Comparing this with classical VM images, where each VM is stored and fetched individually, the overall provisioning delay and total required bandwidth can be significantly reduced. Although overlay-based mechanisms have been proposed for VMs, starting with a common base VM image and only providing the delta, Docker provides a much more elegant and fine-grained solution for efficiently sharing common files across possibly unrelated containers. Also, as Docker containers do not have to contain a full virtual environment, including OS, drivers, etc., the average size of a Docker container is also typically much smaller than a corresponding VM image.

Apart from the back-of-the-envelope calculations above regarding provisioning latencies and bandwidth, we also performed a number of simple performance experiments of the Docker registry component as well as the *docker pull* command for explicitly fetching/provisioning a Docker container (and all its layers) on a local host. Please keep in mind that Docker automatically detects which layers are not locally available yet and need to be fetched from a designated local or remote registry.

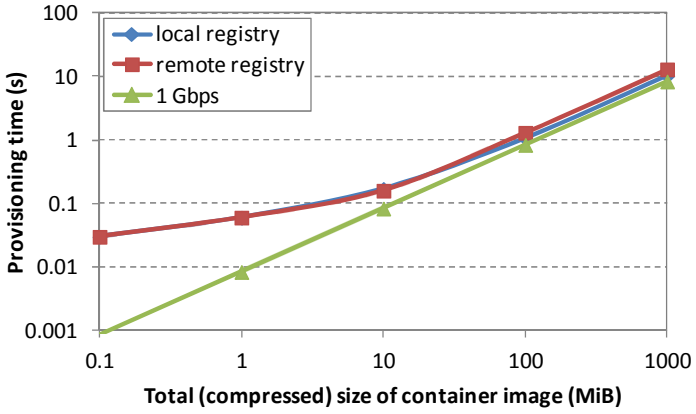


Figure 37 – Docker provisioning time in function of container image size

These results are depicted in Figure 37, where we show the time for running *docker pull* from a local and remote registry, compared to ideal 1Gbps. For the smallest image sizes, there is some additional overhead for simply checking the registry, setting up the connection and setting up the necessary local environment. For larger image sizes, the provisioning time approaches the link capacity.

4.3 Potential of a heterogeneous cloud

This section describes some fundamental experimental results regarding efficiently deploying real-time demanding media applications in a heterogeneous virtualized environment. Through a series of specific experiments, we illustrate the importance of a heterogeneous (SW/HW) cloud environment

for such demanding time-sensitive applications, where the platform (i) adapts itself based on the application requirements and infrastructure capabilities, and (ii) provides additional hardware infrastructure beyond the classic general purpose server blades for higher efficiency (e.g., micro-servers, accelerators, etc.)

4.3.1 Experimental setup

For the experiments described below, we used a SuperMicro server blade, containing a dual AMD Opteron 6174 with at least 64 GiB of RAM installed. For the hardware acceleration experiments, we used two custom scalable video encoder PCIe boards. The machine has 24 cores and 4 NUMA nodes with a high-speed HyperTransport interconnect in between the different processors. A diagram of the hardware configuration is shown in Figure 38.

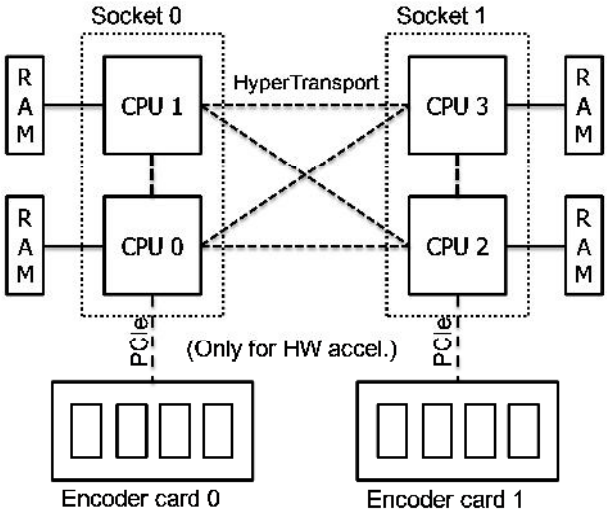


Figure 38 – Diagram of the hardware configuration

We ran the software experiments on an Ubuntu 64-bit distribution, using a wide range of versions of the Linux kernel to monitor the evolution of Linux and KVM, a Linux-based open-source hypervisor, w.r.t. performance and scalability. Unless otherwise stated, the results shown are based on the 3.2.1 baseline kernel. For the hardware accelerator experiments, we used a CentOS 6.3 running a 3.7.1 Linux kernel. To measure the impact of virtualization and isolation, we evaluated the efficiency of KVM-based virtualization as well as light-weight container-based isolation with a bare metal approach. For the latter, we used an in-house developed implementation based on the vanilla cgroups, namespaces and chroot capabilities in Linux, but including a custom life-cycle and package management layer. To capture the low-level performance statistics, we use our Perfex-MT performance counters monitoring tool [VDPU11].

The test applications consist of a number of common real-time media transformations, ranging from a simple pixel copying microbenchmark to more advanced image processing routines as well as video transcoding microbenchmarks, to measure the impact across a range of media transformations. Unless otherwise stated, the results shown are for a media format conversion microbenchmark.

4.3.2 Evaluation results

In this section, we describe a series of experiments, demonstrating the potential impact of a heterogeneous software and hardware cloud platform on performance and QoS.

4.3.2.1 Impact of NUMA

For scalability reasons, SMP architectures typically use non-uniform memory access (NUMA) memory architectures[LAME06], where memory access time depends on the memory location relative to the processor. As a result, NUMA architectures partition the memory system in multiple memory nodes,

each with their own access time, capacity and maximum memory bandwidth. In case the system has four NUMA nodes and all applications running on the system all happen to access memory from just one of these NUMA nodes, only a fraction of the available memory capacity and bandwidth is used, potentially resulting in large performance losses. The impact of NUMA has already been studied intensively, focusing on topics like placement of parallel applications across NUMA systems [BREC93], load balancing and scheduling strategies [CORR05][RAO13], memory management of VMs [RAO10], on-demand memory migration [MISH13], and other NUMA-related performance optimizations [MCCU10][TANG13].

Figure 39 shows the impact of running *N* instances of the media format conversion benchmark under a number of specific NUMA configurations. In this test, we pin each application process and their corresponding memory on a specific NUMA node. The application processes are uniformly distributed across all NUMA nodes (so each processor has the same workload); their corresponding memory is pinned on a particular NUMA node, depending on the test case. The vertical axis shows the average execution delay for performing a single application iteration. As long as the system is not saturated, the average latency, even when running dozens of instances in parallel, remains in the few milliseconds range. However, when the system reaches its saturation point (in this case caused by saturated memory bandwidth), then the average execution latency increases dramatically, crippling the entire system and all its applications.

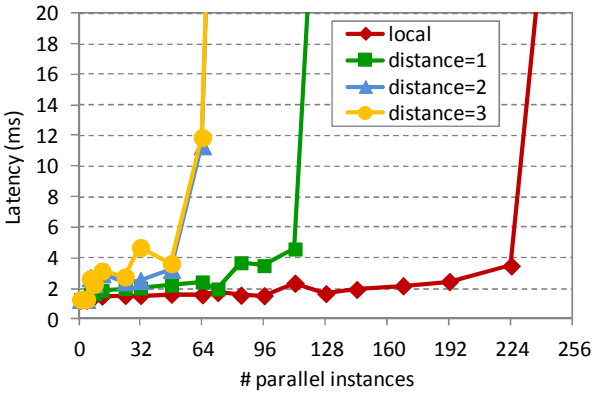


Figure 39 – Impact of local and remote NUMA-placement on overall scalability

Obviously, when the memory is pinned on the same NUMA node as the application process (i.e., local), then overall throughput is maximized. When application process and memory are on different nodes, overall throughput decreases by a factor of 2 or 4, depending on the distance between execution and memory node. A distance of 1 means accessing memory from another NUMA node on the same socket; a distance of 2 or 3 indicates accessing memory from a NUMA node on another CPU socket (straight or cross, respectively). In case of more complex NUMA configurations [ALMA12], or different interconnects, the impact is likely to increase. Obviously, when all applications access memory from fewer NUMA nodes, the aggregated available memory bandwidth is reduced proportionally, with comparable impact on scalability and throughput. In cloud environments where applications come and go, and are potentially moved across NUMA nodes as to rebalance CPU load, this scenario is very likely to occur, as illustrated several times in subsequent graphs.

4.3.2.2 Impact of virtualization

In a second series of experiments, we evaluated the impact of different virtualization strategies. A short description of each virtualization strategy is provided in Table 8.

Table 8 – Virtualization strategies used in experiments

bare	Bare metal (no virtualization or isolation).
nooks	Light-weight virtualization (1 container per application).
kvm	One KVM-based VM guest per application (1 vcpu).
kvmbare	One KVM-based VM guest for all applications (24 vcpus); All applications run directly in the same VM.
kvmnooks	One KVM-based VM guest for all applications (24 vcpus); Light-weight virtualization inside the VM (1 container per appl).
***.numa**	Optimal NUMA-awareness manually enabled.

The impact of different types of virtualization combined with manual NUMA awareness on the overall throughput is depicted in Figure 40. The left graph shows out-of-the-box results, leaving NUMA node selection of the application process and memory up to the operating system. The right graph shows the results when manually pinning each application and its memory on a particular NUMA node in an optimal manner.

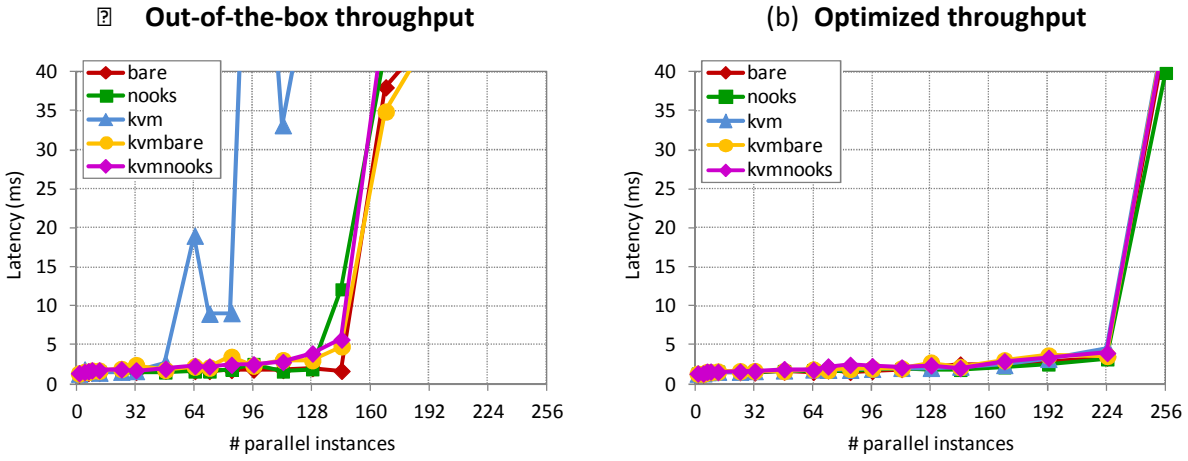


Figure 40 – Impact of virtualization and NUMA-awareness on scalability. The left graph shows out-of-the-box performance; the right graph shows the NUMA-aware optimized throughput results.

A number of observations can be made from these results. First, there is a clear difference in behaviour and scalability between the out-of-the-box and the NUMA-aware performance. Out-of-the-box performance is significantly worse compared to the bare metal NUMA-aware results. Second, when enabling the manual NUMA-aware placement, there is little or no difference in scalability between the tested virtualization options: even when running over 200 KVM-based VMs on the same machine, each containing an actively running media application, the average aggregated throughput remains on par with the bare metal results, albeit with higher variance (see further). Third, for the out-of-the-box results, the situation is very different. Running each application in its own VM (i.e., 'kvm') clearly has a more significant impact on scalability than the other virtualization strategies, with up to 400% reduction in scalability compared to an optimized KVM-based deployment on the same system. On the other hand, the container-based approach ('nooks') seem to have little or no impact on the overall behaviour and throughput of the system compared to bare metal for this test.

Next, in Figure 41, we show the amount of jitter within these applications under the same conditions. We express the amount of jitter by measuring the Coefficient of Variation (CoV) of the execution latencies of all application iterations, which is the ratio of the standard deviation (σ) to the mean (μ).

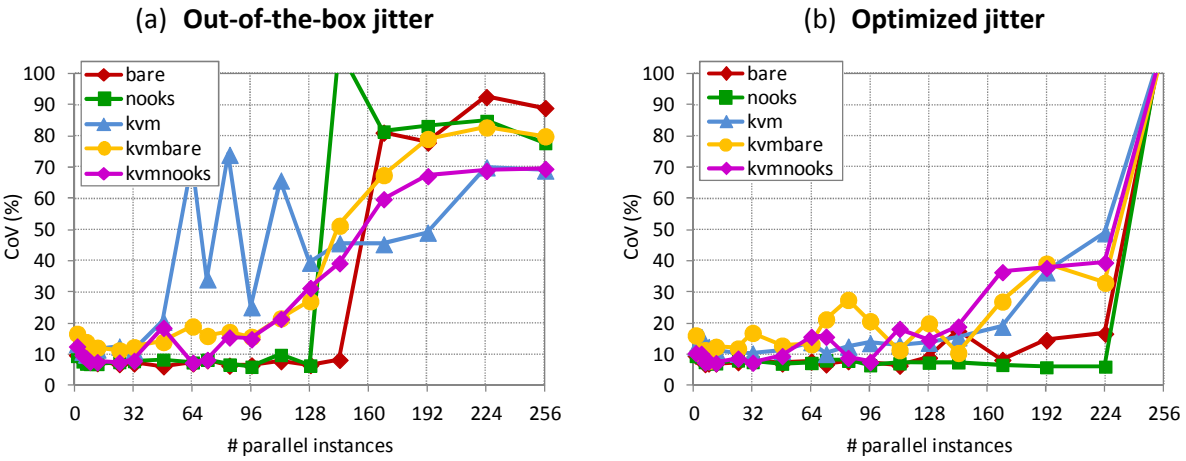


Figure 41 – Impact of virtualization and NUMA-awareness on the amount of variation in application iteration execution delays. The left graph shows the out-of-the-box results, whereas the right graph shows the NUMA-aware optimized jitter results.

The lower the CoV, the less jitter there is between the execution delays of subsequent iterations of the application. Lower jitter results in higher predictability and stability. As can be observed, the graphs show similar results as before, namely that NUMA-awareness clearly has a noticeably positive impact on the overall predictability and stability of the system. The amount of variation of the KVM-based strategies however remain higher than the container-based and bare metal approaches, whereas the container-based virtualization appears to result even in slightly lower jitter than bare metal. In the out-of-the-box scenarios, the KVM-based results show worse jitter, whereas the container-based approach resembles the bare-metal results. Where multimedia is concerned, increasing jitter directly impacts overall scalability. Two other key application-level performance indicators for real-time media are the amount of missed deadlines and the sustained frame rate. The impact of virtualization and NUMA awareness on the former is depicted in Figure 42. The same trends can be observed, whereby, in case of KVM, there are even some deadline misses at very low system load.

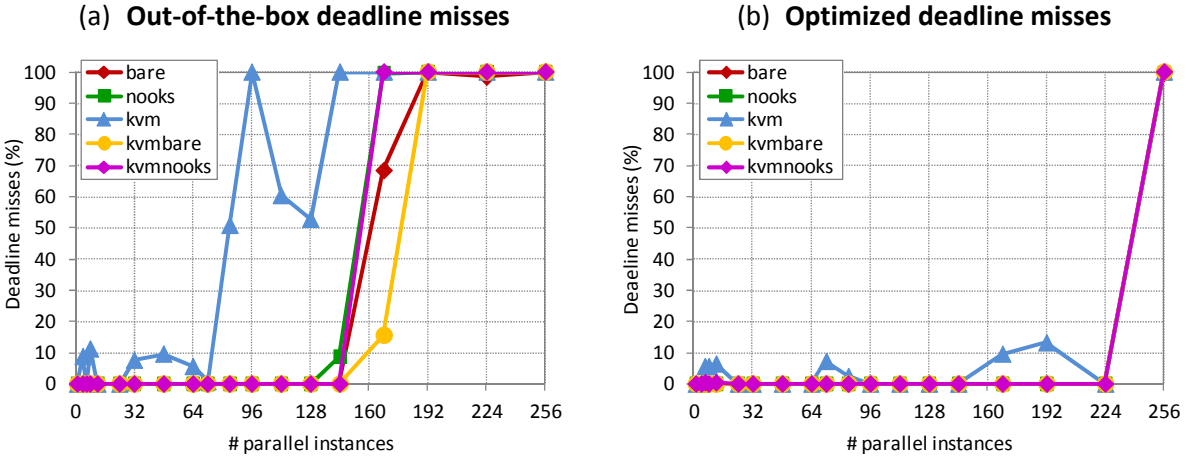


Figure 42 – Impact of virtualization and NUMA-awareness on the amount of missed application deadlines. The left graph shows the out-of-the-box results, whereas the right graph shows the NUMA-aware optimized results.

A fundamental question that arises from these experiments is how to detect or anticipate when the overall saturation point is reached on a particular system for a particular set of workloads.

Oversubscription obviously has a detrimental impact on all real-time applications on the same system that are also impacted by the same bottleneck.

4.3.2.3 Impact of other hardware features

There are several other hardware features of general purpose processors and systems that may have an impact on the overall throughput and predictability of a system. One of such features are huge memory pages. By default, memory is divided into pages of typically 4 KiB to enable efficient mapping of virtual memory addresses onto physical addresses. To avoid look-up of the physical address during each memory access, processors typically store the most recent translations in a TLB cache. Obviously, an application frequently accessing memory locations that are scattered across many pages will trigger many TLB misses, resulting in slower memory accesses and reduced performance. However, modern processors also support huge pages (e.g., 2 MiB) that can reduce the TLB miss penalty at the expense of coarse-grained memory paging.

The above experiments for our microbenchmarks with huge pages enabled did not seem to have a huge impact on throughput since most of these transformations have a rather straight-forward memory access pattern that does not involve scattering. We did see a reduction of up to a factor 5 lower average jitter for the most memory-intensive benchmarks. Other media transformations extensively using linked lists or for example randomly sample images may benefit from using huge memory pages.

Another important hardware feature is the dynamic frequency and voltage scaling in modern processors. To save power and energy (as well as cooling costs), it is beneficial to keep the CPU frequency and voltage levels as low as possible. Throttling up and down could significant impact overall throughput and predictability of applications. In our experiments, the overall throughput is not impacted, but jitter doubles when enabling on-demand frequency scaling by the operating system, and even more so in case of virtualization. Because CPU frequency is reduced up to a factor of 3 for an unsaturated system, the average processing latency is increased by the same factor. Finally, frequent changes in different clock frequencies also results in extra jitter and changing processing latencies.

4.3.2.4 Impact of software implementation

Obviously, a particular algorithm implementation can have a huge impact on the overall scalability. Since cloud abstracts the underlying hardware platform, it is hard to find or tune an algorithm for a particular cloud environment. Even small differences in the behaviour of different processors can significantly impact its actual performance when deployed on a particular hardware platform: different cache sizes, supported SIMD instruction sets, etc.

For example, an innocently looking divide instruction in an inner loop in one of our microbenchmarks made the application run twice as slow on the Opteron compared to the Xeon. Getting rid of this instruction also removed the performance difference. This example illustrates the difficulty to profile and monitor sensitive demanding applications for cloud environments, especially when lacking insight into the specifics of the environment. A heterogeneous cloud platform, directly or indirectly taking into account these issues can significantly improve the performance, efficiency and predictability, which is crucial for demanding time-sensitive applications such as real-time media applications.

4.3.2.5 Impact of hardware accelerators

In this section, we discuss the performance and cost benefits of deploying specialized hardware in a virtualized environment. We use two custom scalable video encoder PCIe boards, as shown in Figure 38 earlier. Each PCIe board contains four chips, allowing to encode up to 10 HD video streams in real-time per chip. Consequently, a single server blade with two boards can ideally encode up to 80 HD

streams in parallel, while freeing the CPU cores for doing other work, resulting in a dense high-performance energy-efficient system.

At present, the encoder boards do not support SR-IOV yet, so the amount of VMs that we will use will be limited to 4 in our tests (i.e., one per NUMA node), which are assigned to each VM via the PCIe pass-through capabilities in KVM. According to our calculations, adding this hardware accelerator (as well as other hardware accelerators like GPUs and others) can easily result in increases in the range of 5x up to 10x times in efficiency and up to 4x-8x times lower CAPEX costs per user, as well as significantly reduce the OPEX costs due to lower power consumption, reducing rack space, etc.

In this test, we evaluate the impact of NUMA-awareness and KVM-based virtualization on the overall scalability of using the hardware accelerators. Note that accessing these hardware accelerators is bandwidth demanding, as raw video frames need to be sent from system memory to the encoders over the PCIe busses.

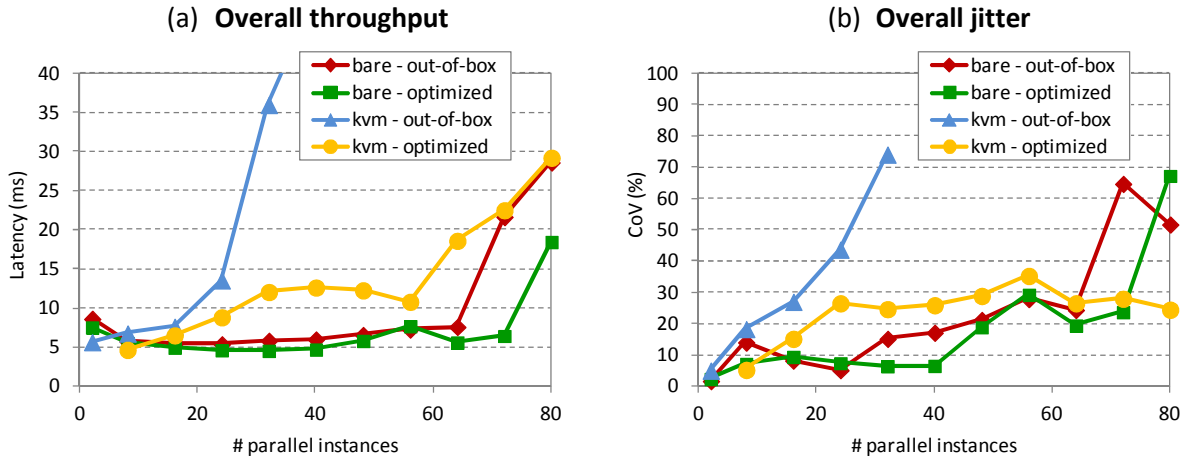


Figure 43 – Impact of virtualization and NUMA-awareness on the overall efficiency of exploiting hardware-accelerator video encoder boards. The left graph shows the average aggregated response time, whereas the right graph shows the overall jitter for encoding video frames.

As can be observed in Figure 43, both NUMA-awareness and virtualization impact the overall scalability. Observing the bare metal results, a modest impact of ~20% between out-of-the-box and NUMA-optimized performance is noted. For the NUMA-optimized results, we not only map particular application processes and their memory onto a particular NUMA node, we also use the hardware board that is directly connected to the corresponding CPU socket as well as tune the IRQ affinity to reduce overall jitter. The impact of the virtualization layer on the other hand is much higher, with up to 300% reduction in efficiency, as in the out-of-the-box scenario only about one third of the available hardware encoders can be used in real-time, even with moderate to high amounts of jitter.

When manually introducing NUMA-awareness, all available hardware encoders can be used in parallel in real-time, though there is still a noticeable impact in average response time as well as jitter. Analysis indicates this is at least partially caused by interrupt routing via the hypervisor to the VM [GORD12]. In summary, although the overall benefits of exploiting heterogeneous hardware in cloud environments can be huge, carelessly doing so can significantly reduce these benefits. Additional effort needs to be invested in finding techniques for optimizing the usage of these accelerators in cloud environments.

4.3.2.6 Impact of resource isolation and real-time guarantees

In this section, we first describe a methodology for mixing real-time and non-real-time applications for increasing the amount of performance isolation of real-time applications deployed onto the same resources while maintaining a high resource utilization ratio. Then, we present an initial set of results based on a number of experiments we have conducted. We focus mainly on the impact of various Linux primitives for providing better CPU performance isolation, and discuss some of the complexities regarding providing better isolation of memory and disk I/O performance in a Linux environment. All presented results have been measured on the same experimental setup as described in Section 4.3.1.

4.3.2.6.1 Methodology for mixing real-time and non-real-time applications

For some services, including many of the target applications services that FUSION should support, a best-effort approach w.r.t. resource utilization, as is typically provided in public clouds, is not enough. Some services fundamentally have particular resource requirements (either with respect to bandwidth, latency or timing constraints, or both).

The goal of our proposed methodology in this section is to investigate and provide stronger *performance isolation* mechanisms for applications that need them, but on the other hand also not to waste too many unused resources. In other words, we ideally want a flexible performance isolation layer, trading off perfect isolation with resource utilization efficiency. This is schematically represented in Figure 44.

Figure 44 – Flexible performance isolation mechanisms to increase overall resource efficiency

The yellow form symbolizes the actual resource utilization pattern of a RT application, and the blue box represents the isolation environment. For example, for NFV, operators now typically deploy only a single VNF or VM per cloud node to avoid interference or performance degradation of deploying multiple VNFs or VMs on the same node. Although this provides better isolation and predictability, this can be very inefficient.

As such, our goal is to find reduce the effective amount of isolation to what is represented as green in the Figure, and leverage the remaining blue area for running other (less sensitive) applications in the background. This is depicted more concretely in Figure 45.

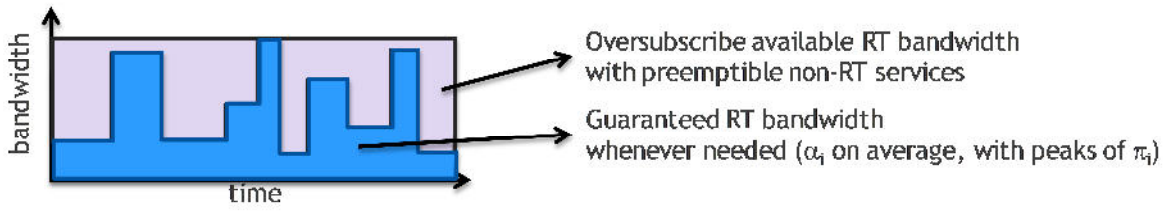


Figure 45 – Oversubscribing non-used RT bandwidth with non-RT applications for higher utilization

First, for each RT applications, we provide some soft bandwidth guarantees, for which we do not oversubscribe that reserved bandwidth for other RT applications. As a consequence, we also provide hard limits on how much bandwidth each RT application can use (i.e., to prevent oversubscription amongst RT applications).

Second, for many RT applications, the effective amount of resource bandwidth (e.g., CPU cycles, memory or network throughput, IOPS, etc.) typically will vary over time (e.g., based on the actual service load or active sessions. Rather than wasting those unused cycles, the main idea is to oversubscribe this available bandwidth with non-RT applications, but only when these non-RT applications can be easily and quickly preempted by the RT application, allowing the RT application to claim its requested amount of resources at any moment in time.

An overall system view of this approach is depicted in Figure 46. The available resource bandwidth is partitioned amongst the various RT applications, each with their own guaranteed amount resource bandwidth (with no oversubscription in between these RT applications); next to this, there may also still be some free bandwidth available for the non-RT best effort (BE) applications. As each of these RT applications will typically consume on average only some fraction of its resources, a particular amount of bandwidth remains per RT service to be used by the BE applications. In terms of oversubscription, the effective RT bandwidth is never oversubscribed (i.e., 1), whereas the available preemptible bandwidth and free bandwidth may be oversubscribed with different oversubscription factors (i.e., ω_1 and ω_2, to take into account that the preemptible bandwidth has a lower QoS score than the free bandwidth.

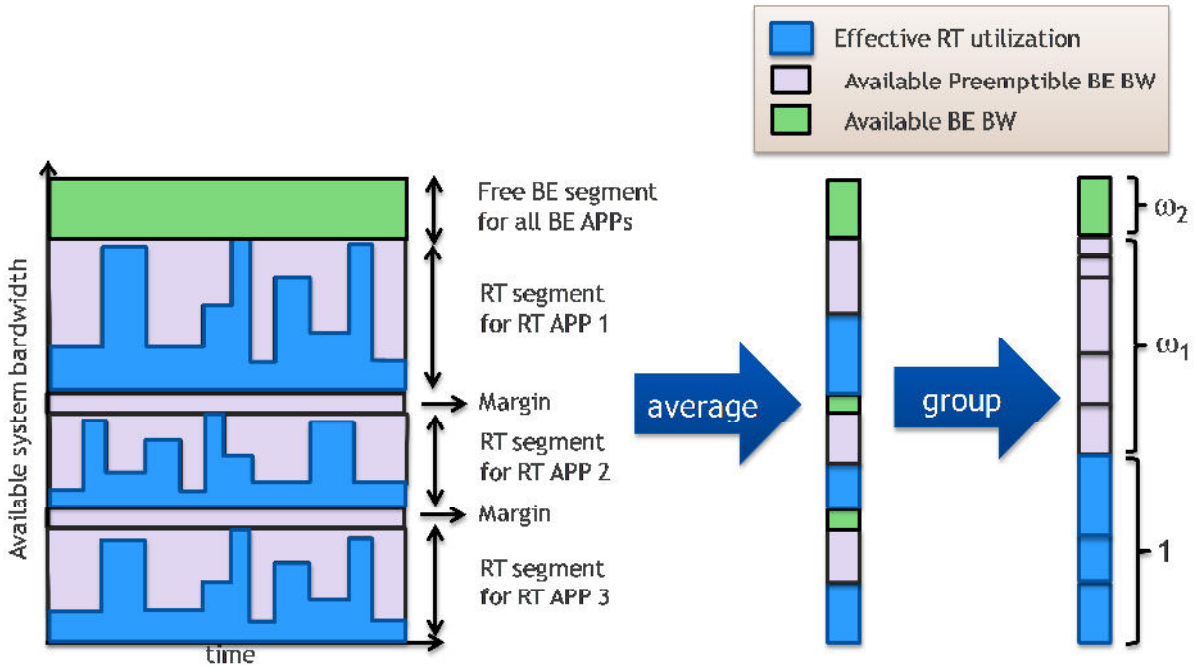


Figure 46 – System view of mixing RT and non-RT applications on a particular set of resources.

In the next few sections, we will evaluate the feasibility and effectiveness of a standard Linux kernel environment for implementing this methodology.

4.3.2.6.2 CCDF graphs

In the following sections, we will present the results using CCDF graphs, of which an example is shown in Figure 47. The graph shows the complementary cumulative distribution function of a particular application metric (in this case application latency). In other words, it represents the probability of particular outliers. In this case, it shows the probability that the latency is at least *X* ms. As such, it shows the tail latency (as a log-log curve) of an application deployed in a particular environment. In our experiments, we used a real-time media encoding application running at 25 FPS, meaning that the application latency for producing a new frame should remain (well) below 40 ms (see the dashed vertical line in the Figure); anything beyond that line result in missed deadlines and reduced QoE.

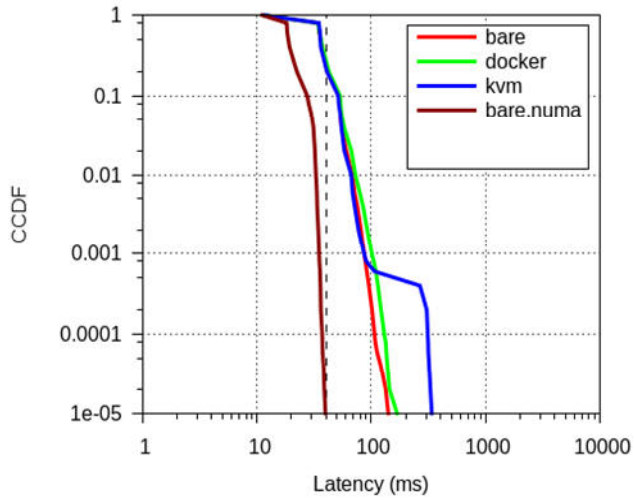


Figure 47 – Example CCDF plot of a real-time media encoding application

In this specific example, it can be observed that the tail latency of the bare metal NUMA optimized environment never goes beyond the 40 ms latency border, whereas the virtualized non-optimized deployment scenarios result in higher tail latencies, going beyond 100 ms once every 1000 frames (i.e., once every 40 seconds). In case of KVM-based deployment, the tail latency is higher than 200 ms once every 4000 frames (i.e., once every 3 minutes).

4.3.2.6.3 CPU performance isolation

In this series of experiments, we evaluate the feasibility of a vanilla Linux kernel environment for providing better-than-best-effort CPU performance isolation in a bare metal and virtualized environment. We do not rely on custom RT kernel patches or RT micro-kernels for achieving this CPU performance isolation.

Linux supports a number of core features for enabling better performance isolation between applications, including the following list:

- **Linux scheduling priority classes**

  The default scheduler is the Completely Fair Scheduler (CFS), which is a pre-emptive priority-based scheduler. The main idea of this scheduler type is that applications scheduled in the same priority class each get a fair share of the CPU resources (and thus can be pre-empted after some time to allow other applications to run). Applications in a higher (RT) priority class get priority over applications in a lower priority class, meaning that applications in the latter class will be pre-empted when an application of a higher priority is ready to run. Linux supports 100 priority levels, and within each priority level it supports 40 nice levels to give more or less relative CPU time (i.e., weight) to particular applications in a particular priority class. Default, user applications are typically started in the lowest (user) priority class. We call this the *best-effort* deployment scenario later.

  Note that recent Linux kernel versions (3.14+) now also support a constrained deadline scheduler based on periodicity instead of priorities. The experiments shown here date from before this scheduler was introduced in the vanilla kernel, and it would be interesting to run the same experiments again, using the deadline scheduler instead of the CFS scheduler.

- **CPU cgroups**

  One of the other enabling features are the cgroups, with allow for more fine-grained resource control, and is typically also used as part of lightweight containers for more fine-grained isolation, but can also be used separately. One of the cgroup resources that can be controlled are the CPU scheduling parameters. For non-RT applications, the *cgroups.cpu_cfs_period_us* and

cgroups.cpu_cfs_quota_us tuning parameters can be used for setting fine-grained CPU quotas for (groups of) applications.

For RT applications on the other hand, the *cgroups.rt_period_us* and *cgroups.rt_runtime_us* parameters can be used. Basically, these two parameters allow to specify how much CPU runtime a particular (set of) RT applications can use within each specified period of time (e.g., every second, or every 40 ms). The latter set of parameters allow to isolate the different RT applications, ensuring that no RT application can exceed its available CPU budget.

- **CPU sets**

  With this feature, one can constrain the CPU cores onto which particular applications, threads, containers or vCPUs can be scheduled on the physical host. This mechanism allows for some strict partitioning of applications across overlapping or non-overlapping CPU sockets and CPU cores.

For this test, we use a real-time video encoding application based on the X264 software encoding libraries. One application in this test encodes a 720p video stream at 25 FPS using the zero-latency settings. We deploy a number (*N*) of these applications on the experimental environment discussed in Section 4.3.1 and observe its encoding tail latency behaviour in a number of CCDF plots. We show the tail latencies of bare metal, Docker-based, KVM-based and NUMA-optimized deployment scenarios, and compare the CCDF graphs of best-effort-only deployments versus RT deployments.

For the RT deployments (either bare metal, Docker or KVM), we combined the RT scheduling priority settings with the maximum RT CPU utilization settings for giving priority to the RT applications while constraining how much CPU each RT application can use. Combined, they allow for a better performance isolation in between the RT applications while also allowing non-RT applications to consume the unused cycles by the RT applications. This is depicted in Figure 48.

Graph (a) in Figure 48 shows the baseline results of not enabling any RT priorities or CPU isolation, but simply running 48 instances of the real-time encoding application on the same physical host. The graph is in fact the same as we used earlier for explaining a CCDF plot. In graph (b), we deploy the same number of applications, but we deploy one of these applications as a RT application, and the CCDF plot represents the tail latency behavior of that RT application. It can be clearly observed that the enabled Linux mechanisms allow for a much better average and tail latency behavior across the various virtualization strategies, thus enabling a better performance isolation with respect of the other applications.

In graph (c), we go even one step further. Although the system was already saturated in graph (b), one can clearly see that saturating the system beyond reasonable amounts (i.e., deploying 120 instances) barely has any impact on the average and tail latency behavior of that single RT application, showing that the underlying mechanisms are not sensitive to the amount of load in the system. In graph (d), we then increase the number of RT applications to 24 under the same extremely high load. The CCDF plots shown here represents the accumulated latency results across all RT applications. Apart from a small bump in case of KVM (which we did not investigate further), the tail latency of these RT applications remains steady, even when the system is under heavy load, coming from other CPU-intensive applications.

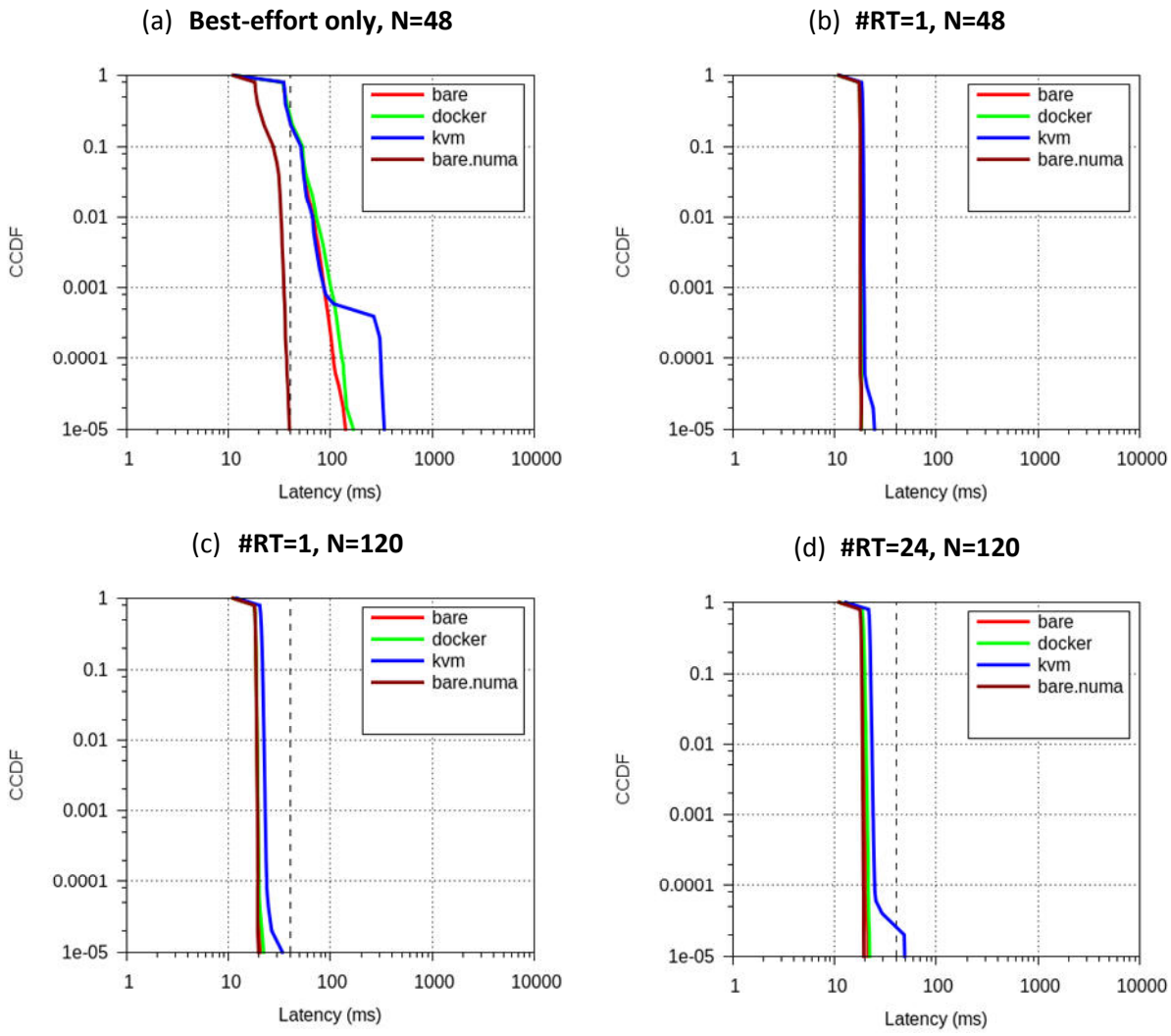


Figure 48 – CCDF latency plots of various deployment scenarios. Graph (a) shows the CCDF of 48 instances of a real-time video encoding application are deployed in a best-effort manner. In graphs (b)-(d), the CCDF plots the RT applications are shown, collocated with non-RT applications.

4.3.2.6.4 Memory throughput isolation

The previous section demonstrated the potential of existing Linux primitives for enabling a better performance isolation for CPU-intensive applications. In Figure 49, we show the tail latency of a memory-bound (CPU-intensive) application, namely a real-time media conversion application.

(a) **N=1** (b) **#RT=72, N=240**

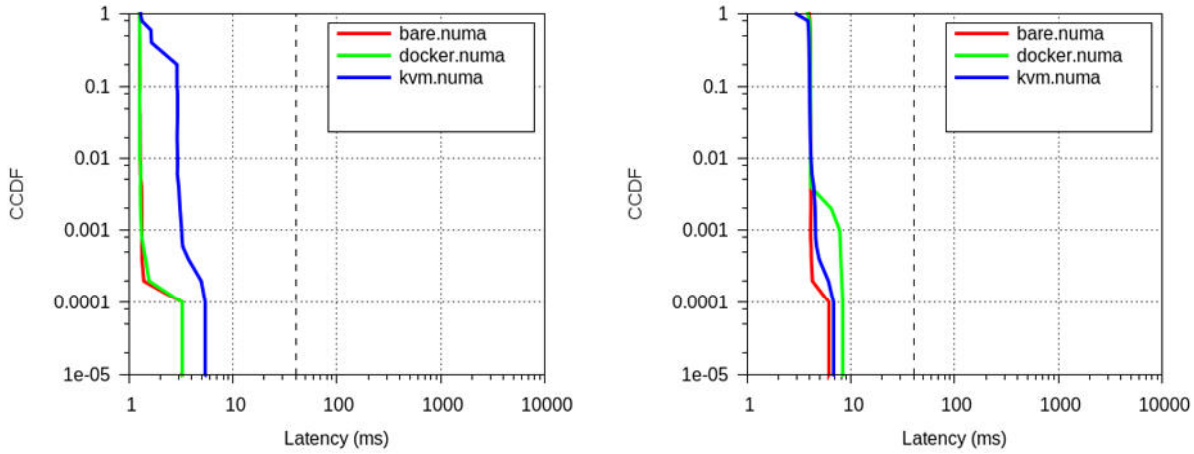


Figure 49 – CCDF latency plots a memory-bound RT media conversion application. The left graph shows the tail latency of a single application, whereas the right graphs shows the tail latency of all 72 RT applications, out of a total of 240 active media conversion applications.

As can be observed, although the tail latency of the RT applications in the right graph is still far under the 40 ms border (even when the system is fully saturated with 240 actively running applications), the average latency is a factor 3x worse compared to only running a single media conversion application. This is caused by the fact that all 240 applications contend for the same system memory bandwidth. So, although the RT applications do have priority over the non-RT applications, the memory controller is not aware of this difference in priority and simply gives each active application a fair share, effectively slowing down all applications, including the RT applications. This means that this reduced efficiency needs to be taken into account when assessing the amount of RT CPU cycles each RT application should minimally have. Vice versa, specific memory throughput isolation mechanisms could allow for better utilization of memory bandwidth (see further).

An even worse scenario is shown in Figure 50. There we run the RT encoding application of the previous section. Although this is a CPU-bound application, it does require a minimum amount of memory bandwidth to encode the frames in real-time. In the upper graph, the baseline behavior of running a single RT encoding application is depicted. Once every second, a new I-frame is created in this example, and the average encoding latency is about 20 ms.

However, when we deploy a very specific memory stressing application on the same machine, generating a huge burst of memory traffic once every 2 seconds for about half a second, the RT encoding application is clearly impacted, overshooting its 40ms deadlines during the memory bursts of the other application. Note that in the setup, the two applications are running on different cores, but share the same cache and memory subsystem. This example clearly shows that CPU isolation is not enough and additional preventing and correcting measures need to be taken, including better placement and selection of applications to be collocated on the same physical host, monitoring probes to detect such cases, etc.

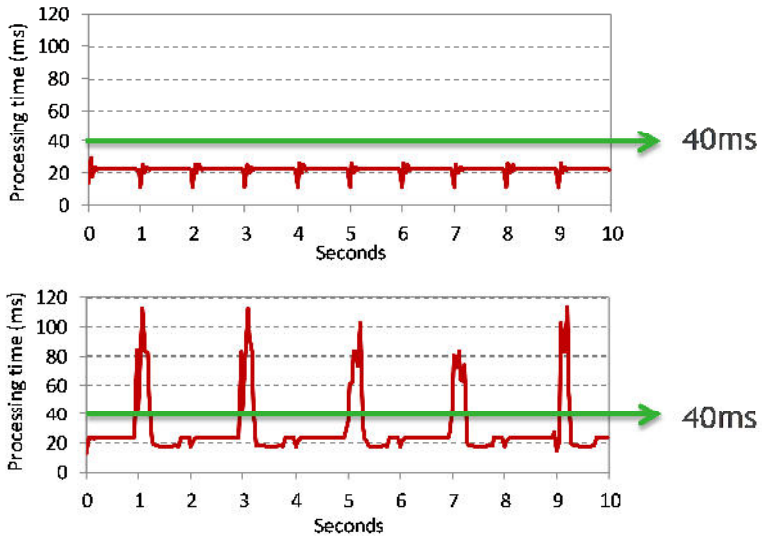


Figure 50 – Impact of an extreme periodic memory-intensive background stress application on the RT X264 encoding application. The upper graph shows the baseline behaviour of running only the encoding applications without the stresser.

However, with respect to memory throughput isolation, there currently are not many mechanisms available. Two coarser-grain mechanisms are to deploy particular applications on different memory subsystems. For example, Linux allows to restrict the memory NUMA nodes that applications can use, which can be done in a number of ways, including via the cgroups resource control or numactl. We are currently investigating other ways for providing more fine-grained memory throughput isolation.

4.3.2.6.5 Disk I/O throughput isolation

Apart from CPU and memory throughput isolation, we also investigated the capabilities of Linux for isolating disk or block I/O. Linux supports a number of block IO schedulers (typically either the CFQ or deadline scheduler), each having a number of tuning parameters for tweaking the priority and relative weight. For example, the CFQ IO scheduler has three main priority classes, each supporting 8 QoS priority levels: the RT priority class, the default best-effort priority class and the idle class, when the disk or block device can only be accessed when the device was idle for some grace period.

Apart from these IO schedulers, as well as a coarse-grained partitioning of applications across non-overlapping block I/O devices, the *cgroups.blkio* resource control settings also allow for fine-grained monitoring and throttling of the overall read and write throughput of an application to each individual block device, both in terms of kbps as well as IOPS, at the minimum granularity of one second. Note that these settings currently do not directly work for buffered writes, but can be tweaked by also setting an overall memory capacity for each application (or group of applications).

As different types of block devices have different physical runtime characteristics (e.g., the spinning disks of regular HDDs compared to the flash-based SSD disks or compared to a networked storage system), each block device should be tweaked separately for proper operation. Similarly, from a performance isolation perspective, the effective isolation behaviour and optimization strategies will vary w.r.t. the device. The results presented here are for a regular local HDD disk.

For the initial disk performance isolation analysis, we leveraged the open-source *fio* benchmarking tool [FIO14], which is a flexible benchmarking tool to measure the performance of a particular I/O device. It allows choosing different access patterns (e.g., sequential/random read/writes), block sizes, buffering, synchronous versus asynchronous I/O handling, I/O depths, etc., and has many configurations knobs for evaluating a particular pattern.

In this Deliverable, we only show a few example graphs, depicting the overall impact of RT I/O scheduling for a regular HDD disk. Different access patterns and devices however can result in different behaviour. As such, in a heterogeneous environment, this should be measured separately in order to have a clear understanding what the impact of a particular optimization in the respective environment.

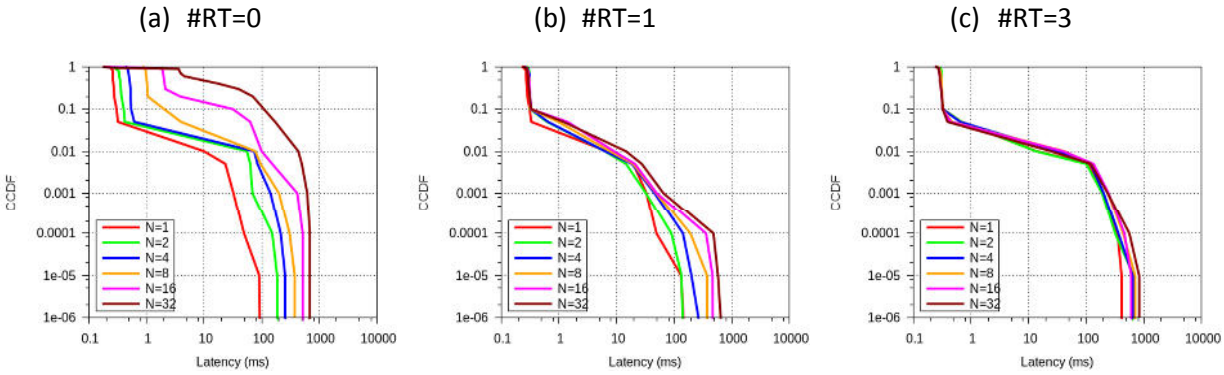


Figure 51 – Impact of RT I/O scheduling class on tail latency when having N applications doing sequential writes to the same physical HDD.

In Figure 51, the impact of enabling the RT I/O scheduling class is depicted on the average and tail latency behaviour of the best-effort and RT applications. Graph (a) depicts the overall tail latency in case there are N best-effort applications (and no RT applications). As can be observed, as N increases, the tail latency also increases accordingly. In graphs (b) and (c) the tail latencies are depicted of the RT application(s) as they are collocated with non-RT applications that also perform the same I/O operations (though at non-RT priority). As can be observed, the upper part of the tail (i.e., the average latency and jitter) is clearly improved, with minimal impact of having one or more non-RT applications accessing the same physical disk. However, the bottom part of the tail latency is still as bad as having no I/O priority classes. This means that, although there is no complete performance isolation, the average behaviour is much better. This is also shown in Figure 52, where the average I/O bandwidth is depicted for a varying number of RT applications. Whereas the available disk I/O bandwidth falls off linearly with the number of parallel applications in case there are no RT applications, the available I/O bandwidth for the RT applications remains much higher, though not perfect, especially for higher throughputs that reach or surpass the maximum disk I/O bandwidth.

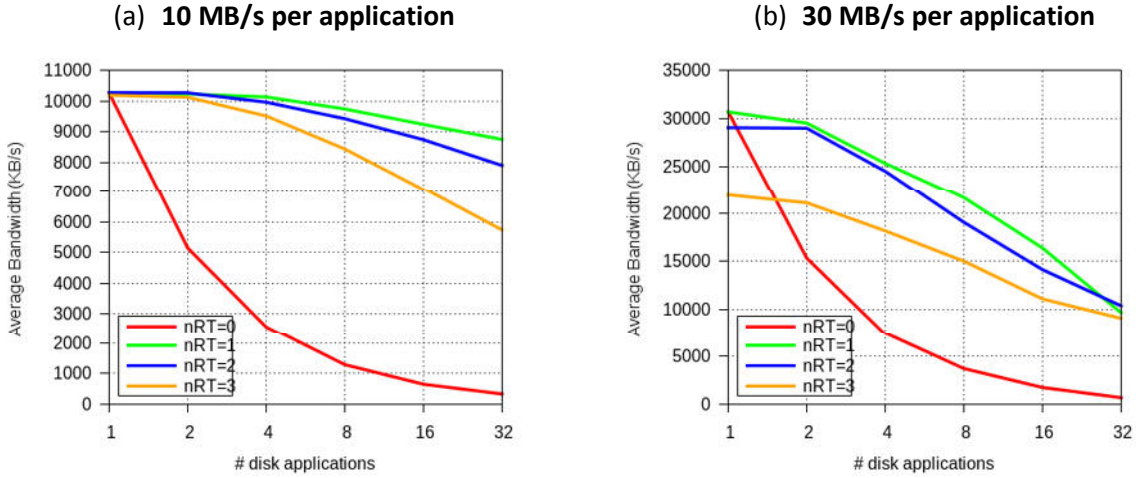


Figure 52 – Impact of RT I/O scheduling class and number of parallel I/O intensive applications on the average effective I/O bandwidth for a particular target I/O bandwidth.

4.3.3 Demonstrator

Next to the numerous experiments we performed, we also developed an initial demonstrator, to showcase these results in a live demonstrator setup and present them both internally and externally.

Specifically, we presented our demonstrator on heterogeneous clouds for demanding (FUSION/NFV) applications during the FutureX days in Bell Labs on October 15 and 16, both to people from various divisions within Alcatel-Lucent Antwerp, including the Bell Labs senior leadership team (SLT) and CEO who were visiting Antwerp at the time, as well as to external people, including operators, customers, universities as well as funding agencies. Overall, the demonstrator and our work in general was very well perceived as being very innovative and extremely relevant.

The demonstrator highlights a small subset of the various experiments related to heterogeneous cloud that we are currently undertaking. For this, we developed an HTML-based frontend to visualize the various metrics and to be able to dynamically interact with the setup, for example to change the number of applications, the virtualization technology, deployment configuration settings, etc. We also added a PHP backend to collect all live logging data and to interact with the actual cloud nodes running the applications.

A snapshot of the demonstrator is depicted in Figure 53. In our setup, we basically have three physical cloud nodes running in our lab. The first two cloud nodes consist of standard rack servers that are typically used in data centres. The last cloud node is an example of a micro-server, where we used an energy-efficient Atom server based CPU instead of the classic Xeon or Opteron CPUs. Being optimized for energy-efficiency, these micro-servers can run fewer workloads per core, but with typically much higher energy efficiency. Consequently, depending on the workload characteristics, it may be beneficial to leverage these Atom servers. Secondly, we also used lightweight containers for deploying the workload instead of KVM-based classical virtualization used in the two left-most cloud nodes. The key difference between the normal cloud node and the software-optimized cloud node is that in the latter, we optimized the way these VMs are being deployed and configured on the server, taking into account the application characteristics and the infrastructure capabilities, rather than leaving everything up to the underlying host and hypervisor.

In this particular demonstrator scenario shown in Figure 53, we show a memory-bound real-time video conversion benchmark, and we evaluate a number of key metrics, such as performance, predictability and efficiency metrics. As can be seen, by optimizing the way this application is being deployed on the same hardware, the performance can be improved by almost a factor of 2 on average, and up to a factor of 3 or 4 in extreme cases for this setup (not shown here), while still having a stable system instead of a completely saturated environment.

Comparing the software optimized and the hardware optimized cloud nodes, one can see that one needs more of these micro-server CPUs to be able to run the same workload. However, these micro-servers are much cheaper and can be packed much more densely in a rack, which (depending on the workload characteristics) may result (CAPEX-wise) in a cheaper and more dense solution.

Even more importantly is the energy efficiency with which these servers typically operate, which directly translates to overall energy consumption and cost (indeed, more than one third of all OPEX costs are related to energy and cooling [RAS11]), which subsequently may result in significant overall TCO reductions. As can be seen for this memory-intensive application, the compute energy efficiency (i.e., FPS per Watt) can be improved by a factor of 3 compared to the software optimized cloud node and a factor of 6 compared to the standard cloud node.

Note that this demonstrator currently only highlights some aspects and complexities related to optimally deploying demanding applications and how a heterogeneous SW/HW environment can result in drastic improvements with respect to QoS and TCO for a specific use case and environment. Current work involves automating this process so that *any* demanding workload can be optimally deployed on an *unknown* heterogeneous cloud infrastructure.

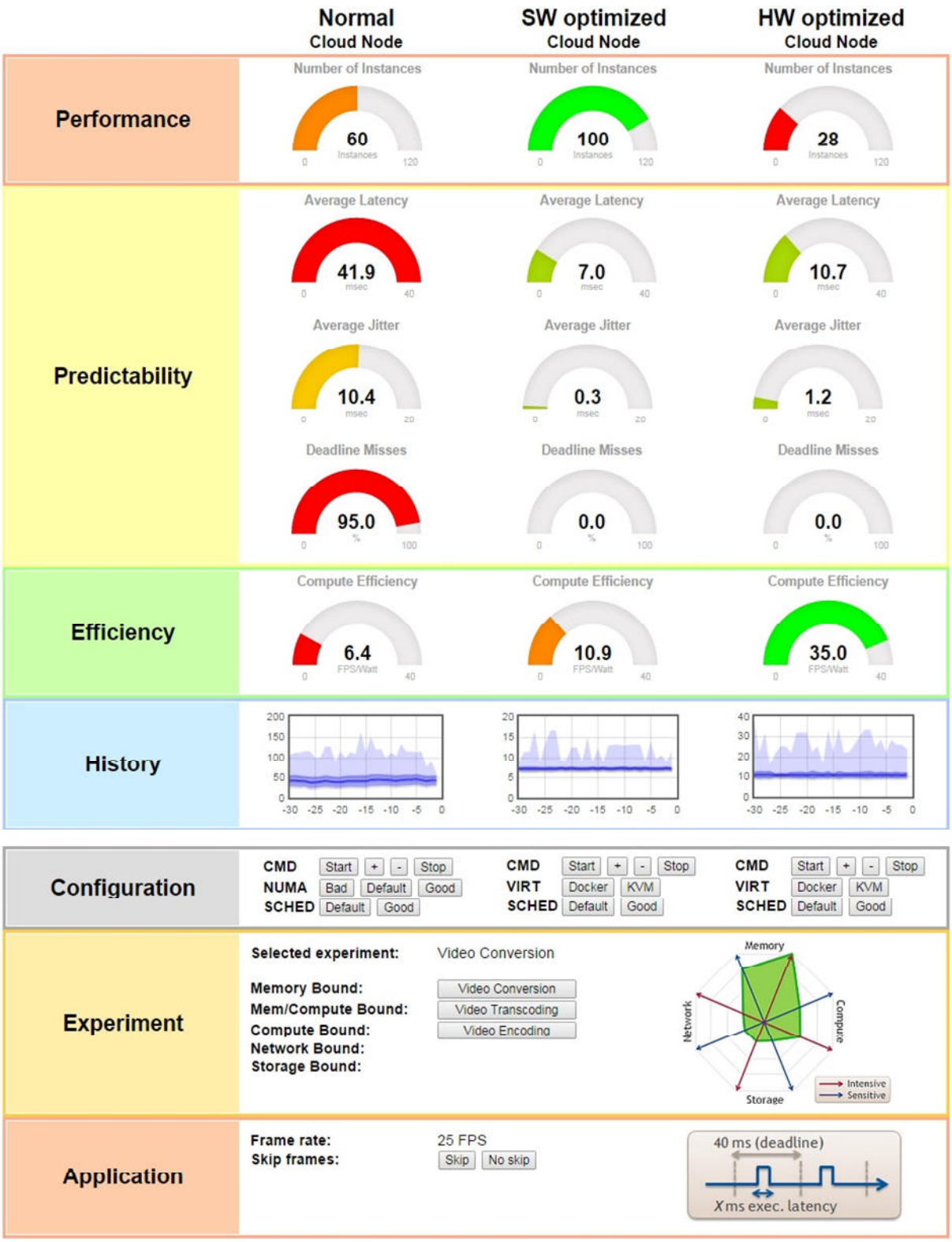


Figure 53 – Snapshot of the Bell Labs demonstrator GUI

4.4 Service placement

4.4.1 Experimental setup

The simulations use real data to infer the geographic location of users of a Massive Open Online Course discussion site. The model has several components. The overall aim is to create a simulation that includes the following elements:

- A realistic distribution of users across the surface of the earth.
- A realistic distribution of data centres on the world.
- An approximation of service deployment cost which is based on the Amazon EC2 cost.

In this modelling the network itself is abstracted away and replaced by assumptions drawn from measurement studies about the delays between latitude/longitude pairs. In general, the dataset includes 2508 data centres distributed in 656 cities on over the world. To map them into the FUSION architecture, we call each city to be an execution zone. We manually set the number of total available session slots so that they are enough to serve all user demands. The latency between users and execution zones are collected based on the Haversine distance, the shortest distance between two points around the planet's surface. We define the three latency thresholds R_{min} = 20 ms, R_{med} = 50 ms and R_{max} = 150 ms. The cost of service deployment at each execution zone is set proportionally to the cost of VM in Amazon EC2. All of these input dataset can be collected from https://github.com/richardclegg/multiuservideostream.

4.4.2 Preliminary evaluation results

4.4.2.1 Trade-off between the total utility and the deployment cost

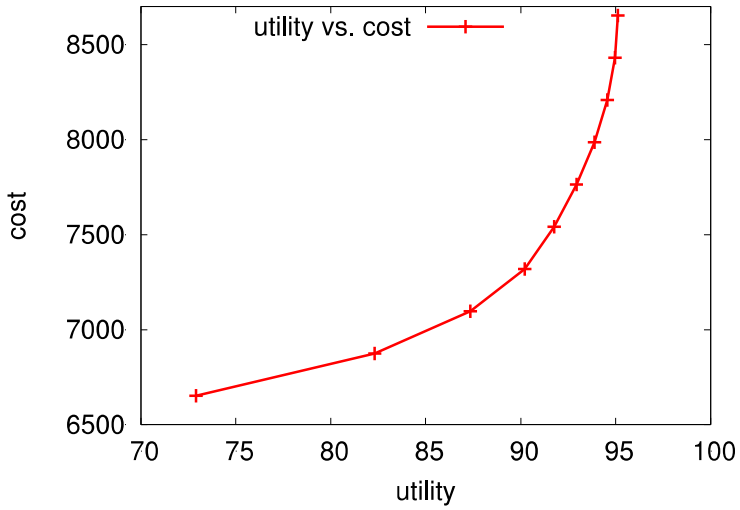


Figure 54 Total utility vs. deployment cost

The utility (x-axis) is computed in percentage in which 100% means all users can get their best QoE (latency is less than R_{min}). As shown in Figure 54, the best utility we can achieve is 95%, which means that approximately 95% of the users can have latency which is less or equal to R_{min} = 20 ms. From the Figure 54, with the budget (cost) of 8650 (units), we can get the maximum utility. On the other hand, with a limited budget of 6650 (units), 73% of users can get their best QoE. Using this graph, the service provider can see the trade-off between the user utility and the cost and then they can choose the best operational point.

4.4.2.2 Distribution of users' latency

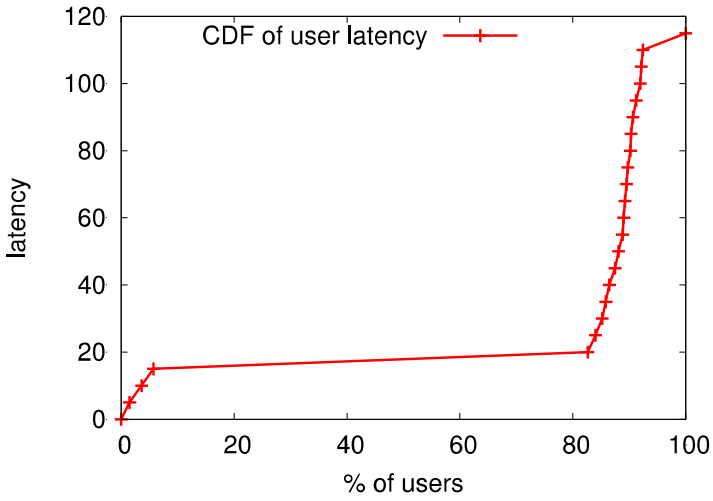


Figure 55 CDF of users' latency

We show in Figure 55 the CDF of users' latency when we have a cost budget 6650 (units) (as in Figure 54). Figure 55 presents more details on the distribution of users' QoE. There are 82% of the users can get the service in less than 20 ms. Only about 12% of the users suffer from high latency, which is larger than 50 ms. Since we have the max-min fairness constraint, even R_{max} = 150 ms, the worst users in the simulation have the latency which is less than 115 ms.

4.4.2.3 Benefit of max-min fairness

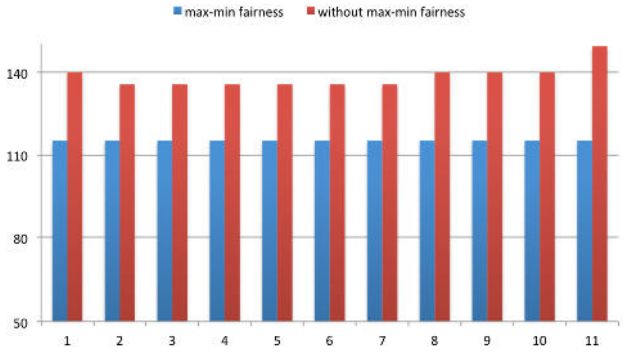
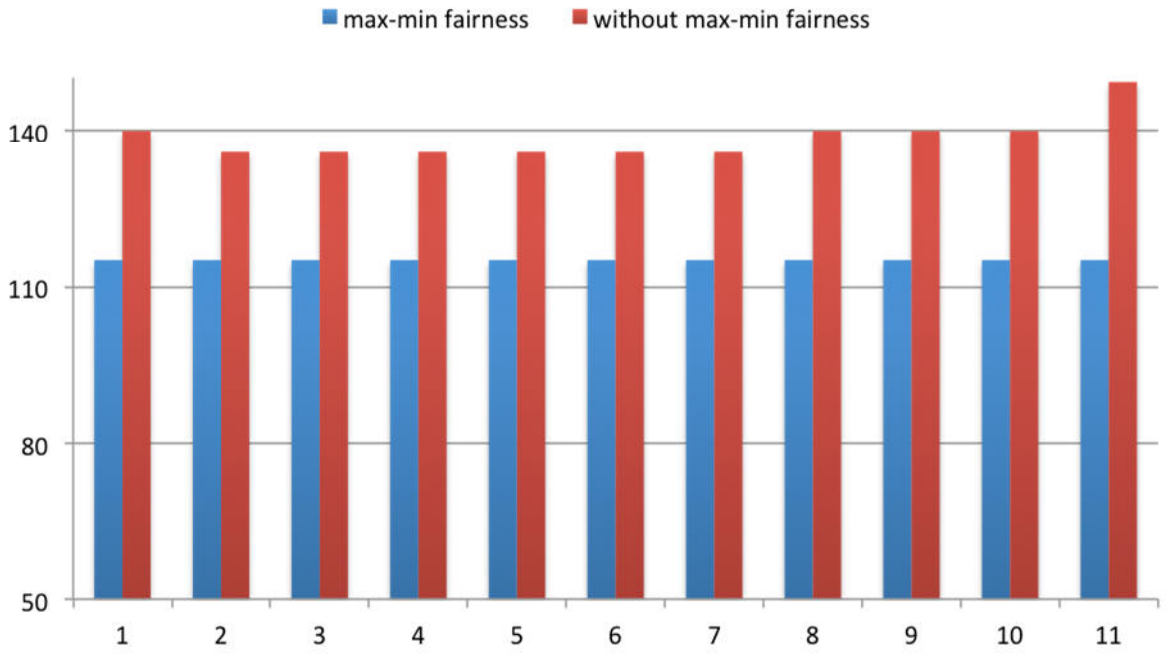


Figure 56 Latency with and without max-min fairness

Figure 56 shows the comparison between the case if we consider max-min fairness (step 1 of the algorithm) and the case we simply maximize the total utility (ignore step 1 in the algorithm). The x-axis is the interval of cost ranging from the maximum and the minimum cost, which correspond to each step in Figure 54. As shown in Figure 56, in all cases, with the max-min fairness constraint, the worst users still can get a better QoE (latency is less than 115 ms) compared to the case without

max-min fairness (maximum latency is equal to R_{max} = 150 ms).



4.5 Late binding measurements

Different networking technologies can be used for inter-VM communication from which some are discussed in the following paragraphs. The technology evaluation aims to indicate:

- the intricate configuration aspects of the different networking technologies;
- describe the data plane benefits each technology can offer;
- the configuration aspect of their use in inter-service communication.

The latter topic will indicate the diversity of the technologies and serves as an input for deciding the abstraction and specification level that a service manifest should deal with when specifying networking requirements of a service at design time.

4.5.1 SR-IOV Inter-VM communication

A technology description has been given in 3.7.3.1.1. In Figure 57, the SR-IOV throughput is shown for some of the possible interconnection realizations. Specifically, inter-VM communication using SR-IOV interfaces whereby the traffic between the VM is transported by the NetworkInterfaceCard is presented in the diagram.

The diagram compares SR-IOV–SR-IOV and SR-IOV–regular. SR-IOV regular indicates communication between one SR-IOV interface and the physical host interface. From the diagram it becomes clear that with increasing packet size, increasing throughput is obtained for SR-IOV – SR-IOV communications and therefore, larger packet sizes are preferred. The observed latency is typically below 1msec.

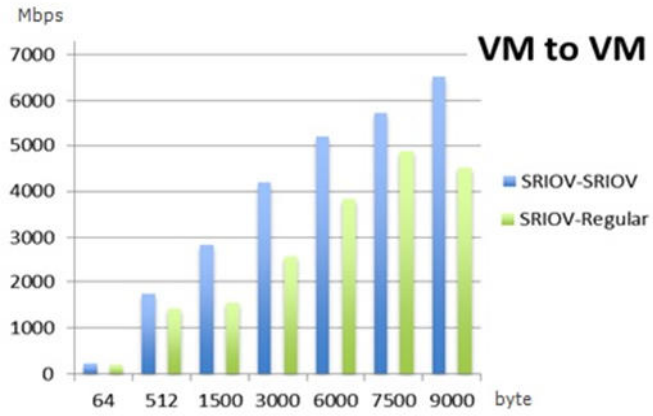


Figure 57 – SR-IOV throughput for different packet sizes

4.5.2 IVSHMEM inter-VM communication

A technology description has been given in 3.7.3.2.1. In [IVSH01], some performance measurements are described. The key take-aways described in this document is that IVSHMEM is an alternative for inter-VM, intra-host InterProcessCommunication mechanisms such as virtio+vhost. Specifically, the document mainly points out latency reduction that can be obtained using the IVSHMEM method and reports an order of magnitude lower latency and higher bandwidth in case of MPI communications.

From our own measurements, we measured an average latency of about 60 microseconds to notify data availability between sender and receiver via the doorbell mechanism (see Figure 58). Combined with the high memory bandwidth for copying data (typically 20-80 GB/s on modern servers), this allows for very high data throughput between VMs that are interconnected with shared memory, especially for large packet sizes.

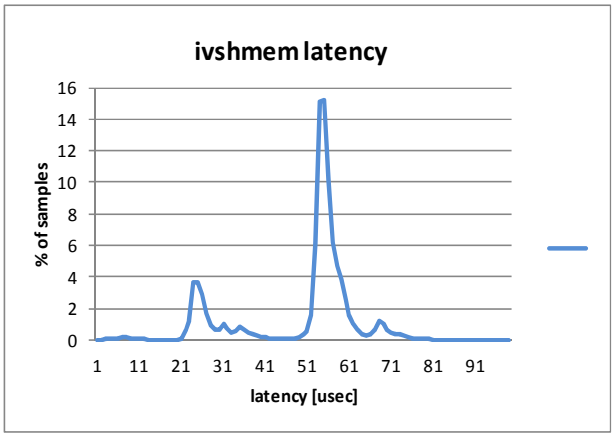


Figure 58 – IVSHMEM inter-VM communication latency

4.5.3 Dctcp

A technology description is given in 3.7.3.3.1. TCP and also DCTCP are negatively impacted by a window size that is smaller than BDP. In case of data centres, the network RTT between servers is below or around 1msec.

The performance of TCP, DCTCP and variants has to be evaluated against their design and purpose and can be roughly summarized by the measure of stability/variability of throughput over time.

Through simulation, the benefits of DCTCP vs TCP have been studied. The model simulates 5 regular TCP flows that are indicated with TCP “RENO” and 5 DCTCP flows. All flows transmit their data into a queue and use their respective algorithms when encountering congestion. The outcome of the

model allows to inspect the throughput of the individual flows and compare these against one another (for example, in equal share).

Mainly 2 payloads have been considered, notably:

- voice call packets: 240Byte/msg [CISC01]
- video frame packets: 1500Byte/msg

for a 10Gbps network link and in case of 10 simultaneous flows.

In Figure 59, some throughput results are shown for DCTCP for a payload of 240 bytes per packet on 10Gbps link with an RTT of 10 and 1msec.

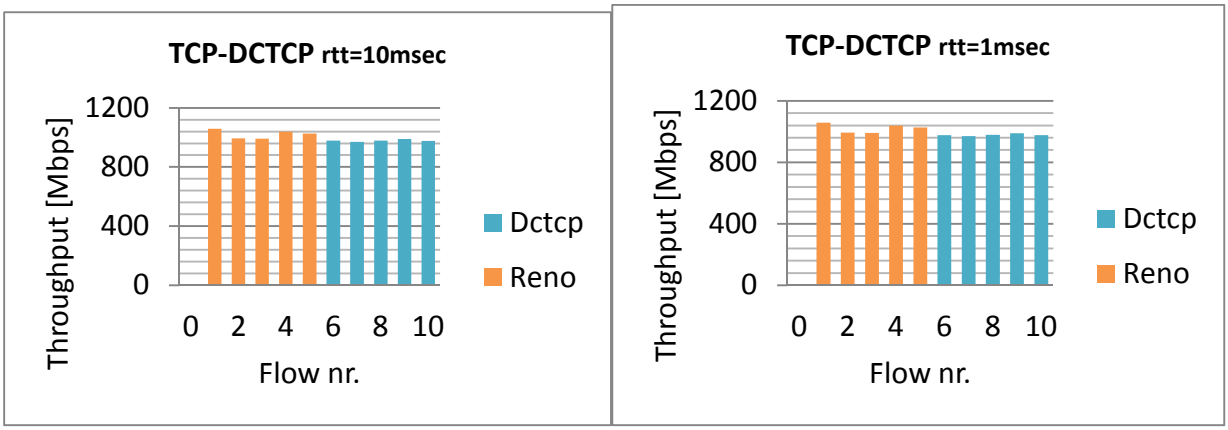


Figure 59 – DCTCP throughput results for packets of 240 bytes across different flows

From the results it is clear that:

- TCP regular flows show quite some throughput differences amongst the flows 1-5, ranging from 1252Mbps to 872Mbps in case of 10msec RTT indicating a deviation of ~25% from theoretical equal share scenario. In case the RTT is 1 msec, throughput ranges between 1135Mbps and 899Mbps (reduction of deviation to ~11% from equal share scenario).
- DCTCP provides a fair sharing in throughput over the different DCTCP flows.

On the same 10Gbps link, a payload of 1500bytes per packet was simulated with 10 simultaneous flows where 5 flows with regular TCP are modelled and 5 flows using DCTCP algorithm with 2 RTT values, 10 and 1msec. These results are depicted in Figure 60.

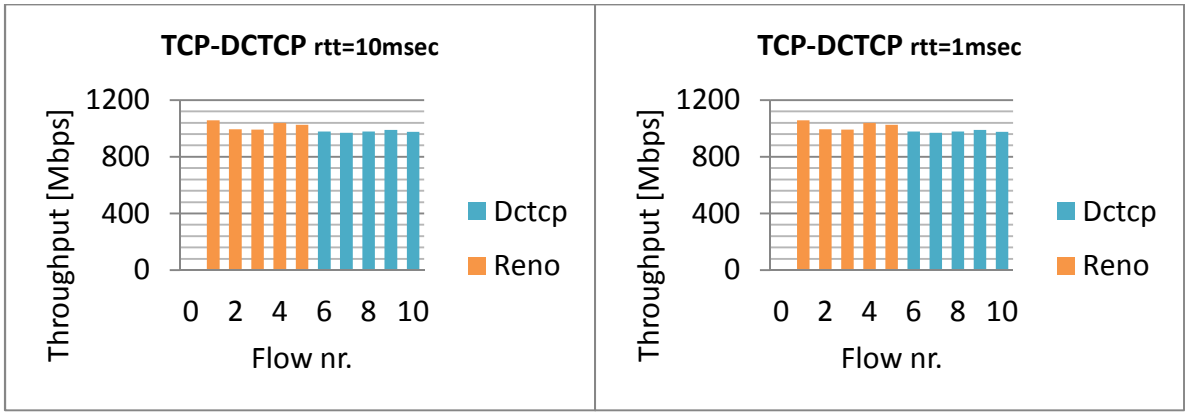


Figure 60 – DCTCP throughput results for packets of 1500 bytes across different flows

From the results it is clear that DCTCP gives overall more consistent fair share for each of the flows. The TCP flows show less spread, ranging between 1103 and 970Mbps in case of 10msec RTT (~7%

deviation from theoretical equal share scenario.) and 1057 to 990Mbps in case of 1msec RTT (~4% deviation from equal share scenario).

Round-trip time has impact on the TCP throughput variability whereby increasing RTT causes increasing throughput differences. RTT has little impact on DCTCP.

4.5.4 Docker libchan

A technology description is given in 3.7.3.4.1. Concerning libchan sessions established between containers on the same host and benefitting from zero copy channels, it is expected that this behaves alike shared memory communication.

Figure 61 shows initial inter-container throughput results over using libchan using regular TCP connections between two Docker containers (LXC – LXC communication), showing an on average throughput of 12.4Gbps. The initial uptake in throughput in the graph is due to TCP's slow start mechanism.

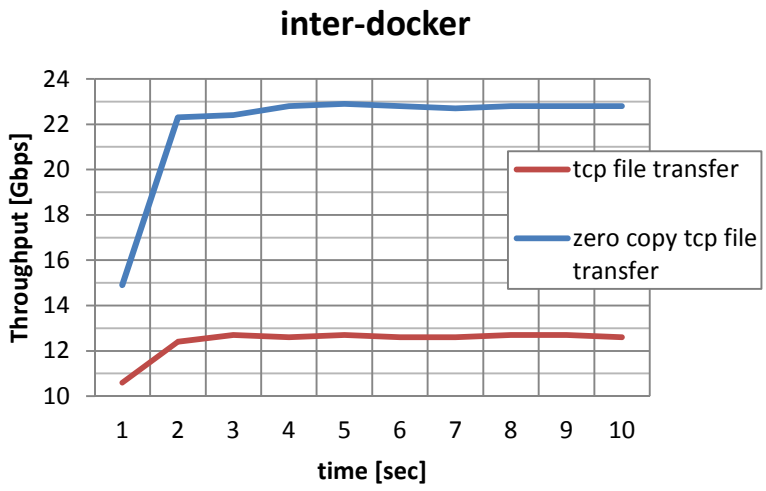


Figure 61 – libchan TCP throughput results over time

In case of using a 'zero copy' method while sending the data, the on average throughput increases to about 22.7Gbps.

When the test is repeated on directly connected hosts, we observe that the interconnect HW is the bottleneck for throughput (assuming 10Gbps interfaces). With respect to latency, the maximum latency was 30.8usec, whereas the minimum latency was 17.2usec.

4.5.5 RoCE

A technology description is given in 3.7.3.5.1. The following results were obtained by measuring latency and throughput between two servers with regular sender and receiver processes on a Linux OS. Four different RoCE working conditions and their corresponding throughput/latency have been measured:

- **Default**: out of the box configuration of RoCE communication, using RoCE specific communication methods (such as SDP).
- **Mtu 2048**: RoCE's MTU are subject to InfiniBand MTU restrictions. The RoCE's MTU values are, 256 byte, 512 byte, 1024 byte and 2K. The actual IB MTU cannot exceed the mlx4_en interface's MTU. Since the mlx4_en interface's MTU is 1560, it will run with MTU of 1K. Increasing interface MTU to 2K allows to select an RoCE MTU of 2048. Increasing the MTU causes greater efficiency and decreased processing of packets due to fewer packets for same throughput.

- **Tcp tuned**: This option refers to actual TCP use for communication in stead of SDP in default case. The tuning refers to the "Tuning the Network Adapter for Improved IPv4 Traffic Performance" chapter 3.8 in http://www.mellanox.com/related-docs/prod_software/Performance_Tuning_Guide_for_Mellanox_Network_Adapters.pdf where parameters like TCP timestamps where disabled, TCP selective ACK enabled etc.
- **Interrupt moderated**: Interrupt moderation is used to decrease the frequency of network adapter interrupts to the CPU. The algorithm checks the transmission (Tx) and receive (Rx) packet rates and modifies the Rx interrupt moderation settings accordingly.

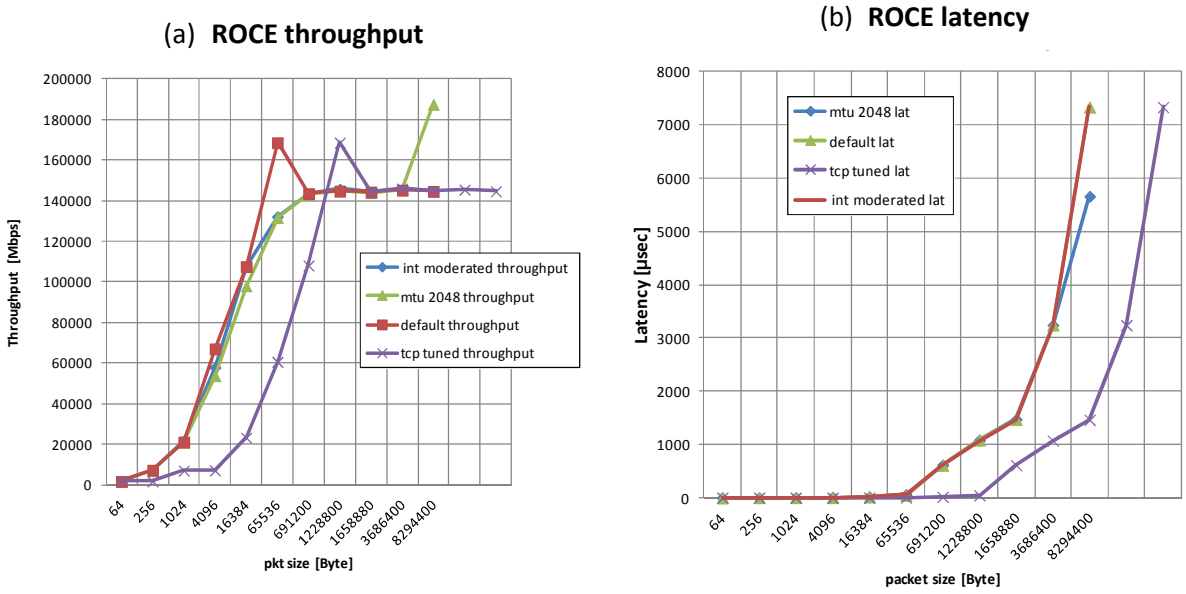


Figure 62 – Impact of packet size and network parameters on ROCE throughput and latency. The left graph shows throughput, whereas the right graph shows latency results.

All of the options show a maximum attainable throughput of ~145Gbps and latency of well below 100usec for packet sizes up to 65K. The tradeoff that needs to be made is either

- Selecting smaller packet size (<65KB) with ultra low latency (<100usec) but reduced throughput compared to max attainable or
- Selecting larger packet size (>65KB), allowing max throughput but at a cost of increased latency ranging between 1-7msec.

5. SUMMARY

In this deliverable, we provided an update on the FUSION service orchestration and management layers. We described a number of patterns for distributed service orchestration as well as enumerated a set of challenges and opportunities for efficiently demanding services on heterogeneous environments.

Second, we introduced a new concept in FUSION called multi-configuration service instances, separating service component instances from FUSION service configurations and corresponding session slots, enabling key benefits such as (i) increased reuse of resources and component instances, (ii) controlled session-based resource oversubscription and (iii) virtual elastic scaling across existing component instances.

Next, we provided an update on the key FUSION functions such as monitoring, lifecycle management, service placement and service scaling, describing new algorithms and corresponding evaluations. Based on this, we updated the design of the key FUSION service management layers and described the status of their initial implementation. Additionally, we also detailed on the various service management and communication protocols.

Finally, we described the results of a number of specific experiments regarding particular enabling technologies and algorithms, such as the potential impact of a heterogeneous cloud environment and the impact of particular service placement algorithms.

In the last year of the project, we will focus on incorporating more complex service and orchestration patterns, extend the initial prototype implementation, implementing more complex algorithms and scenarios and continue our work on heterogeneous cloud platforms for efficiently deploying demanding services. This will result in new concepts and algorithms, some of which will also be worked out and evaluated as part of the integrated demonstrator in WP5.

6. REFERENCES

[ALA09] N. Alam. "Survey On Hypervisors." School Of Informatics and Computing, Indiana University, Bloomington (2009).

[ALIZ11] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar, “Analysis of DCTCP: Stability, Convergence, and Fairness”, (2011), Department of Electrical Engineering, Stanford University{alizade, adelj, balaji}@stanford.edu,

[ALMA12] G. Almaless, F. Wajsburt, "On the scalability of image and signal processing parallel applications on emerging cc-NUMA many-cores, " Proc. Intl. Conf. on Design and Architectures for Signal and Image Processing (2012), pp.1-8

[BALA12] Balajee Vamanan, Jahangir Hasan, and T. N. Vijaykumar, “Deadline-Aware Datacenter TCP (D2TCP)”, SIGCOMM12, (2012)

[BALA98] N. Balakrishnan and C. R. Rao. Order Statistics: Theory and Methods, volume 16 of Handbook of Statistics. Elsevier, 1998.

[BONI10] M. Boniface, B. Nasser, J. Papay, S.C. Phillips, A. Servin, X. Yang, Z. Zlatev, S.V. Gogouvitis, G. Katsaros, K. Konstanteli, "Platform-as-a-service architecture for real-time quality of service management in clouds, " Intl. Conf. on Internet and Web Applications and Services (2010), pp. 155-160.

[BREC93] T. Brecht, "On the importance of parallel application placement in NUMA multiprocessors, " Proc. Intl. Symp. on Experiences with Distributed and Multiprocessor Systems (1993), pp. 1-18

[CEIL01] http://docs.openstack.org/developer/ceilometer/architecture.html

[CISC01] http://www.cisco.com/c/en/us/support/docs/voice/voice-quality/7934-bwidth consume.html

[CORB01] Common Object Request Broker Architecture, http://www.omg.org/gettingstarted/corbafaq.htm

[CORR05] M. Correa, R. Chanin, A. Sales, R. Scheer, A.F. Zorzo, "Multilevel Load Balancing in NUMA Computers, " Proc. Symposium on Operating Systems Principles (2005), pp. 1-9

[DCOM01] Distributed Component Object Model, http://technet.microsoft.com/en-us/library/cc958799.aspx

[DURA13] M. Duranton, D. Black-Schaffer, K. De Bosschere, J. Maebe, "The HIPEAC vision for advanced computing in horizon 2020, " HiPEAC High-Performance Embedded Architecture and Compilation (2013)

[ERLA01] erlang/OTP, a complete development environment for concurrent programming, http://www.erlang.org/doc/

[ETCD01] https://coreos.com/using-coreos/etcd/

[ETSI13] Network Function Virtualisation (NFV); Use cases, ETSI GS NFV 001 V1.1.1, 2013.

[FIO14] Fio I/O benchmarking tool, http://freecode.com/projects/fio.

[GEVE01] gevent, a coroutine-based network library for python, http://www.gevent.org/

[GORD12] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, D. Tsafrir, "ELI: bare-metal performance for I/O virtualization, " ACM SIGARCH Computer Architecture News (2012), pp. 411-422

[HEAT01] http://www.slideshare.net/mirantis/an-introduction-to-openstack-heat

[HEAT02] http://docs.openstack.org/developer/heat/

[HEAT03] https://blueprints.launchpad.net/heat/+spec/tosca-support

[HWAN14] Jaehyun Hwang, Joon Yoo and Nakjung Choi, “Deadline and Incast Aware TCP for cloud data center networks”, computer networks V68, AUG 5, 2014, p20-34.

[IVSH01] http://www.linux-kvm.org/wiki/images/c/cc/2011-forum-nahanni.v5.for.public.pdf

[KEYS01] http://docs.openstack.org/developer/keystone/architecture.html

[LAME06] C. Lameter, "Local and remote memory: Memory in a Linux/NUMA system, " Linux Symposium (2006)

[LEE10] Lee Bill, “RoCE, RDMA over converged ethernet”, 2010, http://www.ethernetalliance.org/wp-content/uploads/2011/10/document_files_RoCE_Intro_and_Update_100716.pdf

[LIBC01] https://github.com/docker/libcontainer/blob/master/PRINCIPLES.md#libcontainer-principles

[LIN08] Y.M. Lin, K.A. Jenkins, A. Valdes-Garcia, J.P. Small, D.B. Farmer, P. Avouris, "Operation of graphene transistors at gigahertz frequencies, " Nano Letters, 9:1, (2008), pp. 422-426.

[LPWIKI] Linear Programming. http://en.wikipedia.org/wiki/Linear_programming

[MALK12] D. Malko, C. Neiss, F. Vines, A. Gorling, "Focus: Graphyne May Be Better than Graphene", Phys. Rev. Letter, 108, (2012).

[MCCU10] C. McCurdy, J. Vetter, "Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms, " Proc. Intl. Symp. on Performance Analysis of Systems & Software (2010), pp. 87-96

[MISH13] V.K. Mishra, D.A. Mehta, "Performance enhancement of NUMA multiprocessor systems with on-demand memory migration, " Proc. Intl. Advance COmputing Conference (2013), pp. 40-43

[NOMS08] D. Carrera, M. Steinder, and J. Torres, “Utility-based Placement of Dynamic Web Applications with Fairness Goals”, in NOMS 2008.

[OFED01] https://www.openfabrics.org/index.php/ofed-for-linux-ofed-for-windows/ofed-overview.html

[OFED02] http://www.snia.org/sites/default/files2/SDC2013/presentations/Hardware/DavidDeming_IBA_Software_RDMA.pdf

[OPNFV] https://www.opnfv.org/

[PITT07] M. Pittau, A. Alimonda, S. Carta, A. Acquaviva, "Impact of task migration on streaming multimedia for embedded multiprocessors: A quantitative evaluation, " Proc. IEEE Workshop on Embedded Systems for Real-Time Multimedia (2007), pp. 59-64.

[RAS11] N. Rasmussen, “Determining Total Cost of Ownership for Data Center and Network Room Infrastructure”, white paper 6, APC Schneider Electric, http://www.apcmedia.com/salestools/CMRP-5T9PQG/CMRP-5T9PQG_R4_EN.pdf.

[RAO13] J. Rao, K. Wang, X. Zhou, C.Z. Xu, "Optimizing virtual machine scheduling in NUMA multicore systems, " Proc. Intl. Symp. on High-Performance Computing Architecture (2013), pp. 306-317

[SAT09] M. Satyanarayanan, et al. "The case for vm-based cloudlets in mobile computing." Pervasive Computing, IEEE 8.4 (2009): 14-23.

[RAO10] D.S. Rao, K. Schwan, Karsten, "vnuma-mgr: Managing vm memory on numa platforms", Proc. Intl. Conf. on High-Performance Computing (2010), pp. 1-10

[SHER02] T. Sherwood, E. Perelman, G. Hamerly, B. Calder, "Automatically characterizing large scale program behavior, " ACM SIGARCH Computer Architecture News, 30:5, (2002), pp. 45-57

[SKIN14] http://people.redhat.com/mskinner/rhug/q3.2014/cloud-init.pdf

[SRIO01] http://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html

[SRIO02] http://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html

[TANG13] L. Tang, J. Mars, X. Zhang, R. Hagmann, R. Hundt, E. Tune, "Optimizing Google's warehouse scale computers: The NUMA experience, " Proc. Intl. Symp. on High-Performance Computing Architecture (2013), pp. 188-197.

[TOSC01] https://www.oasis-open.org/committees/**tosca**/

[TOSC02] http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.html

[TOSC03] https://www.oasis-open.org/committees/document.php?document_id=54428&wg_abbrev=tosca

[TUR14] J. Turnbull. The Docker Book: Containerization is the new virtualization, 2014.

[VDPU11] F. Vandeputte, "Tools for Analyzing the Behavior and Performance of Parallel Applications, " Proc. Intl. Conf. on Parallel COmputing(2011), pp. 491-498

[VERM14] L. Vermoesen, Jean-Paul Belud, Emilio Lastra, Luc Ogena, Frederik Vandeputte, “On the Economical Benefit of Service Orchestration and Routing for Distributed Cloud Infrastructures: Quantifying the Value of Automation.”, White Paper, (2014), http://www.fusion-project.eu/

[Wiki] http://en.wikipedia.org/wiki/Latency_(audio)

[WIKI01] Bandwidth Delay Product, http://en.wikipedia.org/wiki/Bandwidth-delay_product

[WILS13] Christo Wilson, Hitesh Ballani , Thomas Karagiannis and Ant Rowstron , “Better Never than Late: Meeting Deadlines in Datacenter Networks”, SIGCOMM13, (2013)

7. APPENDIX A: PROTOCOL SPECIFICATIONS

This appendix provides an overview of all REST protocol specifications that we defined (and largely implemented) so far with respect to FUSION orchestration and management layers. It contains the protocols of the various orchestration layers (i.e., domain, zone, DCA) as well as the application and evaluator services APIs. The full specifications including their key parameters and message structure will be provided later in a separate reference document.

7.1 Domain orchestration protocols

This section covers all public domain orchestration protocols. We grouped these protocols based on the resource type they control.

7.1.1 Zone management protocols

Below a list of REST protocols for managing zones.

/1.0/zones	
GET	Returns a (filtered) list of all registered zones in this orchestration domain.
POST	Register a new zone in this domain (only can be done by domain admin and authorized zone admins).
DELETE	Remove all zones from this domain (e.g., in the context of a zone admin).

/1.0/zones/<zoneID>	
GET	Fetch detailed information of this zone (i.e., services, state, etc.).
DELETE	Remove this zone from this domain (terminating all running instances first).

/1.0/zones/<zoneID>/services	
GET	Returns a (filtered) list of all registered services in a specific zone in this orchestration domain.
POST	Register a new service in this zone.
DELETE	Remove all services from this zone (terminating all instances and unregistering them first).

7.1.2 Service management protocols

Below a list of REST protocols for managing services.

/1.0/services	
GET	Returns a (filtered) list of all registered services in this orchestration domain, along with their current state and session slot information. Filtering is done based on ACL as well as an optional filtering request parameter.
POST	Register a new service in this domain.
DELETE	Remove all services from this domain (e.g., in the context of a service provider).

/1.0/services/<serviceID>	
GET	Fetch detailed information of this service (i.e., manifest, runtime state, etc.).
DELETE	Remove this service from this domain (possibly terminating all running instances first).

/1.0/services/<serviceID>/state	
GET	Fetch global state information of this service in this domain.
PUT	Change global domain service lifecycle state (e.g., register, provision, deploy, terminate) in the entire domain.
DELETE	Remove this service from this domain (possibly terminating all running instances first).

/1.0/services/<serviceID>/slots	
GET	Fetch detailed information of the number of available and used session slots of this service in the entire domain.
PUT	Change the number of available session slots of this service in the entire domain.
DELETE	Remove all active slots from this service in this domain (i.e., terminate service instances).

/1.0/services/<serviceID>/zones	
GET	Returns a (filtered) list of all zones in this domain where this service is registered.
POST	Register this service in a new zone.
DELETE	Remove this service from all zones in this domain (terminating all instances and unregistering the service first in all respective zones).

7.1.3 Service zone management protocols

Below a list of REST protocols for managing services in a specific zone, and vice versa.

/1.0/services/<serviceID>/zones/<zoneID> /1.0/zones/<zoneID>/services/<serviceID>	
GET	Fetch detailed information of this service in this specific zone.
DELETE	Remove this service entirely from this zone in this domain (possibly terminating all running instances first).

/1.0/services/<serviceID>/zones/<zoneID>/state /1.0/zones/<zoneID>/services/<serviceID>/state	
GET	Fetch global state information of this service in this zone.
PUT	Change service lifecycle state of this service in this specific zone (e.g., register, provision, deploy, terminate).
DELETE	Remove this service from this zone (possibly terminating all running instances first).

/1.0/services/<serviceID>/zones/<zoneID>/slots /1.0/zones/<zoneID>/services/<serviceID>/slots	
GET	Fetch detailed information of the number of available and used session slots of this service in this specific zone.
PUT	Change the number of available session slots of this service in this zone.
DELETE	Remove all active slots from this service in this zone (i.e., terminate service instances in this zone).

7.1.4 User management protocols

Below a list of REST protocols for managing users.

/1.0/users	
GET	Returns a (filtered) list of all registered users in this domain.
POST	Register a new user in this domain.
DELETE	Remove all users from this domain.

/1.0/users/<userID>	
GET	Fetch detailed information of this user.
DELETE	Remove this user from this domain (possibly terminating all running instances first).

7.2 Zone manager protocols

This section provides a list of all current public zone manager protocols, grouped per resource type.

7.2.1 Service management protocols

Below a list of REST protocols for managing services and their slots in a zone.

/1.0/services	
GET	Returns a (filtered) list of all registered services in this zone, along with their current state and session slot information.
POST	Register a new service in this zone.
DELETE	Remove all services from this zone (e.g., in the context of a service provider).

/1.0/services/<serviceID>	
GET	Fetch detailed information of this service in this zone (i.e., slots, runtime state, etc.).
DELETE	Remove this service from this zone (possibly terminating all running instances first).

/1.0/services/<serviceID>/state	
GET	Fetch state information of this service in this zone.
PUT	Change service lifecycle state (e.g., register, provision, deploy, terminate) in this zone.
DELETE	Remove this service from this zone (possibly terminating all running instances first).

/1.0/services/<serviceID>/slots	
GET	Fetch detailed information of the number of available and used session slots of this service in this zone.
PUT	Change the number of available session slots of this service in this zone.
DELETE	Remove all active slots from this service in this zone (i.e., terminate service instances).

/1.0/services/<serviceID>/instances	
GET	Returns a (filtered) list of all (composite) service instances of this service in this zone, along with their current state and session slot information.
POST	Explicitly create a new instance of this specific service in this zone. Typically, this also immediately starts the instance, but this could also simply be a provisioning or creation request.
DELETE	Remove all instances of this service from this zone.

7.2.2 Service evaluation and offer management protocol

Below a list of REST protocols for fetching and creating service evaluation offers as part of service placement and deployment.

/1.0/services/<serviceID>/offers	
GET	Returns a (filtered) list of all stored evaluation offers for this specific service in this zone.
POST	Create a new service evaluation offer for this service in this zone: this is the service evaluation API for the domain to make an evaluation.
DELETE	Remove all stored evaluation offers for this service in this zone (i.e., flush the cache).

/1.0/offers	
GET	Returns a (filtered) list of all stored evaluation offers of all services in this zone.
POST	Create a new service evaluation offer for a service in this zone. The serviceID must be provided in the body of the request.
DELETE	Remove all stored evaluation offers for all services in this zone (i.e., flush the cache).

/1.0/ offers/<offerID>	
GET	Fetch detailed information of a particular offer.
DELETE	Remove this offer from this zone.

7.2.3 Service instance management protocols

Below a list of REST protocols for managing service instances from in a particular zone. Note that these protocols may or may not be visible to a domain orchestrator, as a domain should mainly be concerned with services and the number of slots, and not about the number of individual instances.

/1.0/instances	
GET	Returns a (filtered) list of all (composite) service instances in this zone, along with their current state and session slot information.
POST	Explicitly create a new service instance in this zone. Typically, this also immediately starts the instance, but this could also simply be a provisioning or creation request.
DELETE	Remove all service instances from this zone (e.g., in the context of a service provider).

/1.0/instances/<instanceID>	
GET	Fetch detailed information of this high-level service instance (i.e., session slots, runtime state, etc.).
DELETE	Terminate and remove this service instance from this zone.

/1.0/instances/<instanceID>/state	
GET	Fetch state information of this service instance in this zone.
PUT	Change lifecycle state of the service instance (e.g., provision, deploy, terminate).
DELETE	Remove this service instance from the zone (possibly terminating the instance first).

/1.0/instances/<instanceID>/slots	
GET	Fetch detailed information of the number of available and used session slots of this service instance.
PUT	Change the number of available session slots of this service instance.
DELETE	Remove all active slots from this instance.

7.2.4 General management protocols

Below a list of admin REST protocols for managing users, domain and DCA.

/1.0/domain	
GET	Fetch detailed information on the domain that is currently registered to this zone.
PUT	Register a domain to this zone. In the current model, we only support a zone to be part of one domain.
DELETE	Detach and remove this domain from this zone. This should only be possible if no services are active anymore from this domain.

/1.0/dca	
GET	Fetch detailed information on the DCA that is currently registered to this zone.
PUT	Register a DCA to this zone. In the current model, we only support a zone to be deployed on top of a single DC/DCA (for locality).
DELETE	Detach and remove this DCA from this zone. This should only be possible if no instances are active anymore from this DCA.

/1.0/users	
GET	Returns a (filtered) list of all registered users in this zone.
POST	Register a new user in this zone.
DELETE	Remove all users from this zone.

/1.0/users/<userID>	
GET	Fetch detailed information of this user.
DELETE	Remove this user from this zone. Users can only be removed when all their state is removed first.

7.3 DC adaptor protocols

This section covers all current DCA protocols, grouped per resource type.

7.3.1 Zone management protocols

Below a list of REST protocols for managing zones on top of a DCA. As can be observed, we allow for multiple zones to be deployed on top of the same DCA instance, though not all DCAs are required to support this multi-tenant model.

/1.0/zones	
GET	Returns a (filtered) list of all registered zones in this DCA.
POST	Add a new zone to this DCA.
DELETE	Remove all registered zones from this DCA.

/1.0/zones/<zoneID>	
GET	Fetch detailed information of this zone in this DCA (i.e., instances, slots, state, etc.).
DELETE	Remove this zone from this DCA (terminating all running instances first).

7.3.2 Service instance management protocols

Below a list of REST protocols for managing service component instances from particular zones in this DCA.

/1.0/instances	
GET	Returns a (filtered) list of all service component instances in this DCA, along with their current state and session slot information.
POST	Create a new service component instance in this DCA. Typically, this also immediately starts the instance, but this could also simply be a provisioning or creation request.
DELETE	Remove all instances from this DCA (e.g., in the context of a zone).

/1.0/instances/<instanceID>	
GET	Fetch detailed information of this service component instance (i.e., session slots, runtime state, etc.).
DELETE	Terminate and remove this instance from this DCA.

/1.0/instances/<instanceID>/state	
GET	Fetch state information of this service component instance in this DCA.
PUT	Change lifecycle state of the instance (e.g., provision, deploy, terminate).
DELETE	Remove this instance from the DCA (possibly terminating the instance first).

/1.0/instances/<instanceID>/configs	
GET	Returns a list of all service configurations associated with this instance.
POST	Add a new service configuration to this instance, with proper service instantiation parameters.
DELETE	Remove all service configurations from this instance.

/1.0/instances/<instanceID>/configs/<configID>	
GET	Fetch detailed information a particular service configuration associated with this instance.
DELETE	Remove this service configuration from the instance, possibly terminating all active sessions first.

/1.0/instances/<instanceID>/ports	
GET	Returns a (filtered) list of all port/endpoint mappings with respect to this instance. This mapping provides a translation of the private IP address and ports to the public IP address and ports.

/1.0/instances/<instanceID>/ports/<portID>	
GET	Returns the public endpoint for a specific logical application port.

7.3.3 User management protocols

Below a list of REST protocols for managing users. In practise, this consists of only the zone managers that can deploy and manage their corresponding instances in this DCA.

/1.0/users	
GET	Returns a (filtered) list of all registered users in this DCA.
POST	Register a new user in this DCA.
DELETE	Remove all users from this DCA.

/1.0/users/<userID>	
GET	Fetch detailed information of this user.
DELETE	Remove this user from this DCA (only when the corresponding zone has no running instances anymore).

7.4 FUSION service instance protocols

Below an initial list of REST protocols for communicating with particular services. We expect some additional protocols to be added later. We also envision that services are not required to implement these functions. In such case, a FUSION zone manager will try to estimate (e.g., active session slots), or depend on the service for contacting the zone manager for exchanging runtime information.

/1.0/state	
GET	Fetch current internal state information of this service instance.
PUT	Change internal service lifecycle state (e.g., start, pause, serialize, terminate, etc.) of this instance. This can be used for gracefully shutting down particular instances, (re)starting it, etc.
DELETE	Gracefully terminate this instance.

/1.0/slots	
GET	Fetch session slot information from this service instance. Note that a service instance can also push changes to its session slot information to the zone manager.
PUT	Change the number of available session slots of this service instance.
DELETE	Remove all available slots from this service instance (i.e., terminate service instance).

/1.0/configs	
GET	Returns a (filtered) list of all active service configurations associated with this service instance.
POST	Add a new service configuration to this instance, with proper instantiation parameters.
DELETE	Remove all service configurations from this service instance (possibly terminating all active sessions for these configurations first).

/1.0/configs/<configID>	
GET	Fetch detailed information of a particular service configuration.
DELETE	Remove this service configuration from this instance (possibly terminating all active sessions from this configuration first).

7.5 Evaluator service protocols

Below a list of REST protocols for requesting service evaluation requests from a zone manager to an evaluator service. Evaluator services need to implement this set of protocols.

/1.0/evaluations	
GET	Returns a (filtered) list of all stored evaluations done by this evaluator service. Note that we do not require evaluator services to store all evaluations; some can be with relatively short expiration dates (or even not stored at all).
POST	Request/create a new evaluation. This will return a (cached) score. This is the main function of the evaluator service for doing the evaluations.
DELETE	Remove all stored evaluations from this evaluator service.

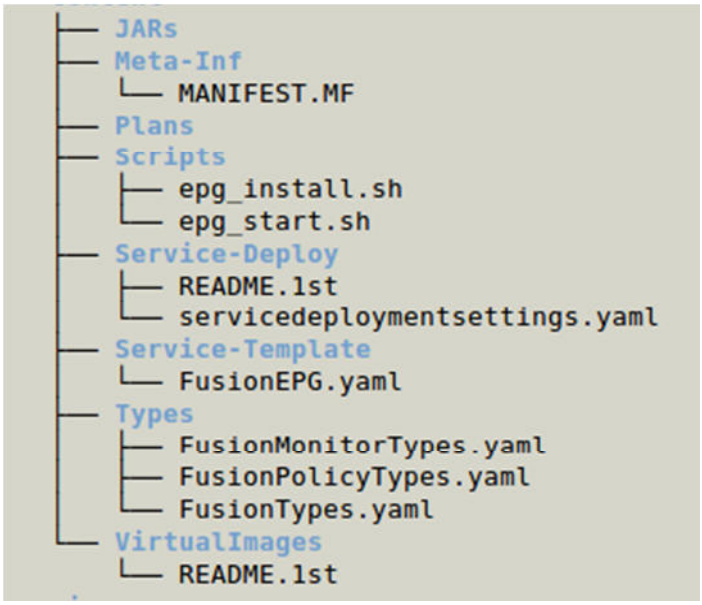
/1.0/evaluations/<evaluationID>	
GET	Fetch detailed information of a particular evaluation done by this evaluator service.
DELETE	Remove this evaluation from the evaluator service (in case the evaluation was stored).

7.6 Service manifest

7.6.1 General

Service description is delivered under the form of a CSAR (Cloud Services ARchive) The "Topology and Orchestration Specification for Cloud Applications Version 1.0, OASIS Standard, 25 November 2013" specification can be found at following link: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html

The EPG TOSCA CSAR has the following layout.



7.6.2 FusionTypes.yaml

```
tosca_definitions_version: tosca_simple_yaml_1_0
tosca.nodes.FusionAuthentication:
    derived_from: tosca.nodes.Root
    capabilities:
```

```
  # IP connection for service
  userid: string
   properties:
    secure: true
    required: true
  pass: string
   properties:
    secure: true
    required: true

tosca.nodes.FusionService:
 derived_from: tosca.nodes.SoftwareComponent

 capabilities:
  # IP connection for service
  app_endpoint: tosca.capabilities.AppEndpoint
   properties:
    secure: false
    initiator: peer
    required: true
    ports:
     userport:
      protocol: tcp
      target: integer
      # target_range: [ 5000, 5099  ]

  # IP connection info for management of the service using FUSION management protocol
  admin_port: tosca.capabilities.Endpoint
   properties:
    secure: false
    initiator: source
    required: true
    ports:
     userport:
      protocol: tcp  #http,https,ftp,tcp,udp,...
      target: 5000
      target_range: [ 5000, 5099  ]

  # the section about distributed context key value store is optional
  # 1st option is to use the link with FusionDomain and therefore associated KeyValue store and get connection info from there
  # 2nd option is to embed the info here as described below.
  # distributed context Key value store (ETCD) IP connection info
  context_endpoint: tosca.capabilities.Endpoint
   properties:
    secure: false
    initiator: source
    required: true
    ports:
     userport:
      protocol: http #rest
      target: <etcdconnection endpoint info>
   context_root:
    type: string
    required: true

  # service scaling (number of instances of this service)
  svc_scaling: tosca.capabilities.Scalable
   properties:
    min_instances: integer
     type: integer
     required: true
     default: 1
     constraints:
      - greater_or_equal: 1
    max_instances:
     type: integer
     required: true
     constraints:
      - greater_or_equal: 1

  # range of session slots
  session_slots:
   properties:
    min_slots:
     type: integer
     required: true
     default: 1
     constraints:
      - greater_or_equal: 1
    max_slots:
     type: integer
     required: true
     constraints:
      - greater_or_equal: 1
    initial_slots:
     type: integer
     required: true
     default: 1
     constraints:
      - greater_or_equal: 1

 requirements:
  - host: tosca.nodes.Compute
  - AttachTo: tosca.nodes.fusioncomponent


# component is a component part of an entire service.
# should cover composite graph functionality.
# todo: correct or complete info
tosca.nodes.fusioncomponent:
 derived_from: tosca.nodes.softwarecomponent
 capabilities:
  # ip connection for service
  app_endpoint: tosca.capabilities.appendpoint
   properties:
    secure: false
    initiator: peer
    ports:
     userport:
      protocol: tcp
      target: integer
      # target_range: [ 5000, 5099  ]

  # ip connection info for management of the service using fusion management protocol
  admin_port: tosca.capabilities.endpoint
   properties:
    secure: false
    initiator: source
    ports:
     userport:
      protocol: tcp  #http,https,ftp,tcp,udp,...
      target: 5000
      target_range: [ 5000, 5099  ]

  # distributed context key value store (etcd) ip connection info
  context_endpoint: tosca.capabilities.endpoint
   properties:
    secure: false
    initiator: source
    ports:
     userport:
      protocol: http #rest
      target: <etcdconnection endpoint info>
   context_root:
    type: string
    required: true
```

```
  # service scaling (number of instances of this service)
  svc_scaling: tosca.capabilities.scalable
   properties:
    min_instances: integer
     type: integer
     required: true
     default: 1
     constraints:
      - greater_or_equal: 1
    max_instances:
     type: integer
     required: true
     constraints:
      - greater_or_equal: 1

  # range of session slots
  session_slots:
   properties:
    min_slots:
     type: integer
     required: true
     default: 1
     constraints:
      - greater_or_equal: 1
    max_slots:
     type: integer
     required: true
     constraints:
      - greater_or_equal: 1
    initial_slots:
     type: integer
     required: true
     default: 1
     constraints:
      - greater_or_equal: 1

# todo: service component configuration parameters

 requirements:
  - host: tosca.nodes.compute
#   host:
#        # should this distinction be made here or are these different types
#     type: [ vm, container ]

# todo: correct or complete info
tosca.nodes.fusionevaluator:
 derived_from: tosca.nodes.softwarecomponent
 capabilities:
  # ip connection for service
  evaluator_endpoint: tosca.capabilities.appendpoint
   properties:
    secure: false
    initiator: peer
    ports:
     endpoint:
      type: string  # (ip@)
     userport:
      protocol: http
      target: integer

  # distributed context key value store (etcd) ip connection info
  context_endpoint: tosca.capabilities.endpoint
   properties:
    secure: false
    initiator: source
    ports:
     userport:
      protocol: http # assume rest API to retrieve evaluation metric
      target: <etcdconnection endpoint info>
    context_root:
     type: string
     required: true

   # with get_attribute function, evaluation result can be retrieved
    evaluationresult:
     type: integer / string  # did not find float...

 requirements:
  - host: tosca.nodes.compute
  - DependsOn: tosca.nodes.fusionservice (/ tosca.nodes.fusioncomponent)
  - AttachTo: tosca.nodes.fusiondomain

#   host:
#        # should this distinction be made here or are these different types
#     type: [ vm, container ]

tosca.nodes.FusionSvcContextRepo:
 derived_from: tosca.nodes.SoftwareComponent
 capabilities:
  # IP connection for service
  # distributed context Key value store (ETCD) IP connection info
  context_endpoint: tosca.capabilities.Endpoint
   properties:
    secure: false
    initiator: source
    required: true
    ports:
     userport:
      protocol: http #rest
      target: <etcdconnection endpoint info>
    context_root:
     type: string
     required: true

  # IP connection info for management of the ETCD service
  admin_port: tosca.capabilities.Endpoint
   properties:
    secure: false
    initiator: source
    required: true
    ports:
     userport:
      protocol: tcp  #http,https,ftp,tcp,udp,...
      target: 5000
      target_range: [ 5000, 5099  ]

  username: string
   properties:
    required: true
  password: string
   properties:
    required: true

 # describes to what it is linked and dependencies
 requirements:
  - host: tosca.nodes.Compute
  - AttachTo: tosca.nodes.FusionDomain

tosca.nodes.FusionDomain:
 derived_from: tosca.nodes.SoftwareComponent
 capabilities:
  # IP connection for service
  # application originated config setting (App=> fusion)
  public_domain_endpoint: tosca.capabilities.AppEndpoint
   properties:
```

```
      secure: true
      initiator: peer
      ports:
        userport:
          protocol: http # rest interface endpoint
          target: 6000
    requirements:
      - AttachTo: tosca.nodes.FusionAuthentication

  # IP connection for service
  # application configuration setting (Fusion ==> App)
  public_domain_endpoint: tosca.capabilities.AppCfgEndpoint
    properties:
      secure: true
      initiator: peer
      ports:
        userport:
          protocol: http # rest interface endpoint
          target: 6000
    requirements:
      - AttachTo: tosca.nodes.FusionAuthentication

  # IP connection info for management of the service using
FUSION management protocol
    admin_port: tosca.capabilities.Endpoint
      properties:
        secure: false
        initiator: source
        ports:
          userport:
            protocol: http  #http,https,ftp,tcp,udp,...
            target: 5000

    # either include key value store like this or user AttachTo
    # distributed context Key value store (ETCD) IP connection
info
    servicecontextrepo_endpoint: tosca.capabilities.Endpoint
      properties:
        secure: false
        initiator: source
        ports:
          userport:
            protocol: http #rest
            target: <etcdconnection endpoint info>
      context_root:
        type: string
        required: true

  # describes to what it is linked and dependencies
  requirements:
    - host: tosca.nodes.Compute
    - AttachTo: tosca.nodes.FusionAuthentication
    - AttachTo: tosca.nodes.FusionSvcContextRepo
```

7.6.3 FusionMonitorTypes.yaml

```
tosca_definitions_version: tosca_simple_yaml_1_0

monitoring_types:
# --- types proposed as part of TOSCA-189
# --- Begin
# Metric base type
  tosca.monitoring.Metric:
    derived_from: tosca.nodes.Root
    description: The basic metric type all other TOSCA metric
types derive from
    properties:
      polling_schedule:
          type: string
      return_type:
          type: string
      metric_unit:
          type: string
      aggregation_method:
          type: string
          constraints:
            - valid_values: [SUM, AVG, MIN, MAX, COUNT]

# A single metric sample
  tosca.monitoring.MetricSample:
    derived_from: tosca.monitoring.Metric
    description: A single metric sample,application KPI, like CPU,
MEMORY, etc.
    properties:
      node_state:
          type: string
          constraints:
            - valid_values: [RUNNING, CREATING, STARTING,
TERMINATING, ..]
    requirements:
          #a sample metric requires an endpoint
      - endpoint: tosca.capabilities.Endpoint

#An aggregated metric
  tosca.monitoring.AggregatedMetric:
      derived_from: tosca.monitoring.Metric
      description: An aggregated metric
      properties:
        # The time window in millis for aggregating the metric
        msec_window:
            type: integer
            constraints:
              - greater_than: 0
      requirements:
        - basedonmetric: tosca.monitoring.Metric
# --- End
# --- types proposed as part of TOSCA-189

#
# session slot metrics
  tosca.monitoring.SessionSlotMetric:
    derived_from: tosca.nodes.Root
    description: reporting of SessionSlotInformation (resource
and )
    properties:
      slots:
        type: integer
        constraints:
          - greater_or_equal: 0
    requirements:
      - basedon: tosca.nodes.FusionComponent


relationship_types:
# --- types proposed as part of TOSCA-189
# --- Begin

# a relationship between sample and endpoint
  tosca.relationships.monitoring.EndPoint:
      short_name: endpoint
      derived_from: tosca.relationships.Root
      valid_targets: [ tosca.nodes.FusionComponent ]
```

```
#this is a relationship to enforce that aggregated metric is based
on other sample/aggregate metric
   tosca.relationships.monitoring.BasedOnMetric:
    short_name: basedonmetric
    derived_from: tosca.relationships.DependsOn
    valid_targets:                                                [
alu.capabilities.Monitorable.MetricSample,alu.capabilities.Moni
torable.AggregatedMetric ]

# --- End
# --- types proposed as part of TOSCA-189

# a relationship between sample and endpoint
  tosca.relationships.monitoring.FusionComponent:
    short_name: fusioncomponentmonitor
    derived_from: tosca.relationships.Root
    valid_targets:      [      tosca.nodes.SessionSlotMetric      ]
```

7.6.4 FusionPolicyTypes

```
tosca_definitions_version: tosca_simple_yaml_1_0

# policies are not defined yet for the simple profile
# therefor first proposal to define tosca.policy
tosca.policy.Root:
   description: The tosca Policy type all other TOSCA base Policy
Types derive from
   requirements:
      - dependency:
        type: tosca.capabilities.feature
        lowerbound: 0
        upper_bound: unbounded
   capabilities:
    feature: tosca.capabilities.Feature

tosca.policy.FusionLateBind:
   description: fusion late binding Policy Type
   short_name: latebind
   derived_from: tosca.policy.Root
   # following allows to attach policy onto "DependsOn",
"AppliesTo" relationships.
   applies_to: tosca.relationships.Root
   properties:
      allow_latebinding: string
            constraints:
               - valid_values: [ yes, no]
      max_latency:
            type: string
      latency_unit:
            type: string
            constraints:
               - valid_values: [ USEC, MSECT]
      min_throughput:
         type: string
      throughput_unit:
            type: string
            constraints:
               - valid_values: [ Kbps, Mbps, Gbps]
     # still to discuss what valid target this policy type can attach
to.
     valid_targets: [ tosca.nodes.FusionComponent ]

tosca.policy.FusionPlacementPolicy:
   description: fusion placement Policy Type
   short_name: placementpolicy
   derived_from: tosca.policy.Root
   # following allows to attach policy onto "DependsOn",
"AppliesTo" relationships.
   applies_to: tosca.relationships.Root
   properties:
      # any fusion generial placement constraints to be placed
here.

   # still to discuss what valid target this placement policy type
can attach to.
   # options are
   # * components but these are more constraints
   # * likely at relationship between fusioncomonents
   # * fusionservice.
   valid_targets: [ tosca.relationships.DependsOn]
```

7.6.5 ServiceDeploymentSettings.yaml

```
tosca_definitions_version: tosca_simple_yaml_1_0_0

description: Template for deploying  in time

tosca.capabilities.slot:
 derived_from: tosca.capabilities.Root (ToCheck)
 msec_window:
   type: integer
   required: true
   constraints:
   - greater_or_equal: 0  # 0 indicated
 timestamp:
   type: time in day
   required: false  # optional setting
 # extend for days, weeks, years ...timestamp:

slot_immediate:
 type: tosca.capabilities.slot
 msec_window: 0

slot_1:
 type: tosca.capabilities.slot
 msec_window: 10000
 timestamp: 00:00:00
```

```
# etc.
```

7.6.6 FusionEPG.yaml

```
tosca_definitions_version: tosca_simple_yaml_1_0_0

description: Template for deploying a single service with predefined properties.

inputs:
 cpus:
   type: integer
   description: Number of CPUs for the server.
   constraints:
    - valid_values: [ 1, 2, 4, 8 ]

 maxsessionslots:
   type: integer
   description: Number of session slots for this service
   constraints:
    - in_range: [ 50, 500 ]

node_templates:
 myEPG:
   type: tosca.nodes.FusionService
   properties:  # or is it properties:
    app_endpoint: tosca.capabilities.AppEndpoint
     properties:
      ports:
       user_port:
        target: 5000
    admin_port: tosca.capabilities.Endpoint
    properties:
     secure: false
     initiator: source
     ports:
      user_port:
       protocol: tcp  #http,https,ftp,tcp,udp,...
       target: 5000
       target_range: [ 5000, 5099  ]
    context_endpoint: tosca.capabilities.Endpoint
     properties:
      ports:
       user_port:
        protocol: http #rest
        target: <etcdconnection endpoint info>
    context_root:
     type: <root for context in ETCD>
     required: true

    svc_scaling: tosca.capabilities.Scalable
     properties:
     min_instances: 1
     max_instances: 10

    session_slots:
     properties:
      min_slots: 1
      max_slots: { get_input: maxsessionslots }
      initial_slots: 10

    # demo description of scripts that are launched when described.
    interfaces:
     standard:
      create: scripts/epg_install.sh
      start: scripts/epg_start.sh

   requirements:
   - host: my_server

 myEPGevaluator:
  type: tosca.nodes.fusionevaluator
  properties:  # or is it properties:
   evaluator_endpoint: tosca.capabilities.AppEndpoint
    properties:
     ports:
      endpoint: 127.0.0.1
      userport:
       target: 5000
   context_endpoint: tosca.capabilities.Endpoint
    properties:
     ports:
      user_port:
       protocol: http #rest
       target: <etcdconnection port endpoint info>
   context_root:
    type: <root for context in ETCD>
    required: true

   evaluationresult: tosca.capabilities.Scalable

  requirements:
  - host: my_server
  - DependsOn: myEPG
  - AttachTo: my_fusiondomain

 my_server:
  type: tosca.nodes.Compute
  properties:
   # compute properties
   disk_size: 10
   num_cpus: 2
   mem_size: 4
   # host image properties
   os_arch: x86_64
   os_type: linux
   os_distribution: ubuntu
   os_version: 12.04


 my_fusionserver:
  type: tosca.nodes.Compute
  properties:
   # compute properties
   disk_size: 10
   num_cpus: 4
   mem_size: 16
   # host image properties
   os_arch: x86_64
   os_type: linux
   os_distribution: ubuntu
   os_version: 12.04

 my_fusiondomain:
  type: tosca.nodes.FusionDomain
  properties:
   public_domain_endpoint: tosca.capabilities.AppEndpoint
   properties:
    ports:
     user_port:
      target: 6080
   requirements:
    - AttachTo: my_fusiondomainauthentication

   # IP connection info for management of the service using FUSION management protocol
```

```
    admin_port: tosca.capabilities.Endpoint
      properties:
        ports:
          user_port:
            target: 5070
  requirements:
    - host: my_fusionserver
    - AttachTo: my_fusiondomainauthentication
    - AttachTo: my_fusiondomainkeyvaluestore


my_fusiondomainauthentication:
  type: tosca.nodes.FusionAuthentication:
  properties:
    userid: MY_FUSIONDOMAIN_ID_HERE
    pass: MY_FUSIONDOMAIN_HERE

my_fusiondomainkeyvaluestore:
  type: tosca.nodes.FusionSvcContextRepo:
  properties:
    context_endpoint: tosca.capabilities.Endpoint
      properties:
        ports:
          target:  127.0.0.1
        context_root: /domain1
    admin_port: tosca.capabilities.Endpoint
      properties:
        ports:
          user_port:
            target: 5090
    username: MY_ETCD_USERNAME
    password: MY_ETCD_PWD

  # describes to what it is linked and dependencies
  requirements:
    - host: tosca.nodes.Compute
    - AttachTo: my_fusiondomain

outputs:
  server_ip:
    description: The private IP@ of the provisioned server
    value: { get_property: [ my_server, ip_address ] }
```